

Samurai Clash Documentation

1. Description

This section provides an overview of 'Samurai Clash', a browser-based game. The game features two samurai characters in a duel, with the objective of taking the opponent's health within one minute.

'Samurai Clash' is made with HTML, CSS, and JavaScript. HTML structures the main content of the game, while CSS is utilized for styling. JavaScript handles the game logic, from handling character movements, collision detection, health management to timing functions.

The game is designed with straightforward controls to move and attack, using 'W', 'A', 'D', and 'Space' for player one, and the arrow keys for player two, making it easy to play.

2. Objectives

The aim of 'Samurai Clash' is creating an interactive and user-friendly game using basic web technologies.

Objectives of 'Samurai Clash':

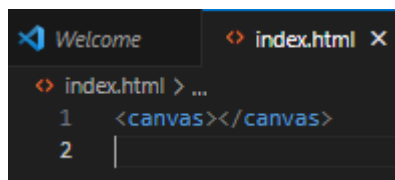
- Illustrate my practical understanding of HTML, CSS, and JavaScript through game development.
- Ensure the game is intuitive, with easy-to-understand controls and a clean user interface.
- Guarantee smooth gameplay and responsive controls.
- Use graphics and a visual theme to create an engaging aesthetic.
- Showcase abilities in programming logic and event handling on the web.

3. Design and Code

In the beginning stages of the game, one of the first steps was to set up the appearance, which is done using HTML5 Canvas. The Canvas API provides a way to draw graphics with JavaScript and the HTML `<canvas>` element.

HTML Canvas Creation:

The first file created was 'index.html', which includes a canvas element. This element acts as the placeholder for where the action unfolds.



Canvas Selection and Dimensioning:

With the canvas in place, the next step involved writing the 'index.js' file, which starts by selecting the canvas element using `document.querySelector("canvas")`. This allows the JavaScript code to reference the canvas defined in the HTML.

```

JS index.js > ...
1 // Get the canvas HTML element and its drawing context
2 const canvas = document.querySelector("canvas");
3 const c = canvas.getContext("2d");
4
5 // Set the width and height for the canvas
6 canvas.width = 1024;
7 canvas.height = 576;

```

The `.getContext("2d")` method is called on the canvas to obtain the rendering context and its drawing functions. The context, the variable `c`, is what is used to draw on the Canvas. After this, the width and height of the canvas are set to establish the area used for the game.

Moving past the static canvas, the next step was bringing the protagonists of the game, the player and the enemy, to life. Initially represented as rectangles, these entities were the foundation upon which the game was built.

Object-Oriented Approach

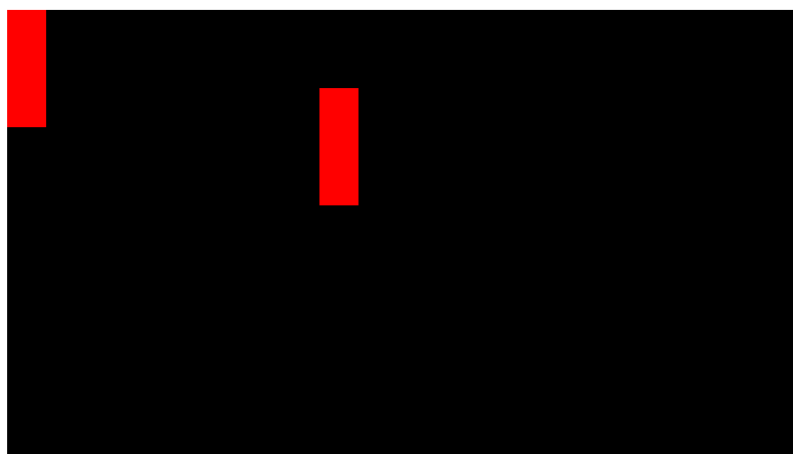
To implement these characters, object-oriented programming (OOP) was used, allowing each entity to possess unique properties and behaviors. A class named `Sprite` was created, serving as a start for the player and enemy objects.

The `Sprite` class encapsulates the qualities of the game's sprites. A constructor method within this class instantiates new sprite objects, assigning them properties such as position. Two instances of `Sprite` were created one for the player and another for the enemy and the draw method paints these sprites onto the canvas, visualizing as red rectangles. Below is the code and visual representation:

```

JS index.js > ...
1 // Get the canvas HTML element and its drawing context
2 const canvas = document.querySelector("canvas");
3 const c = canvas.getContext("2d");
4
5 // Set the width and height for the canvas
6 canvas.width = 1024;
7 canvas.height = 576;
8
9 c.fillRect(0, 0, canvas.width, canvas.height);
10
11 class Sprite {
12   constructor(position) {
13     this.position = position;
14   }
15
16   draw() {
17     c.fillStyle = "red";
18     c.fillRect(this.position.x, this.position.y, 50, 150);
19   }
20 }
21
22 const player = new Sprite({
23   x: 0,
24   y: 0,
25 });
26
27 player.draw();
28
29 const enemy = new Sprite({
30   x: 400,
31   y: 100,
32 });
33
34 enemy.draw();

```



After creating the player objects, the introduction of gravity and velocity was essential. The Sprite class was expanded to accommodate the new velocity property alongside position. This setup allows sprites to have a sense of movement along both axes.

```
11 // Class representing any drawable object in the game
12 class Sprite {
13     constructor({ position, velocity }) {
14         this.position = position;
15         this.velocity = velocity;
16     }
17
18     // Method to draw the sprite as a red rectangle on the canvas
19     draw() {
20         c.fillStyle = "red";
21         c.fillRect(this.position.x, this.position.y, 50, 150);
22     }
23
24     // Update the sprite's position and redraw it
25     update() {
26         this.draw();
27         this.position.y += 10;
28     }
29 }
30
```

To simulate continuous motion, the animate function was employed. It utilizes 'window.requestAnimationFrame' to recursively call itself, creating an animation loop.

```
31 // Create the player sprite with initial position and velocity
32 const player = new Sprite({
33     position: {
34         x: 0,
35         y: 0,
36     },
37     velocity: {
38         x: 0,
39         y: 0,
40     },
41 });
42
43 // Create the enemy sprite with initial position and velocity
44 const enemy = new Sprite({
45     position: {
46         x: 400,
47         y: 100,
48     },
49     velocity: {
50         x: 0,
51         y: 0,
52     },
53 });
54
55 function animate() {
56     // Setup the next animation frame
57     window.requestAnimationFrame(animate);
58     c.fillStyle = "black";
59     c.fillRect(0, 0, canvas.width, canvas.height);
60     console.log("goo");
61     player.update();
62     enemy.update();
63 }
64
65 animate();
```

Each update call within this loop renders the sprite's new position on the canvas, creating movement.

To prevent the painting effect as sprites moved, the canvas was cleared at the beginning of each animation frame using 'c.clearRect(0, 0, canvas.width, canvas.height)'. This ensures that only the current frame's rendering is visible, not the previous ones.

Gravity

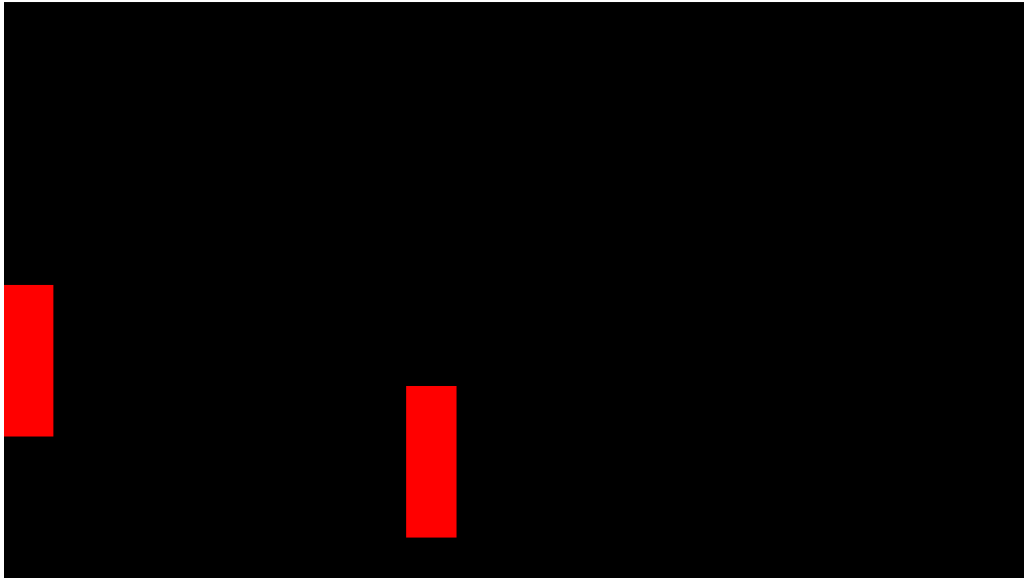
A simple yet effective approach was taken to simulate gravity, incrementing the y position of the sprites over time. This continuous addition in the update method mimics the pull of gravity, drawing the sprites downwards.

```

11 // Class representing any drawable object in the game
12 class Sprite {
13     constructor(( position, velocity )) {
14         this.position = position;
15         this.velocity = velocity;
16     }
17
18     // Method to draw the sprite as a red rectangle on the canvas
19     draw() {
20         c.fillStyle = "red";
21         c.fillRect(this.position.x, this.position.y, 50, 150);
22     }
23
24     // Update the sprite's position and redraw it
25     update() {
26         this.draw();
27         this.position.y += 10;
28     }
29 }
30

```

Now the rectangles will fall down over time:



Next, gravity was implemented

Gravity and Stopping Mechanism

A constant (gravity = 0.2) was defined to represent the force of gravity. This constant is added to the sprite's vertical velocity (this.velocity.y) on each frame, simulating the effect of gravity.

Sprite Class Update

The Sprite class now includes a height property. This was used to determine when the sprite has reached the ground (the bottom of the canvas).

In the update method, before adjusting the sprite's position, it checks if the sprite is about to go below the canvas's bottom edge.

If the bottom of the sprite (this.position.y + this.height) plus its current vertical velocity (this.velocity.y) would place it beyond the bottom of the canvas (canvas.height), it means the sprite would be on the ground so it sets this.velocity.y to 0 to stop it from going any further.

If the sprite has not yet reached the bottom, it increases the sprite's vertical velocity by the gravity constant, simulating acceleration due to gravity.

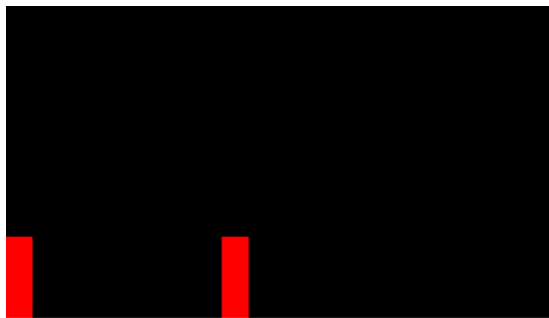
How It Works

Initially, both sprites start with a vertical velocity of 0, meaning they're not moving. As the game updates each frame, gravity is applied, increasing their downward velocity, making them fall towards the canvas floor.

Once a sprite's calculated next position would place it below the canvas floor, its vertical velocity is set to 0. This prevents it from moving any further down.

Here is the updated code and the canvas:

```
8
9 c.fillRect(0, 0, canvas.width, canvas.height);
10 const gravity = 0.2;
11 // Class representing any drawable object in the game
12 class Sprite {
13   constructor({ position, velocity }) {
14     this.position = position;
15     this.velocity = velocity;
16     this.height = 150; // Height of the sprtie to calculate ground collision
17   }
18
19   // Method to draw the sprite as a red rectangle on the canvas
20   draw() {
21     c.fillStyle = "red";
22     c.fillRect(this.position.x, this.position.y, 50, this.height);
23   }
24
25   // Update the sprite's position and redraw it
26   update() {
27     this.draw();
28
29     // Update the sprite's position based on its velocity
30     this.position.y += this.velocity.y;
31
32     // If the sprite is about to move beyond the canvas floor
33     if (this.position.y + this.height + this.velocity.y >= canvas.height) {
34       this.velocity.y = 0;
35     } else {
36       this.velocity.y += 0.2;
37     }
38   }
39 }
40
41 // Create the player sprite with initial position and velocity
42 const player = new Sprite({
43   position: {
44     x: 0,
45     y: 0,
46   },
47   velocity: {
```



Following this was integrating player movement through handling keyboard input for a responsive player experience. This was achieved through the use of event listeners to detect key presses and releases, enabling continuous movement in both directions.

```
67 // Object to track the pressed state of the keys
68 const keys = {
69   a: {
70     pressed: false,
71   },
72   d: {
73     pressed: false,
74   },
75 };
76
77 // Variable to remember the last key pressed
78 let lastKey;
79
80 // Animation loop to update game frame
81 function animate() {
82   // Setup the next animation frame
83   window.requestAnimationFrame(animate);
84   c.fillStyle = "black";
85   c.fillRect(0, 0, canvas.width, canvas.height);
86   console.log("soo");
87   player.update();
88   enemy.update();
89   player.velocity.x = 0;
90
91   // adjust player's velocity based on the pressed keys
92   if (keys.a.pressed && lastKey === "a") {
93     player.velocity.x = -1;
94   } else if (keys.d.pressed && lastKey === "d") {
95     player.velocity.x = 1;
96   }
97 }
98
99
100
101 animate();
102
103 // Event listener handle the start of the movement
104 window.addEventListener("keydown", (event) => {
105   console.log(event.key);
106   // Check which key was pressed and mark it as pressed
107   switch (event.key) {
108     case "d":
109       keys.d.pressed = true;
110       lastKey = "d";
111       break;
112     case "a":
113       keys.a.pressed = true;
114       lastKey = "a";
115       break;
116   }
117 });
118
119 // Event listener handle the stop of the movement
120 window.addEventListener("keyup", (event) => {
121   // Reset the pressed state of keys when they are released
122   switch (event.key) {
123     case "d":
124       keys.d.pressed = false;
125       break;
126     case "a":
127       keys.a.pressed = false;
128       break;
129   }
130 });
131
132
```

Implementing Right and Left movement

Initial Movement Implementation: This was done by adding event listeners for keydown and keyup events to track when the player presses and releases movement keys (specifically "a" for moving left and "d" for moving right). Movement was initially directly linked to these events, setting the player's velocity accordingly.

A problem was handling simultaneous key presses. For example, if a player pressed both "a" and "d" together, the game needed to decide which direction to take. The initial approach led to situations where the player character would stop moving if one key was released, even if the other direction key remained pressed.

To address this, a keys object was made to track the pressed state of each movement key and a ,lastKey' variable to remember the last key pressed. This allowed the game to prioritize movement in the direction of the last key pressed, ensuring correct movement and solving the issue of stopping movement incorrectly on key release.

Instead of changing the player's velocity directly in the event listeners, the actual movement logic was moved to the animate function. This made it easier to manage, as the player's velocity is updated every frame.

Adding enemy control

Previously the left and right movement were added just for the player. To control an enemy character, the keys object was modified to include ,ArrowRight' and ,ArrowLeft' keys. This change allowed for the control between the player and the enemy, enabling two players to play using the same keyboard.

```
68 // Object to track the pressed state of the keys
69 const keys = {}
70   a: {
71     pressed: false,
72   },
73   d: {
74     pressed: false,
75   },
76   ArrowRight: {
77     pressed: false,
78   },
79   ArrowLeft: {
80     pressed: false,
81   },
82 };
83
```

Event listeners for ,keydown' and ,keyup' were updated to track the state of arrow keys. This enabled the enemy character to move left and right, same as the player's movement logic but with different keys. This ensures that both characters can be moved independently.

Jumping Mechanics

Moreover, jumping for both characters was achieved by setting the velocity in the negative direction on the y-axis upon the press of "w" for the player and "ArrowUp" for the enemy. This change in velocity creates a jump effect, with gravity eventually pulling the characters down.

Here is the modified code for the enemy control and jumping mechanics for both characters:

```

101 if (keys.a.pressed && player.lastKey === "a") {
102   player.velocity.x = -5;
103 } else if (keys.d.pressed && player.lastKey === "d") {
104   player.velocity.x = 5;
105 }
106
107 // adjust enemy's velocity based on the pressed keys
108 if (keys.ArrowLeft.pressed && enemy.lastKey === "ArrowLeft") {
109   enemy.velocity.x = -5;
110 } else if (keys.ArrowRight.pressed && enemy.lastKey === "ArrowRight") {
111   enemy.velocity.x = 5;
112 }
113 }
114
115 animate();
116
117 // Event listener handle the start of the movement
118 window.addEventListener("keydown", (event) => {
119   console.log(event.key);
120   // Check which key was pressed and mark it as pressed
121   switch (event.key) {
122     case "d":
123       keys.d.pressed = true;
124       player.lastKey = "d";
125       break;
126     case "a":
127       keys.a.pressed = true;
128       player.lastKey = "a";
129       break;
130
131     case "w":
132       player.velocity.y = -20;
133       break;
134
135     case "ArrowRight":
136       keys.ArrowRight.pressed = true;
137       enemy.lastKey = "ArrowRight";
138
139       break;
140     case "ArrowLeft":
141       keys.ArrowLeft.pressed = true;
142       enemy.lastKey = "ArrowLeft";
143       break;
144
145     case "ArrowUp":
146       enemy.velocity.y = -20;
147       break;
148   }
149 });
150
151 // Event listener handle the stop of the movement
152 window.addEventListener("keyup", (event) => {
153   // Reset the pressed state of keys when they are released
154   switch (event.key) {
155     case "d":
156       keys.d.pressed = false;
157       break;
158
159     case "a":
160       keys.a.pressed = false;
161       break;
162
163     case "ArrowRight":
164       keys.ArrowRight.pressed = false;
165       break;
166
167     case "ArrowLeft":
168       keys.ArrowLeft.pressed = false;
169       break;
170   }
171 });

```

Also, unlike movement controls, the jump action does not require a keyup event to stop the action because the jump's upward motion is counteracted by gravity.

Another thing is that, to prevent the overlapping of controls between the player and the enemy, the sprite constructor was modified to include individual ,lastKey' properties for the enemy and player so the same variable isn't overwritten.

After these changes, the players can now move the characters left and right and also jump.

Adding Attack

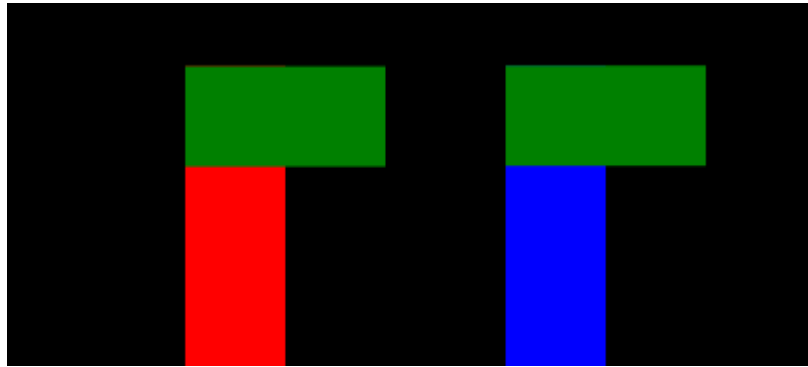
The Sprite class constructor was changed to include a new ,attackBox' property and a color parameter. The ,attackBox' represents the area where the attack action will be effective, basically the hitbox for the character's attacks. The addition of a color parameter allows for differentiating the player and the enemy.

```

12 class Sprite {
13   constructor({ position, velocity, color = "red" }) {
14     this.position = position;
15     this.velocity = velocity;
16     this.width = 50;
17     this.height = 150; // Height of the sprite to calculate ground collision
18     this.lastKey;
19
20     this.attackBox = {
21       position: this.position,
22       width: 100,
23       height: 50,
24     };
25
26     this.color = color;
27   }
28
29   // Method to draw the sprite on the canvas

```

The ,draw' method was updated as well to see the ,attackBox' alongside the character but with a different color



After this, collision detection for the player was implemented in the ,animate' function.

```
113 // Animation loop to update game frame
114 function animate() {
115     // Setup the next animation frame
116     window.requestAnimationFrame(animate);
117     c.fillStyle = "black";
118     c.fillRect(0, 0, canvas.width, canvas.height);
119     // console.log("goo");
120     player.update();
121     enemy.update();
122
123     player.velocity.x = 0;
124     enemy.velocity.x = 0;
125
126     // adjust player's velocity based on the pressed keys
127     if (keys.a.pressed && player.lastKey === "a") {
128         player.velocity.x = -5;
129     } else if (keys.d.pressed && player.lastKey === "d") {
130         player.velocity.x = 5;
131     }
132
133     // adjust enemy's velocity based on the pressed keys
134     if (keys.ArrowLeft.pressed && enemy.lastKey === "ArrowLeft") {
135         enemy.velocity.x = -5;
136     } else if (keys.ArrowRight.pressed && enemy.lastKey === "ArrowRight") {
137         enemy.velocity.x = 5;
138     }
139
140     //detect for collision
141     if (
142         player.attackBox.position.x + player.attackBox.width >= enemy.position.x &&
143         player.attackBox.position.x <= enemy.position.x + enemy.width &&
144         player.attackBox.position.y + player.attackBox.height >= enemy.position.y &&
145         player.attackBox.position.y <= enemy.position.y + enemy.height &&
146         this.isAttacking
147     ) {
148         console.log("hit");
149     }
150 }
151
152 animate();
```

The logic checks if the attacking character's attackBox overlaps with the target character's area.

It evaluates several conditions to determine a hit:

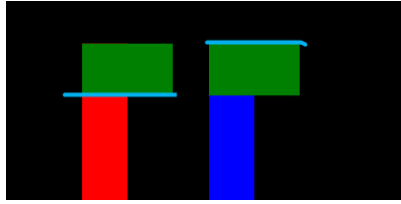
- The right side of the ,attackBox' must be greater than or equal to the left side of the target.



- The left side of the ,attackBox' must be less than or equal to the right side of the target.



- The bottom of the ,attackBox' must be greater than or equal to the top of the target. (In a 2D coordinate system in games, the y-value increases as you go down the screen so "greater than or equal to the top of the target" means the ,attackBox' has moved down enough to touch or overlap the target from above)



- The top of the ,attackBox' must be less than or equal to the bottom of the target.

If all these conditions are met, it indicates that the attackBox is overlapping with the target, signifying a hit.

After this initial setup for the attacks, some improvements have been added.

Improvement of Attack Mechanism

Dynamic Attack Box Positioning: The attack box now updates its position dynamically with the movement of the sprite. Besides this, an offset property was added. This will help for the ,attackBox' to be placed correctly, for example the ,attackBox' for the enemy must face the player, and this wouldnt be possible without it accordingly.

```

11 // Class representing any drawable object in the game
12 class Sprite {
13     constructor({ position, velocity, color = "red", offset }) {
14         this.position = position;
15         this.velocity = velocity;
16         this.width = 50;
17         this.height = 150; // Height of the sprtie to calculate ground collision
18         this.lastKey;
19
20         this.attackBox = {
21             position: {
22                 x: this.position.x,
23                 y: this.position.y,
24             },
25             offset,
26             width: 100,
27             height: 50,
28         };
29
30         this.color = color;
31         this.isAttacking;
32     }

```

Conditional Attack Box Rendering: The attack mechanism within the Sprite class has been changed to provide a better interaction. Initially, the constructor was build to include the ,attackBox' property. After this, the ,draw' method was enhanced to conditionally render the ,attackBox' only during an attack, making an impact on the visual representation . This also prevents confusion during gameplay by only displaying the hitbox when relevant.

```

34 // Method to draw the sprite as a red rectangle on the canvas
35 draw() {
36     c.fillStyle = this.color;
37     c.fillRect(this.position.x, this.position.y, this.width, this.height);
38
39     //attack box
40     if (this.isAttacking) {
41         c.fillStyle = "green";
42         c.fillRect(
43             this.attackBox.position.x,
44             this.attackBox.position.y,
45             this.attackBox.width,
46             this.attackBox.height
47         );
48     }
49 }

```

Collision: In the ,animate' function, collision detection logic was moved into a function called ,rectangularCollision', improving the process of determining a hit.

```

133
134 // Checks for a succesfull hit
135 function rectangularCollision({ rectangle1, rectangle2 }) {
136     return (
137         rectangle1.attackBox.position.x + rectangle1.attackBox.width >=
138         rectangle2.position.x &&
139         rectangle1.attackBox.position.x <=
140         rectangle2.position.x + rectangle2.width &&
141         rectangle1.attackBox.position.y + rectangle1.attackBox.height >=
142         rectangle2.position.y &&
143         rectangle1.attackBox.position.y <= rectangle2.position.y + rectangle2.height
144     );
145 }
146

```

And this is how its added into the ,update' function:

```

51 // Update the sprite's position and redraw it
52 update() {
53     this.draw();
54
55     // Updating the attackbox position
56     this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
57     this.attackBox.position.y = this.position.y;
58
59     this.position.x += this.velocity.x;
60
61     // Update the sprite's position based on its velocity
62     this.position.y += this.velocity.y;
63
64     // If the sprite is about to move beyond the canvas floor
65     if (this.position.y + this.height + this.velocity.y >= canvas.height) {
66         this.velocity.y = 0;
67     } else {
68         this.velocity.y += gravity;
69     }
70 }
71
72 attack() {
73     this.isAttacking = true;
74     console.log(this.isAttacking);
75     setTimeout(() => {
76         this.isAttacking = false;
77         console.log(this.isAttacking);
78     }, 100);
79 }
80

```

Hotkeys for Attack Execution: Now the ,space' for the player and ,ArrowDown' for the enemy keys will provide players with an attack command. This ensures the attack action is a deliberate, rather than a constant collision mistake.

```

201 // Event listener handle the start of the movement
202 window.addEventListener("keydown", (event) => {
203     console.log(event.key);
204     // Check which key was pressed and mark it as pressed
205     switch (event.key) {
206         case "d":
207             keys.d.pressed = true;
208             player.lastKey = "d";
209             break;
210         case "a":
211             keys.a.pressed = true;
212             player.lastKey = "a";
213             break;
214         case "w":
215             player.velocity.y = -20;
216             break;
217         case " ":
218             console.log("attack");
219             player.attack();
220             break;
221         case "ArrowRight":
222             keys.ArrowRight.pressed = true;
223             enemy.lastKey = "ArrowRight";
224             break;
225         case "ArrowLeft":
226             keys.ArrowLeft.pressed = true;
227             enemy.lastKey = "ArrowLeft";
228             break;
229         case "ArrowUp":
230             enemy.velocity.y = -20;
231             break;
232         case "ArrowDown":
233             console.log("attack");
234             enemy.attack();
235             break;
236     }
237 });
238

```

Attack Window: The attack box will appear for a brief moment, simulating an attack motion. This is achieved by setting ,isAttacking' to true upon pressing the attack hotkey and then using setTimeout to revert it to false.

```

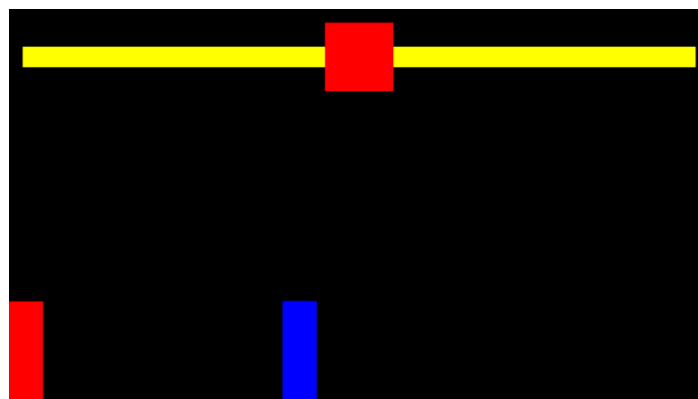
50
51 // Update the sprite's position and redraw it
52 update() {
53   this.draw();
54
55   // Updating the attackbox position
56   this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
57   this.attackBox.position.y = this.position.y;
58
59   this.position.x += this.velocity.x;
60
61   // Update the sprite's position based on its velocity
62   this.position.y += this.velocity.y;
63
64   // If the sprite is about to move beyond the canvas floor
65   if (this.position.y + this.height + this.velocity.y >= canvas.height) {
66     this.velocity.y = 0;
67   } else {
68     this.velocity.y += gravity;
69   }
70 }
71
72 attack() {
73   this.isAttacking = true;
74   console.log(this.isAttacking);
75   setTimeout(() => {
76     this.isAttacking = false;
77     console.log(this.isAttacking);
78   }, 100);
79 }
80

```

These changes to the attack system have significantly improved gameplay, making combat interactions more engaging. Players will now have more control over when and how to attack.

Designing the health bar

The 'index.html' was modified to accomodate the health bar which looks like this:



This is the code:

```

1  index.html > body > script
2  <head>
3    <style>
4      /* Ensures that padding and borders are included in the total width and height of elements */
5      * {
6        box-sizing: border-box;
7      }
8    </style>
9  </head>
10 <body>
11   <!-- Container div wraps the entire game interface including the health bars and the canvas -->
12   <div style="position: relative; display: inline-block">
13     <!-- div that contains the health bars and the timer, positioned absolutely to overlay the canvas -->
14     <div
15       style="
16         position: absolute;
17         display: flex;
18         width: 100%;
19         align-items: center;
20         padding: 20px; /* Adds space around the health bars and timer for visual spacing */
21       "
22     >
23       <!-- Player's health bar, taking up the full available width initially -->
24       <div style="background-color: yellow; height: 30px; width: 100%;></div>
25       <!-- Timer square in the center, with a fixed size and no flex shrink to maintain its size -->
26       <div
27         style="
28           background-color: red;
29           width: 100px;
30           height: 100px;
31           flex-shrink: 0; /* Prevents the timer from shrinking when other elements grow or shrink */
32         "
33       ></div>
34       <!-- Enemy's health bar, also taking up the full width available initially -->
35       <div
36         id="enemyHealth"
37         style="background-color: yellow; height: 30px; width: 100%"
38       ></div>
39     </div>
40     <!-- The game canvas where the actual game is rendered -->
41     <canvas></canvas>
42   </div>
43   <!-- Link to the game's JavaScript file -->
44   <script src="index.js"></script>
45 </body>

```

And the changes are as follows

- The outer div sets the context and contains the entire game interface. It uses `,display: inline-block'` to ensure it only takes up as much width as its children, preventing it from stretching across the entire width of the browser window.
- The inner div is absolutely positioned, allowing it to overlay the canvas. This div holds the health bars and the timer. It uses `,display: flex'` to align its child elements horizontally and `,align-items: center'` to vertically center these children within the parent div. Padding is applied for visual spacing.
- The player's health bar is styled with a yellow background and is set to take the full width of its container initially. The width will change as the player's health decreases.
- The timer is represented by a red square. `,flex-shrink: 0'` ensures that the timer does not resize when other elements change size.
- The enemy's health bar is similar to the player's, with the same initial full width and yellow background. Its width will also dynamically change based on the enemy's health status.

Refinements added to the enemy and player health bar

Players Health Bar: A container div is created with `,position: relative'` to be a reference point for the absolutely positioned child div that represents the player's health bar. The `,justify-content: flex-end'` style ensures that the contents are aligned to the right end of the container.

Yellow Bar: Serves as the background indicating the total health capacity of the player.

Blue Bar (#playerHealth): Positioned absolutely to overlay the yellow bar, this represents the current health of the player. As the player takes damage, the width of this blue bar will decrease to visually represent the loss of health.

```
<!-- Players health bar, taking up the full available width initially -->
<div
  style="
    position: relative;
    height: 30px;
    width: 100%;
    display: flex;
    justify-content: flex-end;
  "
>
  <div style="background-color: yellow; height: 30px; width: 100%"></div>
  <div
    id="playerHealth"
    style="
      position: absolute;
      background: blue;
      top: 0;
      right: 0;
      bottom: 0;
      width: 100%;
    "
  ></div>
</div>
```

Enemy's Health Bar: Initially set up similarly to the player's health bar. However, the `,left: 0'` style ensures that the blue bar's reduction starts from the left side.

```
54 <!-- Enemy's health bar, also taking up the full width available initially -->
55 <div style="position: relative; height: 30px; width: 100%">
56   <div style="background-color: yellow; height: 30px"></div>
57   <div
58     id="enemyHealth"
59     style="
60       position: absolute;
61       background: blue;
62       top: 0;
63       right: 0;
64       bottom: 0;
65       left: 0;
66     "
67   ></div>
68 </div>
69 </div>
```

Now, this is the process of implementing the health reduction:

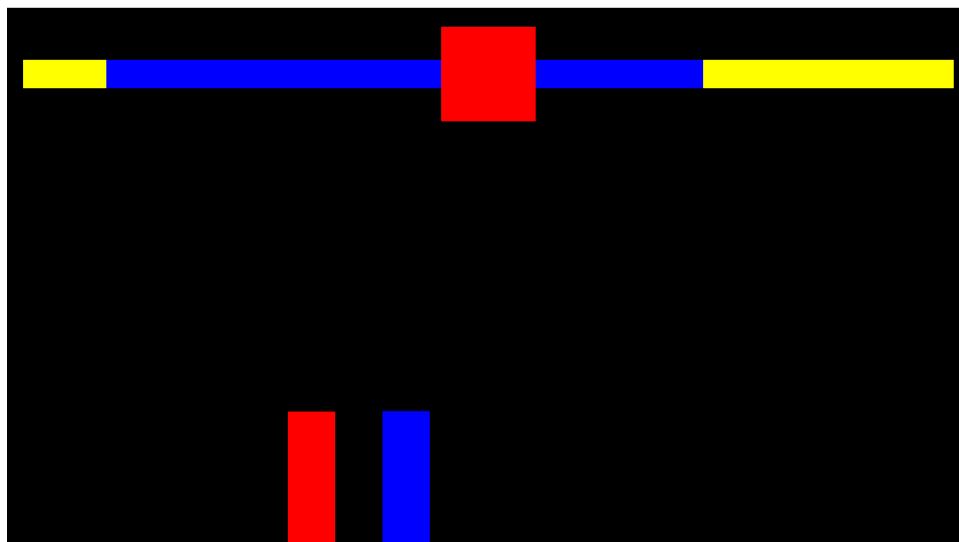
- Health Property: A health property is added to the Sprite class constructor to track the health of the sprites (player and enemy).

```
12 class Sprite {
13   constructor({ position, velocity, color = "red", offset }) {
14     this.position = position;
15     this.velocity = velocity;
16     this.width = 50;
17     this.height = 150; // Height of the sprtie to calculate ground collision
18     this.lastKey;
19
20     this.attackBox = {
21       position: {
22         x: this.position.x,
23         y: this.position.y,
24       },
25       offset,
26       width: 100,
27       height: 50,
28     };
29
30     this.color = color;
31     this.isAttacking;
32     this.health = 100;
33   }
34 }
```

- Health Decrease on Collision: When a collision is detected during an attack, the target sprite's health property is reduced by an amount and the ,style.width' of the health bar (#enemyHealth or #playerHealth) is updated to match the new health value.

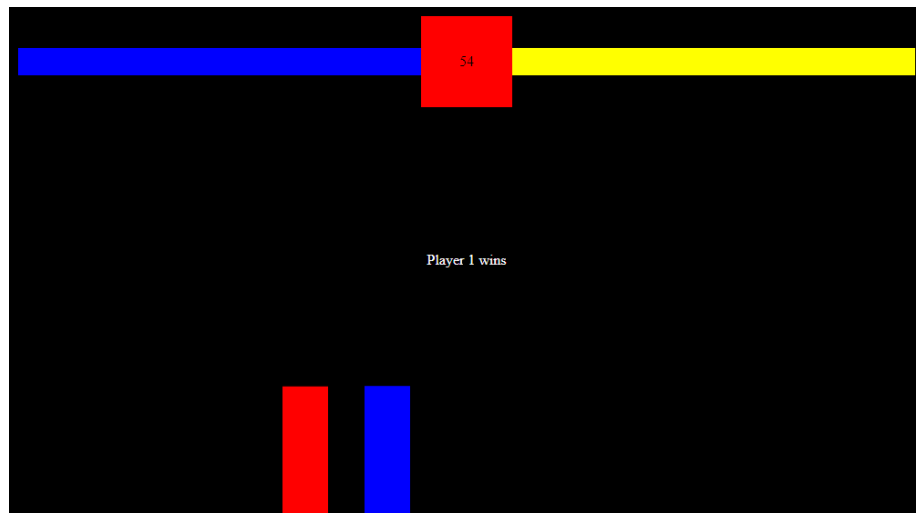
```
175 //detect for collision between player and enemy
176 if (
177   rectangularCollison({
178     rectangle1: player,
179     rectangle2: enemy,
180   }) &&
181   player.isAttacking
182 ) {
183   player.isAttacking = false;
184   enemy.health -= 20;
185   document.querySelector("#enemyHealth").style.width = enemy.health + "%";
186 }
187
188 //detect for collision between enemy and player
189 if (
190   rectangularCollison({
191     rectangle1: enemy,
192     rectangle2: player,
193   }) &&
194   enemy.isAttacking
195 ) {
196   player.health -= 20;
197   enemy.isAttacking = false;
198   document.querySelector("#playerHealth").style.width = player.health + "%";
199   console.log("hit");
200 }
201 }
202 }
```

This is how the healthbar will look after a few hits:



Adding a Game Timer and Displaying the Winner

To introduce a time constraint, a countdown timer was added to the central red square in the user interface. The countdown begins at 60 seconds and decrements each second. When the timer reaches zero, the game checks the health of both the player and the enemy to determine the winner. The winner is then displayed in the center of the screen, replacing the timer.



The Structure for the Timer has a new div element with the ID #timer. This div has the text content initially set to '10', representing the countdown start value.

The ,flex-shrink: 0;' style ensures that this element maintains its size and does not adjust when other elements change their sizes.

```
45 <!-- Timer square in the center, with a fixed size and no flex shrink to maintain its size -->
46 <div
47   id="timer"
48   style="
49     background-color: red;
50     width: 100px;
51     height: 100px;
52     flex-shrink: 0; /* Prevents the timer from shrinking when other elements grow or shrink */
53     display: flex;
54     align-items: center;
55     justify-content: center;
56   "
57 >
58   10
59 </div>
```

JavaScript Functions for Timer Logic:

- The ,decreaseTimer()' function decrements the countdown every second. It updates the inner HTML of the timer div to reflect the value.
- If the countdown reaches zero, the ,determineWinner()' function is called to evaluate the winner based on the remaining health of the player and enemy.
- The ,determineWinner()' function uses ,clearTimeout(timerId)' to stop the countdown, ensuring the timer does not go into negative numbers.
- The winner (or a tie) is then displayed by updating the inner HTML of a separate div with the ID #displayText, which is styled to be absolutely positioned and centered over the canvas, with white text to contrast against the game's black background.

Game Over Display Logic:

- The #displayText div is initially hidden to avoid changing the game view.
- When a winner is determined, the #displayText div's style is changed to ,display: flex;', which makes it visible, and its content is updated with the result of the game ('Tie', 'Player 1 wins', or 'Player 2 wins').

Incorporating the Timer and Display Logic into the Game:

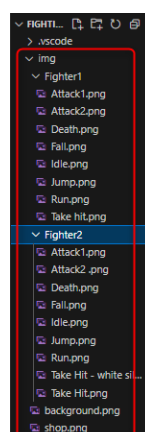
- The ,decreaseTimer()' function is invoked once when the game starts to begin the countdown.
- The game's existing collision detection logic, which decrements health upon successful attacks, now also checks the timer. If the timer reaches zero, the game transitions to the end state, showing the result and preventing further interaction.

Here you can see the added functions:

```
148 // determine the winner of the game once the timer runs out
149 function determineWinner({ player, enemy, timerId }) {
150   // Stops the timer from counting down further
151   clearTimeout(timerId);
152
153   // Shows the display text with the result of the game
154   document.querySelector("#displayText").style.display = "flex";
155
156   // Compares the health of the player and the enemy to determine the outcome
157   if (player.health === enemy.health) {
158     document.querySelector("#displayText").innerHTML = "Tie";
159   } else if (player.health > enemy.health) {
160     document.querySelector("#displayText").innerHTML = "Player 1 wins";
161   } else if (player.health < enemy.health) {
162     document.querySelector("#displayText").innerHTML = "Player 2 wins";
163   }
164 }
165
166 // Initialize the countdown timer value
167 let timer = 60;
168
169 let timerId;
170
171 // Function to decrease the game timer every second
172 function decreaseTimer() {
173   if (timer > 0) {
174     // Set up the timer to call decreaseTimer again after 1s
175     timerId = setTimeout(decreaseTimer, 1000);
176
177     timer--;
178     // Update the timer display on the screen.
179     document.querySelector("#timer").innerHTML = timer;
180   }
181
182   if (timer === 0) {
183     determineWinner({ player, enemy, timerId });
184   }
185 }
186
187 // Start the timer countdown
188 decreaseTimer();
189
```

Adding Background and Shop Sprites

Two sets of assets were taken from itch.io to enhance the visuals of the game, the background and shop sprite from Brullov's Oak Woods asset pack, and fighter sprites and animations from LuizMelo's Martial Hero packs. The images were organized into an ,img' folder in the project directory.



Class Design and Refactoring

The existing Sprite class was changed to represent static image sprites such as the background and shop, while a new Fighter class (a copy of the original Sprite class) was created for the fighters. This separation simplifies the code. The Sprite class, for instance, no longer contains methods for attack or health management, because they aren't useful for static images.

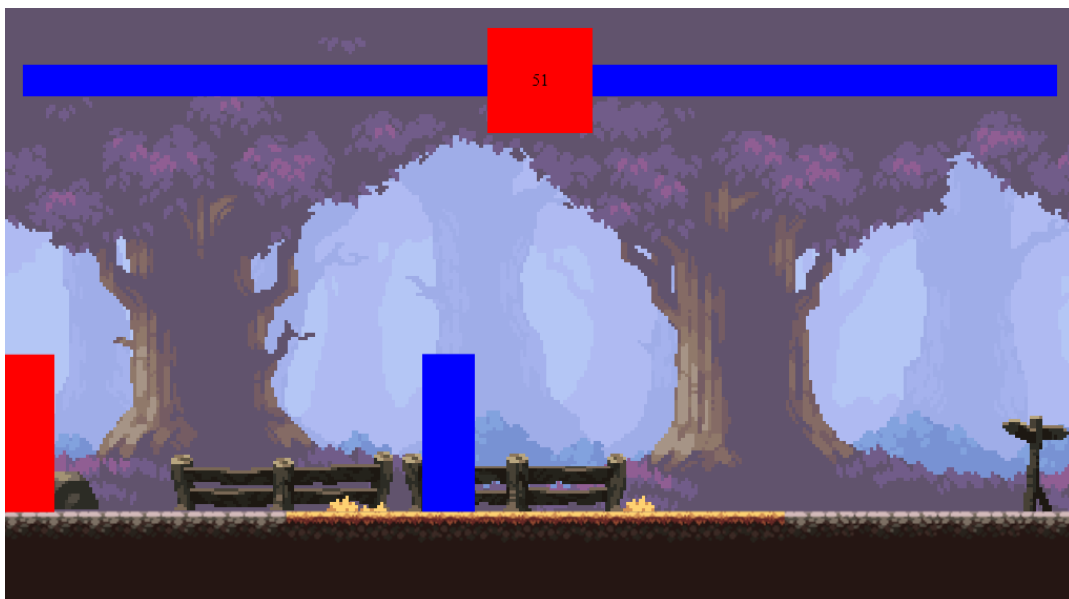
Implementation in Code

The Sprite class underwent modifications to include an 'imageSrc' parameter in its constructor, allowing for the instantiation of image objects that can be drawn onto the canvas using the 'drawImage' method.

An important modification in the Fighter class's update method ensures that the fighters don't fall past the visual ground in the background image. This was achieved by adjusting the condition that uses the gravity, stopping the fighters' downward movement just above the ground.

```
62 update() {
63   this.draw();
64
65   // Updating the attackbox position
66   this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
67   this.attackBox.position.y = this.position.y;
68
69   this.position.x += this.velocity.x;
70
71   // Update the sprite's position based on its velocity
72   this.position.y += this.velocity.y;
73
74   // If the sprite is about to move beyond the background floor
75   if (this.position.y + this.height + this.velocity.y >= canvas.height - 96) {
76     this.velocity.y = 0;
77   } else {
78     this.velocity.y += gravity;
79   }
80 }
81
82 attack() {
83   this.isAttacking = true;
84   console.log(this.isAttacking);
85   setTimeout(() => {
86     this.isAttacking = false;
87     console.log(this.isAttacking);
88   }, 100);
89 }
90
```

This is how it looks:



Moreover, Both classes were moved to a new file, 'classes.js', to separate them from the main game file and they can be seen below:


```

1 // Class representing the images
2 class Sprite {
3   constructor({ position, imageSrc }) {
4     this.position = position;
5     this.width = 50;
6     this.height = 150;
7     this.image = new Image();
8     this.image.src = imageSrc;
9   }
10
11   draw() {
12     c.drawImage(this.image, this.position.x, this.position.y);
13   }
14
15   update() {
16     this.draw();
17   }
18 }
19
20 // Class representing the fighters in the game
21 class Fighter {
22   constructor({ position, velocity, color = "red", offset }) {
23     this.position = position;
24     this.velocity = velocity;
25     this.width = 50;
26     this.height = 150; // Height of the sprite to calculate ground collision
27     this.lastKey;
28
29     this.attackBox = {
30       position: {
31         x: this.position.x,
32         y: this.position.y,
33       },
34       offset,
35       width: 100,
36       height: 50,
37     };
38
39     this.color = color;
40     this.isAttacking;
41     this.health = 100;
42   }
43
44   // Method to draw the sprite as a red rectangle on the canvas
45   draw() {
46     c.fillStyle = this.color;
47     c.fillRect(this.position.x, this.position.y, this.width, this.height);
48
49     // attack box
50     if (this.isAttacking) {
51       c.fillStyle = "green";
52       c.fillRect(
53         this.attackBox.position.x,
54         this.attackBox.position.y,
55         this.attackBox.width,
56         this.attackBox.height
57       );
58     }
59   }
60
61   // Update the sprite's position and redraw it
62   update() {
63     this.draw();
64
65     // Updating the attackBox position
66     this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
67     this.attackBox.position.y = this.position.y;
68
69     this.position.x += this.velocity.x;
70
71     // Update the sprite's position based on its velocity
72     this.position.y += this.velocity.y;

```

```

69     this.position.x += this.velocity.x;
70
71     // Update the sprite's position based on its velocity
72     this.position.y += this.velocity.y;
73
74     // If the sprite is about to move beyond the canvas floor
75     if (this.position.y + this.height + this.velocity.y >= canvas.height - 96) {
76       this.velocity.y = 0;
77     } else {
78       this.velocity.y += gravity;
79     }
80
81     attack() {
82       this.isAttacking = true;
83       console.log(this.isAttacking);
84       setTimeout(() => {
85         this.isAttacking = false;
86         console.log(this.isAttacking);
87       }, 100);
88     }
89   }
90 }

```

Also, utility functions like collision detection, determining the winner, and the timer were placed in a ,utils.js' file.

Next, the shop sprite has been created and it contains 6 smaller shop frames. For this the sprite function has been modified to be able to clip the 6 photos and give an animation effect.



Here is how the sprite class has been changed:

```

1 // Class representing the images
2 class Sprite {
3   constructor({ position, imageSrc, scale = 1, framesMax = 1 }) {
4     this.position = position;
5     this.width = 50;
6     this.height = 150;
7     this.image = new Image();
8     this.image.src = imageSrc;
9     this.scale = scale;
10    this.framesMax = framesMax;
11    this.framesCurrent = 0;
12    this.framesElapsed = 0;
13    this.framesHold = 10; // How many frames to hold before transitioning to the next frame
14  }
15
16  draw() {
17    c.drawImage(
18      this.image,
19      this.framesCurrent * (this.image.width / this.framesMax), // The x coordinate where to start clipping
20      0, // The y coordinate where to start clipping
21      this.image.width / this.framesMax, // The width of the clipped image
22      this.image.height, // The height of the clipped image
23      this.position.x, // The x position where to place the image on the canvas
24      this.position.y, // The y position where to place the image on the canvas
25      (this.image.width / this.framesMax) * this.scale, // The width of the image to be drawn on the canvas, scaled
26      this.image.height * this.scale // The height of the image to be drawn on the canvas, scaled
27    );
28  }
29
30  update() {
31    this.draw();
32
33    this.framesElapsed++;
34    if (this.framesElapsed % this.framesHold === 0) {
35      if (this.framesCurrent < this.framesMax - 1) {
36        this.framesCurrent = this.framesCurrent + 1; // Move to the next frame
37      } else this.framesCurrent = 0; // Reset to the first frame
38    }
39  }
40 }

```

,c.drawImage()' in the ,draw' method:

The *,c.drawImage'* method can take up to nine arguments. The first three are straightforward:

- *this.image*: The source image (the sprite sheet).
- *this.framesCurrent * (this.image.width / this.framesMax)*: The X coordinate on the sprite where the clipping starts, which selects the current frame.
- *0*: The Y coordinate on the sprite sheet where the clipping starts (assuming all frames are on the same horizontal line).

The next two arguments define the size of the frame to be clipped from the sprite sheet:

- *this.image.width / this.framesMax*: The width of each frame on the sprite sheet.
- *this.image.height*: The height of each frame (assuming the entire height of the image is one frame).

The last four arguments define where and how large the clipped frame will be drawn on the canvas:

- *this.position.x*: The X coordinate on the canvas where the image will be drawn.
- *this.position.y*: The Y coordinate on the canvas where the image will be drawn.
- *(this.image.width / this.framesMax) * this.scale*: The width of the frame as it will appear on the canvas, scaled by *this.scale*.
- *this.image.height * this.scale*: The height of the frame as it will appear on the canvas, also scaled by *this.scale*.

Update method

The *,update'* method handles the logic for changing frames to animate the sprite. It increments *,framesCurrent'* after a set number of *,framesHold'* have elapsed, looping back to the first frame after the last frame is drawn.

For everything to work the shop is created with a larger scale to fit the space in the game and with *,framesMax'* set to 6, corresponding to the number of frames in the shop sprite sheet.

```
21 //Initialize shop sprite
22 const shop = new Sprite({
23   position: {
24     x: 600,
25     y: 128,
26   },
27   imageSrc: "./img/shop.png",
28   scale: 2.75,
29   framesMax: 6,
30 });
31
```

Now the shop will have an animation that displays the smoke changing:



After this the idle movement has been added to the player. For this to be done the following changes have occurred:

- The Fighter class will extend the Sprite and can now display animations like, idle, using the draw method from Sprite, which handles the rendering.
- The Fighter class uses the super constructor to pass animation properties to the Sprite class, ensuring that fighters are initialized with the correct data, including the new offset property for positioning.
- The update method in the Sprite class now calls ,this.animateFrames()' to update the frame animation. By moving the animation logic to it avoids duplicating code.

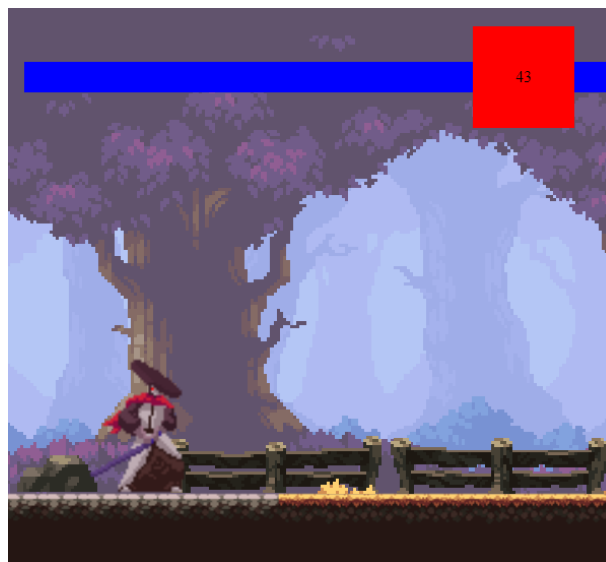
```
1 // Class representing the images
2 class Sprite {
3   constructor({
4     position,
5     imageSrc,
6     scale = 1,
7     framesMax = 1,
8     offset = { x: 0, y: 0 },
9   }) {
10    this.position = position;
11    this.width = 50;
12    this.height = 150;
13    this.image = new Image();
14    this.image.src = imageSrc;
15    this.scale = scale;
16    this.framesMax = framesMax;
17    this.framesCurrent = 0;
18    this.framesElapsed = 0;
19    this.framesHold = 5; // How many frames to hold before transitioning to the next frame
20    this.offset = offset;
21  }
22
23  draw() {
24    c.drawImage(
25      this.image,
26      this.framesCurrent * (this.image.width / this.framesMax), // The x coordinate where to start clipping
27      0, // The y coordinate where to start clipping
28      this.image.width / this.framesMax, // The width of the clipped image
29      this.image.height, // The height of the clipped image
30      this.position.x - this.offset.x, // The x position where to place the image on the canvas
31      this.position.y - this.offset.y, // The y position where to place the image on the canvas
32      (this.image.width / this.framesMax) * this.scale, // The width of the image to be drawn on the canvas, scaled
33      this.image.height * this.scale // The height of the image to be drawn on the canvas, scaled
34    );
35  }
36
37  animateFrames() {
38    this.framesElapsed++;
39    if (this.framesElapsed % this.framesHold === 0) {
40      if (this.framesCurrent < this.framesMax - 1) {
41        this.framesCurrent = this.framesCurrent + 1;
42        // console.log("Frames current ", this.framesCurrent, " Frames max ", this.framesMax)
43      } else this.framesCurrent = 0;
44    }
45  }
46
47  update() {
48    this.draw();
49    this.animateFrames();
50  }
51 }
52
53 // Class representing the fighters in the game
54 class Fighter extends Sprite {
55   constructor({
56     position,
57     velocity,
58     color = "red",
59     imageSrc,
60     scale = 1,
61     framesMax = 1,
62     offset = { x: 0, y: 0 },
63   }) {
64     super({
65       position,
66       imageSrc,
67       scale,
68       framesMax,
69       offset,
70     });
71
72     this.position = position;
73     this.velocity = velocity;
74     this.width = 50;
75     this.height = 150; // Height of the sprtie to calculate ground collision
76     this.lastKey;
77
78     this.attackBox = {
79       position: {
80         x: this.position.x,
81         y: this.position.y,
82       },
83       offset,
84       width: 100,
85       height: 50,
86     };
87
88     this.color = color;
89     this.isAttacking;
90     this.health = 100;
91     this.framesCurrent = 0;
92     this.framesElapsed = 0;
93     this.framesHold = 10; // How many frames to hold before transitioning to the next frame
94
95     // Method to draw the sprite as a red rectangle on the canvas
96
97     // Update the sprite's position and redraw it
98     update() {
99       this.draw();
100       this.animateFrames();
101     }
102   }
103 }
```

Now The ,imageSrc' property for the fighter is set to the path of the idle animation sprite sheet. The ,framesMax' is set to the number of frames in the idle animation.

```
32 // Create the player sprite with initial position and velocity
33 const player = new Fighter({
34   position: {
35     x: 0,
36     y: 0,
37   },
38   velocity: {
39     x: 0,
40     y: 10,
41   },
42
43   imageSrc: "./img/Fighter2/idle.png",
44   framesMax: 8,
45   scale: 2.5,
46   offset: {
47     x: 215,
48     y: 157,
49   },
50 });
```

The offset is necessary because the sprite sheet often includes extra transparent padding around the actual image. The offset allow to compensate for this padding and position the sprite correctly relative to the game's coordinate system. This ensures that collision detection and other game mechanics work as expected.

This is how the interface looks like



To improve the character animations in the game, running animation for the fighter was added. The process involved several modifications and additions to the existing classes.

Adding sprites Property in Fighter Constructor:

The Fighter class, which extends the Sprite class, now includes a new sprites property. This property is an object that holds the states of the fighter, like idle and running, with the specific image sources and the number of frames in the animation.

```

52 class Fighter extends Sprite {
53   constructor({
54     position,
55     velocity,
56     color = "red",
57     imageSrc,
58     scale = 1,
59     framesMax = 1,
60     offset = { x: 0, y: 0 },
61     sprites,
62   }) {
63     super({
64       position,
65       imageSrc,
66       scale,
67       framesMax,
68       offset,
69     });
70
71     this.position = position;
72     this.velocity = velocity;
73     this.width = 50;
74     this.height = 150; // Height of the sprite to calculate ground collision
75     this.lastKey;
76
77     this.attackBox = {
78       position: {
79         x: this.position.x,
80         y: this.position.y,
81       },
82       offset,
83       width: 100,
84       height: 50,
85     };
86
87     this.color = color;
88     this.isAttacking;
89     this.health = 100;
90     this.framesCurrent = 0;
91     this.framesElapsed = 0;
92     this.framesHold = 10; // How many frames to hold before transitioning to the next frame
93     this.sprites = sprites; // Initialize the sprites property with all the sprites needed for the animation
94   }

```

Initializing Sprites Object:

Within the Fighter, the sprites object is declared, containing the image sources and frame counts for both idle and running animations.

```

32 // Create the player sprite with initial position and velocity
33 const player = new Fighter({
34   position: {
35     x: 0,
36     y: 0,
37   },
38   velocity: {
39     x: 0,
40     y: 10,
41   },
42   imageSrc: "./img/Fighter2/idle.png",
43   framesMax: 8,
44   scale: 2.5,
45   offset: {
46     x: 215,
47     y: 157,
48   },
49   sprites: {
50     idle: {
51       imageSrc: "img/Fighter2/Idle.png",
52       framesMax: 8,
53     },
54     run: {
55       imageSrc: "img/Fighter2/Run.png",
56       framesMax: 8,
57     },
58   },
59 });

```

Dynamically Creating Image Properties:

To manage the animation, a loop within the Fighter class iterates over each sprite in the sprites object, dynamically creating an Image object and assigning the source.

```
96 // Loop over each sprite action and create an Image object for it
97 for (const sprite in this.sprites) {
98   sprites[sprite].image = new Image();
99   sprites[sprite].image.src = sprites[sprite].imageSrc;
100 }
101
```

Animation Switching Logic:

In the animation loop (animate function in index.js), the default image for the player is set to the idle. When the player moves, the image switches to the running one. By creating an Image object for each sprite state, and setting their sources during the initialization, it ensures that all images are loaded and ready to be rendered.

This approach:

- Pre-loads Images: Creates an image loading them into memory at the start. This provides smooth transitions.
- Manages Memory Efficiently: Stores a single instance of each Image object, avoiding redundant reloads from the source.
- Ensures Synchronization: Guarantees that sprite state changes are reflected immediately in the next cycle, as the images are already loaded.

```
118 //Default movement
119 player.image = player.sprites.idle.image;
120
121 // adjust player's velocity based on the pressed keys
122 if (keys.a.pressed && player.lastKey === "a") {
123   player.velocity.x = -5;
124   player.image = player.sprites.run.image;
125 } else if (keys.d.pressed && player.lastKey === "d") {
126   player.velocity.x = 5;
127   player.image = player.sprites.run.image;
128 }
129
```

The character now responds with a running animation when moving, and transitions back to idle when the movement stops.

After this, new sprite animations for jumping and falling were added to the game to enhance the visual feedback for the player's actions:

- The sprites object within the Fighter class will include additional states for jumping and falling, each with their respective image sources and frame count.

```
50 sprites: {
51   idle: {
52     imageSrc: "img/Fighter2/Idle.png",
53     framesMax: 8,
54   },
55   run: {
56     imageSrc: "img/Fighter2/Run.png",
57     framesMax: 8,
58   },
59   jump: {
60     imageSrc: "img/Fighter2/Jump.png",
61     framesMax: 2,
62   },
63   fall: {
64     imageSrc: "img/Fighter2/Fall.png",
65     framesMax: 2,
66   },
67 },
68 };
69
```

- The 'switchSprite' function was created in the 'Fighter' class to handle switching between animations with different frame counts and to reset the animation sequence.

```

137 switchSprite(sprite) {
138     // Checks for the sprite that matches the case
139     switch (sprite) {
140         case "idle":
141             if (this.image !== this.sprites.idle.image) {
142                 this.image = this.sprites.idle.image;
143                 this.framesMax = this.sprites.idle.framesMax;
144                 this.framesCurrent = 0; // Reset the current frame to 0 to start the animation from the beginning
145             }
146             break;
147
148         case "run":
149             if (this.image !== this.sprites.run.image) {
150                 this.image = this.sprites.run.image;
151                 this.framesMax = this.sprites.run.framesMax;
152                 this.framesCurrent = 0;
153             }
154             break;
155
156         case "jump":
157             if (this.image !== this.sprites.jump.image) {
158                 this.image = this.sprites.jump.image;
159                 this.framesMax = this.sprites.jump.framesMax;
160                 this.framesCurrent = 0;
161             }
162             break;
163         case "fall":
164             if (this.image !== this.sprites.fall.image) {
165                 this.image = this.sprites.fall.image;
166                 this.framesMax = this.sprites.fall.framesMax;
167                 this.framesCurrent = 0;
168             }
169             break;
170     }
171 }
172 }
173

```

The 'switchSprite' function is called to change the sprite image depending on the player's movement. The jump animation is played when the player is moving up while the fall animation is displayed when the player falls down.

```

126 // adjust player's velocity based on the pressed keys
127 if (keys.a.pressed && player.lastKey === "a") {
128     player.velocity.x = -5;
129     player.switchSprite("run");
130 } else if (keys.d.pressed && player.lastKey === "d") {
131     player.velocity.x = 5;
132     player.switchSprite("run");
133 } else {
134     player.switchSprite("idle");
135 }
136
137 //jumping and falling
138 if (player.velocity.y < 0) {
139     player.switchSprite("jump");
140 } else if (player.velocity.y > 0) {
141     player.switchSprite("fall");
142 }

```

Adding the Attack Animation

The attack animation frames are added to the sprites object of the Fighter class

```

50 sprites: {
51     idle: {
52         imageSrc: "img/Fighter2/Idle.png",
53         framesMax: 8,
54     },
55     run: {
56         imageSrc: "img/Fighter2/Run.png",
57         framesMax: 8,
58     },
59     jump: {
60         imageSrc: "img/Fighter2/Jump.png",
61         framesMax: 2,
62     },
63     fall: {
64         imageSrc: "img/Fighter2/Fall.png",
65         framesMax: 2,
66     },
67     attack1: {
68         imageSrc: "img/Fighter2/Attack1.png",
69         framesMax: 6,
70     },
71 },
72 });
73

```

Triggering the Attack

When the spacebar is pressed, the `,attack()` function is called. This function uses `switchSprite()` to change the fighter to the attack animation and sets `,isAttacking` to true, indicating that an attack is in progress.

```
127  
128     attack() {  
129         this.switchSprite("attack1");  
130         this.isAttacking = true;  
131         console.log(this.isAttacking);  
132         setTimeout(() => {  
133             this.isAttacking = false;  
134             console.log(this.isAttacking);  
135         }, 100);  
136     }  
137 }
```

Now the `,switchSprite()` will include a check at the beginning to prevent the attack animation from being interrupted if it's already playing:

```
138     switchSprite(sprite) {  
139         //prevents the animation from looping the attack  
140         if (  
141             this.image === this.sprites.attack1.image &&  
142             this.framesCurrent < this.sprites.attack1.framesMax - 1  
143         )  
144             return;  
145         // Checks for the sprite that matches the case  
146         switch (sprite) {  
147             case "idle":
```

This is how the attack animation will look like:



Following this was adding the animations for the enemy player as well. The process above was repeated, from copying the properties inside the player, to using the `,switchSprite()` function inside the animation loop:


```

74 // Create the enemy sprite with initial position and velocity
75 const enemy = new Fighter({
76   position: {
77     x: 400,
78     y: 100,
79   },
80   velocity: {
81     x: 0,
82     y: 0,
83   },
84   offset: {
85     x: -50,
86     y: 0,
87   },
88   imageSrc: "img/Fighter1/idle.png",
89   framesMax: 4,
90   scale: 2.5,
91   offset: {
92     x: 215,
93     y: 167,
94   },
95   sprites: {
96     idle: {
97       imageSrc: "img/Fighter1/Idle.png",
98       framesMax: 4,
99     },
100     run: {
101       imageSrc: "img/Fighter1/Run.png",
102       framesMax: 8,
103     },
104     jump: {
105       imageSrc: "img/Fighter1/Jump.png",
106       framesMax: 2,
107     },
108     fall: {
109       imageSrc: "img/Fighter1/Fall.png",
110       framesMax: 2,
111     },
112     attack1: {
113       imageSrc: "img/Fighter1/Attack1.png",
114       framesMax: 4,
115     },
116   },
117 });

```

```

158 // adjust player's velocity based on the pressed keys
159 if (keys.a.pressed && player.lastKey === "a") {
160   player.velocity.x = -5;
161   player.switchSprite("run");
162 } else if (keys.d.pressed && player.lastKey === "d") {
163   player.velocity.x = 5;
164   player.switchSprite("run");
165 } else {
166   player.switchSprite("idle");
167 }
168
169 //jumping and falling
170 if (player.velocity.y < 0) {
171   player.switchSprite("jump");
172 } else if (player.velocity.y > 0) {
173   player.switchSprite("fall");
174 }
175
176 // adjust enemy's velocity based on the pressed keys
177 if (keys.ArrowLeft.pressed && enemy.lastKey === "ArrowLeft") {
178   enemy.velocity.x = -5;
179   enemy.switchSprite("run");
180 } else if (keys.ArrowRight.pressed && enemy.lastKey === "ArrowRight") {
181   enemy.velocity.x = 5;
182   enemy.switchSprite("run");
183 } else {
184   enemy.switchSprite("idle");
185 }
186
187 //jumping and falling - enemy
188 if (enemy.velocity.y < 0) {
189   enemy.switchSprite("jump");
190 } else if (enemy.velocity.y > 0) {
191   enemy.switchSprite("fall");
192 }

```

The canvas will now showcase both of the fighters:



To continue, the attack box was correctly placed and synchronized with the animation frames. For this, several steps were taken to improve the game mechanics:

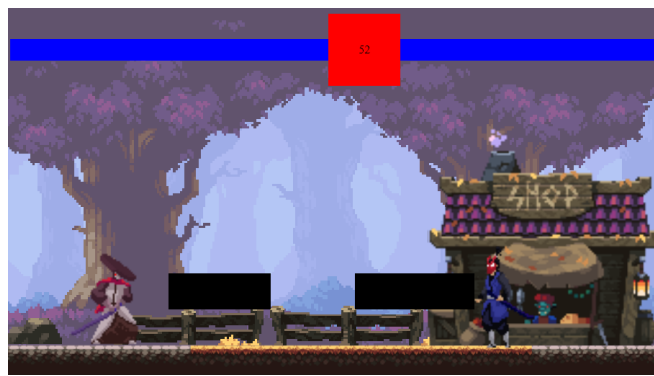
Correctly Positioning the Attack Box

Initially, to see the attack box's placement, it was drawn on the canvas using `c.fillRect()` in the update function. This helped to see where the attack box was in relation to the fighter.

```

115 //draw attack box to position it
116 c.fillRect(
117   this.attackBox.position.x,
118   this.attackBox.position.y,
119   this.attackBox.width,
120   this.attackBox.height
121 );
122

```

After that, in the Fighter class, a new property for the attack box was introduced to define its position, offset, width, and height based on the fighter's state and position.

```

13 // Class representing the fighters in the game
14 class Fighter extends Sprite {
15     constructor({
16         position,
17         velocity,
18         color = "red",
19         imageSrc,
20         scale = 1,
21         framesMax = 1,
22         offset = { x: 0, y: 0 },
23         sprites,
24     }) {
25         this.attackBox = { offset: {}, width: undefined, height: undefined },
26     },
27     super({
28         position,
29         imageSrc,
30         scale,
31         framesMax,
32         offset,
33     });
34     this.position = position;
35     this.velocity = velocity;
36     this.width = 50;
37     this.height = 150; // Height of the sprite to calculate ground collision
38     this.lastKey;
39
40     this.attackBox = {
41         position: {
42             x: this.position.x,
43             y: this.position.y,
44         },
45         offset: attackBox.offset,
46         width: attackBox.width,
47         height: attackBox.height,
48     };
49
50     this.color = color;
51     this.isAttacking;
52     this.health = 100;
53     this.framesCurrent = 0;
54     this.framesElapsed = 0;
55     this.framesHeld = 5; // how many frames to hold before transitioning to the next frame
56     this.sprites = sprites; // Initialize the sprites property with all the sprites needed for the animation
57
58     // Loop over each sprite action and create an Image object for it
59     for (const sprite in this.sprites) {
60         sprites[sprite].image = new Image();
61         sprites[sprite].image.src = sprites[sprite].imageSrc;
62     }
63 }

```

For both player and enemy, the attack box was added with offsets, widths, and heights to ensure it aligns with the attack animations.

```

83 const enemy = new Fighter({
84     position: {
85         x: 400,
86         y: 100,
87     },
88     velocity: {
89         x: 0,
90         y: 0,
91     },
92     offset: {
93         x: -50,
94         y: 0,
95     },
96     imageSrc: "img/Fighter1/Idle.png",
97     framesMax: 4,
98     scale: 2.5,
99     offset: {
100         x: 215,
101         y: 167,
102     },
103     sprites: {
104         idle: {
105             imageSrc: "img/Fighter1/Idle.png",
106             framesMax: 4,
107         },
108         run: {
109             imageSrc: "img/Fighter1/Run.png",
110             framesMax: 8,
111         },
112         jump: {
113             imageSrc: "img/Fighter1/Jump.png",
114             framesMax: 2,
115         },
116         fall: {
117             imageSrc: "img/Fighter1/Fall.png",
118             framesMax: 2,
119         },
120         attack1: {
121             imageSrc: "img/Fighter1/Attack1.png",
122             framesMax: 4,
123         },
124     },
125     attackBox: {
126         offset: {
127             x: -166,
128             y: 50,
129         },
130         width: 166,
131         height: 50,
132     },
133 });

```

Synchronizing Attack Frames

The attack's impact on health was synchronized with the frame of the attack animation where the actual hit occurs. This was done by checking if 'framesCurrent' matches the frame where the attack is displayed in the collision function.

```

210 //detect for collision between player and enemy
211 if (
212     rectangularCollision({
213         rectangle1: player,
214         rectangle2: enemy,
215     }) &&
216     player.isAttacking &&
217     player.framesCurrent === 4
218 ) {
219     player.isAttacking = false;
220     enemy.health -= 20;
221     document.querySelector("#enemyHealth").style.width = enemy.health + "%";
222 }
223

```

Also setting ,isAttacking' to false was done with a timeout and that was removed from the attack function, because the condition above couldn't be met in the span of 100ms.

This was done by adding a condition in the animation loop to set ,isAttacking' to false when the attack animation completes or misses.

```

239 //if player misses
240 if (player.isAttacking && player.framesCurrent === 4) {
241     player.isAttacking = false;
242 }
243
244 //if enemy misses
245 if (enemy.isAttacking && enemy.framesCurrent === 2) {
246     enemy.isAttacking = false;
247 }
248

```

After the ,attackBox' was correctly placed, it's representation using rectangles in the update function was removed.

Implementation of Take Hit and Death Animations

Besides the player and enemy sprites being modified to contain the animations for the ,takehit' and ,death' like it can be observed below for the enemy:

```

101 const enemy = new Fighter({
102     position: {
103         x: 400,
104         y: 300,
105     },
106     velocity: {
107         x: 0,
108         y: 0,
109     },
110     offset: {
111         x: -50,
112         y: 0,
113     },
114     imageSrc: "img/fighter/idle.png",
115     framesMax: 4,
116     scale: 2.5,
117     offset: {
118         x: 215,
119         y: 10,
120     },
121     sprites: {
122         idle: {
123             imageSrc: "img/fighter/idle.png",
124             framesMax: 4,
125         },
126         run: {
127             imageSrc: "img/fighter/run.png",
128             framesMax: 8,
129         },
130         jump: {
131             imageSrc: "img/fighter/jump.png",
132             framesMax: 2,
133         },
134         fall: {
135             imageSrc: "img/fighter/fall.png",
136             framesMax: 2,
137         },
138         attack: {
139             imageSrc: "img/fighter/attack.png",
140             framesMax: 4,
141         },
142         takeHit: {
143             imageSrc: "img/fighter/take hit.png",
144             framesMax: 1,
145         },
146         death: {
147             imageSrc: "img/fighter/death.png",
148             framesMax: 7,
149         },
150     },
151     attackBox: {
152         offset: {
153             x: -100,
154             y: 50,
155         },
156         width: 100,
157         height: 50,
158     },
159 });

```

A function named ,takeHit()' has been introduced in the Fighter class. This function deducts health from the fighter upon taking a hit and switches the sprite animation to either "takeHit" if the fighter is still alive or "death" if the fighter's health is below 0. The health deduction, which was previously not contained, is now encapsulated.

```

145 takeHit() {
146     this.health -= 20;
147
148     if (this.health <= 0) {
149         this.switchSprite("death");
150     } else this.switchSprite("takeHit");
151 }
152

```

The ,switchSprite()' method in the Fighter class has been updated to include cases for "takeHit" and "death" animations. Additional checks prevent the animation from looping indefinitely and ensure that the "death" animation only plays once by setting a new dead property to true. This property was added in the Fighter class and it's triggered when a fighter is no longer active.

```

153 switchSprite(sprite) {}
154 //if the player dies you cant continue
155 if (this.image === this.sprites.death.image) {
156   if (this.framesCurrent === this.sprites.death.framesMax - 1)
157     this.dead = true;
158   return;
159 }
160 //prevents the animation from looping the attack
161 if (
162   this.image === this.sprites.attack1.image &&
163   this.framesCurrent < this.sprites.attack1.framesMax - 1
164 )
165   return;
166
167 //override when fighter gets hit
168 if (
169   this.image === this.sprites.takeHit.image &&
170   this.framesCurrent < this.sprites.takeHit.framesMax - 1
171 )
172   return;
173 // Checks for the sprite that matches the case
174 switch (sprite) {
175   case "takeHit":
176     if (this.image !== this.sprites.takeHit.image) {
177       this.image = this.sprites.takeHit.image;
178       this.framesMax = this.sprites.takeHit.framesMax;
179       this.framesCurrent = 0;
180     }
181     break;
182   case "death":
183     if (this.image !== this.sprites.death.image) {
184       this.image = this.sprites.death.image;
185       this.framesMax = this.sprites.death.framesMax;
186       this.framesCurrent = 0;
187     }
188     break;

```

Moreover, the global event listeners for keydown events now check If a fighter is dead. If it is, the game no longer responds to movement and commands for that fighter, preventing any actions.

```

276 // Event listener handle the start of the movement
277 window.addEventListener("keydown", (event) => {
278   console.log(event.key);
279   if (!player.dead) {
280     // Check which key was pressed and mark it as pressed
281     switch (event.key) {
282       case "d":
283         keys.d.pressed = true;
284         player.lastKey = "d";
285         break;
286       case "a":
287         keys.a.pressed = true;
288         player.lastKey = "a";
289         break;
290       case "w":
291         player.velocity.y = -20;
292         break;
293       case " ":
294         console.log("attack");
295         player.attack();
296         break;
297     }
298   }
299 }
300
301
302 if (!enemy.dead) {
303   switch (event.key) {
304     case "ArrowRight":
305       keys.ArrowRight.pressed = true;
306       enemy.lastKey = "ArrowRight";
307       break;
308     case "ArrowLeft":
309       keys.ArrowLeft.pressed = true;
310       enemy.lastKey = "ArrowLeft";
311       break;
312     case "ArrowUp":
313       enemy.velocity.y = -20;
314       break;
315     case "ArrowDown":
316       console.log("attack");
317       enemy.attack();
318       break;
319   }
320 }
321
322
323
324
325
326

```

To finalize the game, the last touches included applying styles to the health bars and the timer, enhancing their appearance and the font was changed to "Press Start 2P".

```

9 </style>
10 <link rel="preconnect" href="https://fonts.googleapis.com" />
11 <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />
12 <link href="https://fonts.googleapis.com/css2?family=Press+Start+2P&display=swap" rel="stylesheet" />
13 </head>
14 <body>
15 <!-- Container div wraps the entire game interface including the health bars and the canvas -->
16 <div class="container" style="position: relative; display: inline-block">
17 <!-- smaller red container div -->
18 <div class="inside-container"
19 style="
20 position: absolute;
21 display: flex;
22 width: 100%;
23 align-items: center;
24 padding: 20px;
25 ">
26
27 <!-- player health -->
28 <div class="player-health" style="
29 position: relative;
30 width: 100%;
31 display: flex;
32 justify-content: flex-end;
33 border-top: 4px solid #fff;
34 border-left: 4px solid #fff;
35 border-bottom: 4px solid #fff;
36 ">
37
38 <div style="background-color: #ff0000; height: 30px; width: 100%;></div>
39 <div id="playerHealth" style="
40 position: absolute;
41 background: #881cf8;
42 top: 0;
43 right: 0;
44 bottom: 0;
45 width: 100%;
46 "></div>
47
48 <!-- timer -->
49 <div class="timer" id="timer" style="
50 background-color: #000000;
51 width: 100px;
52 height: 30px;
53 flex-shrink: 0;
54 display: flex;
55 align-items: center;
56 justify-content: center;
57 color: #fff;
58 border: 4px solid #fff;
59 ">
60
61 10
62 </div>
63 <!-- enemy health -->
64 <div class="enemy-health" style="
65 position: relative;
66 width: 100%;
67 border-top: 4px solid #fff;
68 border-bottom: 4px solid #fff;
69 border-right: 4px solid #fff;
70 ">
71
72 <div style="background-color: #ff0000; height: 30px;></div>
73 <div id="enemyHealth" style="
74 position: absolute;
75 background: #881cf8;
76 top: 0;
77 right: 0;
78 bottom: 0;
79 left: 0;
80 "></div>
81
82 <div></div>
83 <div id="displayText"
84 style="
85 position: absolute;
86 color: #fff;
87 align-items: center;
88 justify-content: center;
89 top: 0;
90 right: 0;
91 bottom: 0;
92 left: 0;
93 display: none;
94 ">
95
96 Tie
97 </div>

```

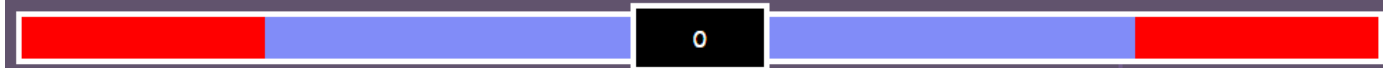
For a smoother transition of health deduction, the GSAP library was utilized, ensuring that the decrease in health bars is smoothly animated.

```

229 //detect for collision between player and enemy
230 if (
231   rectangularCollison({
232     rectangle1: player,
233     rectangle2: enemy,
234   }) &&
235   player.isAttacking &&
236   player.framesCurrent === 4
237 ) {
238   player.isAttacking = false;
239   enemy.takeHit();
240
241   // document.querySelector("#enemyHealth").style.width = enemy.health + "%";
242   gsap.to("#enemyHealth", {
243     width: enemy.health + "%",
244   });
245 }
246
247 //detect for collision between enemy and player
248 if (
249   rectangularCollison({
250     rectangle1: enemy,
251     rectangle2: player,
252   }) &&
253   enemy.isAttacking &&
254   enemy.framesCurrent === 2
255 ) {
256   player.takeHit();
257   enemy.isAttacking = false;
258   // document.querySelector("#playerHealth").style.width = player.health + "%";
259   gsap.to("#playerHealth", {
260     width: player.health + "%",
261   });
262   console.log("hit");
263 }
264
265 //if player misses
266 if (player.isAttacking && player.framesCurrent === 4) {
267   player.isAttacking = false;
268 }

```

These final changes wrapped up the development, ensuring the game functions well and also looks great, offering players a visually rich and engaging experience. The final product looks like this:



Tie

