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':

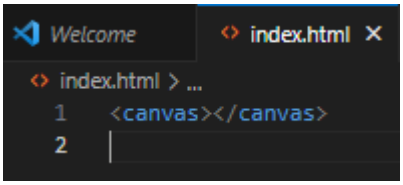
- Illustrate my practical understanding of HTML, CSS, and JavaScript through game development.
- Ensure the game is intuitive, with easy-to-understand controls and a clean user interface.
- Guarantee smooth gameplay and responsive controls
- Use graphics and a visual theme to create an engaging aesthetic.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

```
// Class representing any drawable object in the game
class Sprite {
  constructor({ position, velocity }) {
    this.position = position;
    this.velocity = velocity;
  }

  // Method to draw the sprite as a red rectangle on the canvas
  draw() {
    c.fillStyle = "red";
    c.fillRect(this.position.x, this.position.y, 50, 150);
  }

  // Update the sprite's position and redraw it
  update() {
    this.draw();
    this.position.y += 10;
  }
}

```

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

```
// Create the player sprite with initial position and velocity
const player = new Sprite({
  position: {
    x: 0,
    y: 0,
  },
  velocity: {
    x: 0,
    y: 0,
  },
});

// Create the enemy sprite with initial position and velocity
const enemy = new Sprite({
  position: {
    x: 400,
    y: 100,
  },
  velocity: {
    x: 0,
    y: 0,
  },
});

function animate() {
  // Setup the next animation frame
  window.requestAnimationFrame(animate);
  c.fillStyle = "black";
  c.fillRect(0, 0, canvas.width, canvas.height);
  console.log("goo");
  player.update();
  enemy.update();
}

animate();
```

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

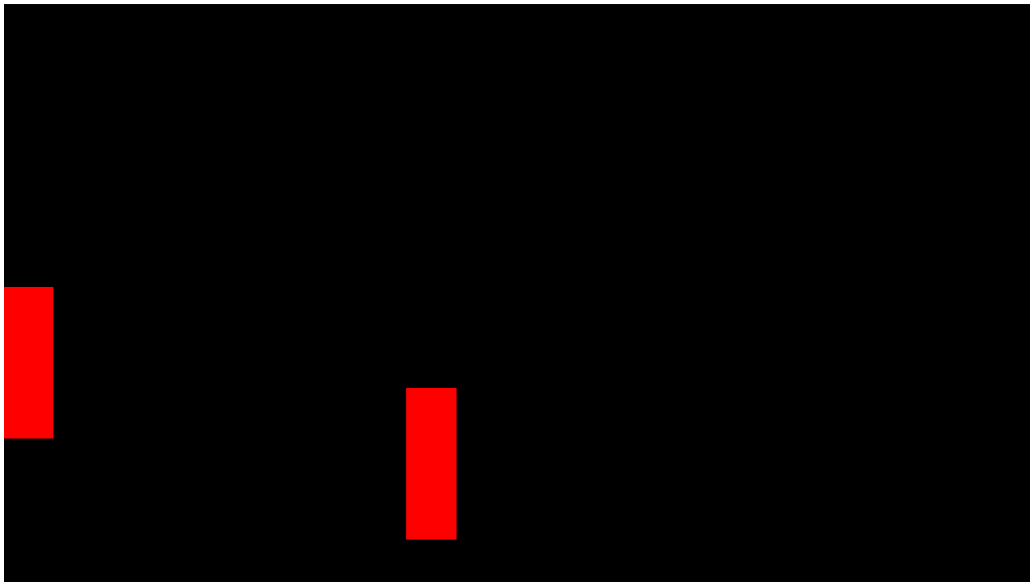
To prevent the painting effect as sprites moved, the canvas was cleared at the beginning of each animation frame using ‚c.clearRect(0, 0, canvas.width, canvas.height)'. This ensures that only the current frame's rendering is visible, not the previous ones.

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.

```
// Class representing any drawable object in the game
class Sprite {
  constructor({ position, velocity }) {
    this.position = position;
    this.velocity = velocity;
  }

  // Method to draw the sprite as a red rectangle on the canvas
  draw() {
    c.fillStyle = "red";
    c.fillRect(this.position.x, this.position.y, 50, 150);
  }

  // Update the sprite's position and redraw it
  update() {
    this.draw();
    this.position.y += 10;
  }
}
```

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:

```
c.fillRect(0, 0, canvas.width, canvas.height);
const graivty = 0.2;
// Class representing any drawable object in the game
class Sprite {
  constructor({ position, velocity }) {
    this.position = position;
    this.velocity = velocity;
    this.height = 150; // Height of the sprtie to calculate ground collision
  }

  // Method to draw the sprite as a red rectangle on the canvas
  draw() {
    c.fillStyle = "red";
    c.fillRect(this.position.x, this.position.y, 50, this.height);
  }

  // Update the sprite's position and redraw it
  update() {
    this.draw();

    // Update the sprite's position based on its velocity
    this.position.y += this.velocity.y;

    // If the sprite is about to move beyond the canvas floor
    if (this.position.y + this.height + this.velocity.y >= canvas.height) {
      this.velocity.y = 0;
    } else {
      this.velocity.y += 0.2;
    }
  }
}

// Create the player sprite with initial position and velocity
const player = new Sprite({
  position: {
    x: 0,
    y: 0,
  },
  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.

```
// Object to track the pressed state of the keys
const keys = {
  a: {
    pressed: false,
  },

  d: {
    pressed: false,
  },
};

// Variable to remember the last key pressed
let lastKey;

// Animation loop to update game frame
function animate() {
  // Setup the next animation frame
  window.requestAnimationFrame(animate);
  c.fillStyle = "black";
  c.fillRect(0, 0, canvas.width, canvas.height);
  console.log("goo");
  player.update();
  enemy.update();

  player.velocity.x = 0;

  //  adjust player's velocity based on the pressed keys
  if (keys.a.pressed && lastKey === "a") {
    player.velocity.x = -1;
  } else if (keys.d.pressed && lastKey === "d") {
    player.velocity.x = 1;
  }
}

animate();

// Event listener  handle the start of the movement
window.addEventListener("keydown", (event) => {
  console.log(event.key);
  // Check which key was pressed and mark it as pressed
  switch (event.key) {
    case "d":
      keys.d.pressed = true;
      lastKey = "d";
      break;
    case "a":
      keys.a.pressed = true;
      lastKey = "a";
      break;
  }
});

// Event listener  handle the stop of the movement
window.addEventListener("keyup", (event) => {
  // Reset the pressed state of keys when they are released
  switch (event.key) {
    case "d":
      keys.d.pressed = false;
      break;

    case "a":
      keys.a.pressed = false;
      break;
  }
});
```

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.

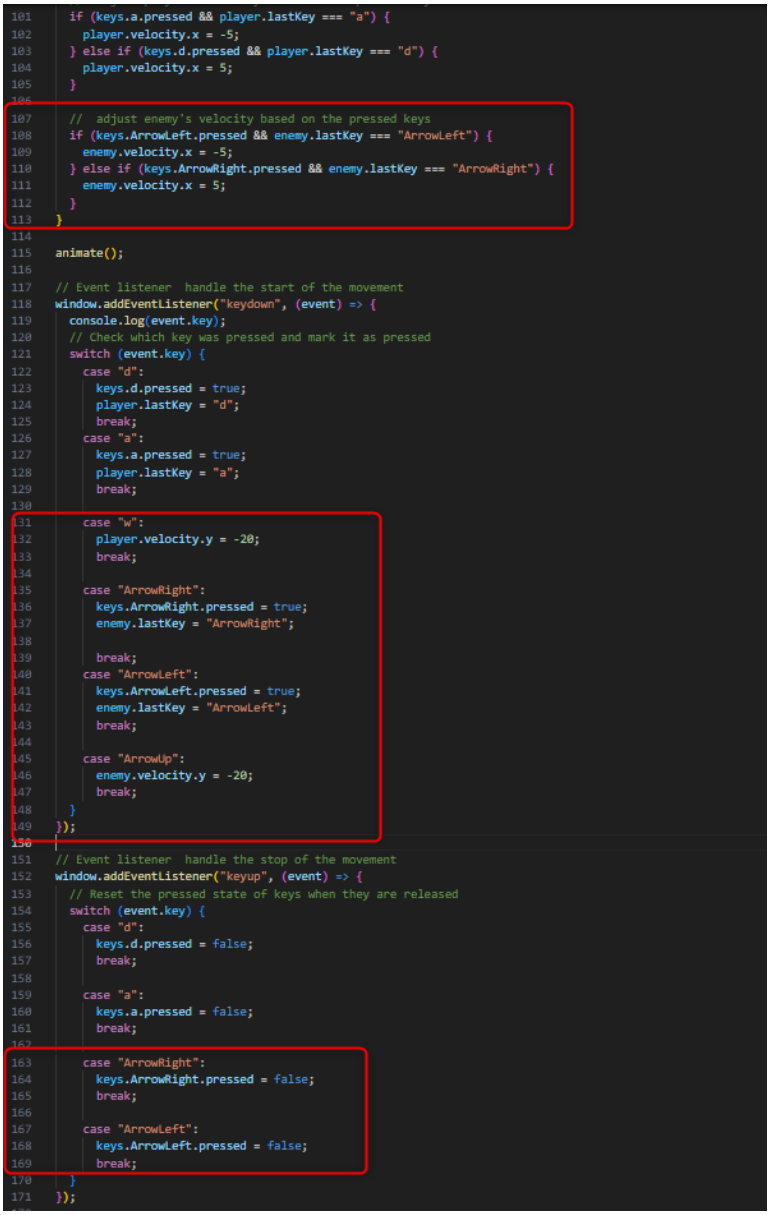
```
68  // Object to track the pressed state of the keys
69  const keys = {
70    a: {
71      pressed: false,
72    },
73
74    d: {
75      pressed: false,
76    },
77
78    ArrowRight: {
79      pressed: false,
80    },
81
82    ArrowLeft: {
83      pressed: false,
84    },
85  };
```

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 modifed code for the enemy control and jumping mechanics for both charcters:



```
  if (keys.a.pressed && player.lastKey === "a") {
    player.velocity.x = -5;
  } else if (keys.d.pressed && player.lastKey === "d") {
    player.velocity.x = 5;
  }

  //  adjust enemy's velocity based on the pressed keys
  if (keys.ArrowLeft.pressed && enemy.lastKey === "ArrowLeft") {
    enemy.velocity.x = -5;
  } else if (keys.ArrowRight.pressed && enemy.lastKey === "ArrowRight") {
    enemy.velocity.x = 5;
  }
}

animate();

// Event listener  handle the start of the movement
window.addEventListener("keydown", (event) => {
  console.log(event.key);
  // Check which key was pressed and mark it as pressed
  switch (event.key) {
    case "d":
      keys.d.pressed = true;
      player.lastKey = "d";
      break;
    case "a":
      keys.a.pressed = true;
      player.lastKey = "a";
      break;

    case "w":
      player.velocity.y = -20;
      break;

    case "ArrowRight":
      keys.ArrowRight.pressed = true;
      enemy.lastKey = "ArrowRight";

      break;
    case "ArrowLeft":
      keys.ArrowLeft.pressed = true;
      enemy.lastKey = "ArrowLeft";
      break;

    case "ArrowUp":
      enemy.velocity.y = -20;
      break;
  }
});

// Event listener  handle the stop of the movement
window.addEventListener("keyup", (event) => {
  // Reset the pressed state of keys when they are released
  switch (event.key) {
    case "d":
      keys.d.pressed = false;
      break;

    case "a":
      keys.a.pressed = false;
      break;

    case "ArrowRight":
      keys.ArrowRight.pressed = false;
      break;

    case "ArrowLeft":
      keys.ArrowLeft.pressed = false;
      break;
  }
});
```

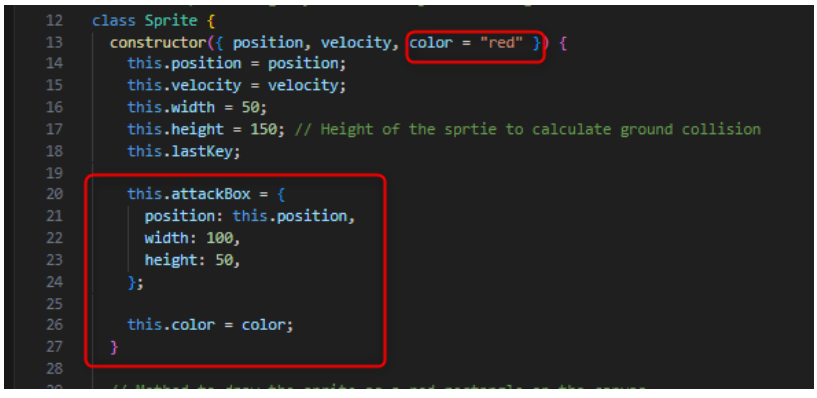
Also, unlike movement controls, the jump action does not require a keyup event to stop the action because the jump's upward motion is counteracted by gravity.

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 isnt overwritten.

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

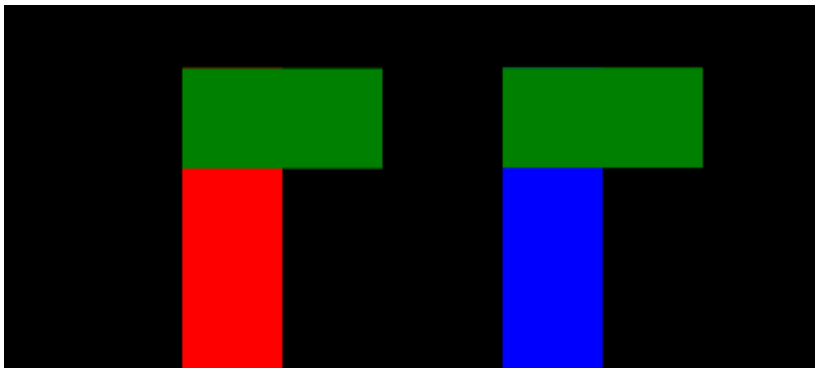
Adding Attack

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.



```
class Sprite {
  constructor({ position, velocity, color = "red" }) {
    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: this.position,
      width: 100,
      height: 50,
    };

    this.color = color;
  }
```

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.

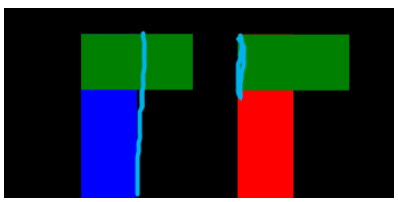
```
// Animation loop to update game frame
function animate() {
  // Setup the next animation frame
  window.requestAnimationFrame(animate);
  c.fillStyle = "black";
  c.fillRect(0, 0, canvas.width, canvas.height);
  //   console.log("goo");
  player.update();
  enemy.update();

  player.velocity.x = 0;
  enemy.velocity.x = 0;

  //  adjust player's velocity based on the pressed keys
  if (keys.a.pressed && player.lastKey === "a") {
    player.velocity.x = -5;
  } else if (keys.d.pressed && player.lastKey === "d") {
    player.velocity.x = 5;
  }

  //  adjust enemy's velocity based on the pressed keys
  if (keys.ArrowLeft.pressed && enemy.lastKey === "ArrowLeft") {
    enemy.velocity.x = -5;
  } else if (keys.ArrowRight.pressed && enemy.lastKey === "ArrowRight") {
    enemy.velocity.x = 5;
  }

  //detect for collision
  if (
    player.attackBox.position.x + player.attackBox.width >= enemy.position.x &&
    player.attackBox.position.x <= enemy.position.x + enemy.width &&
    player.attackBox.position.y + player.attackBox.height >= enemy.position.y &&
    player.attackBox.position.y <= enemy.position.y + enemy.height &&
    this.isAttacking
  ) {
    console.log("hit");
  }
}

animate();
```

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

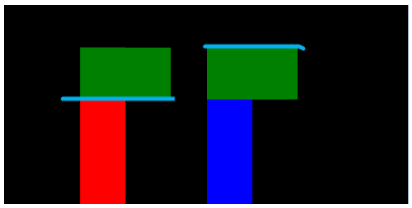
It evaluates several conditions to determine a hit:

- 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.

```
// Class representing any drawable object in the game
class Sprite {
  constructor({ position, velocity, color = "red", offset }) {
    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: {
        x: this.position.x,
        y: this.position.y,
      },
      offset,
      width: 100,
      height: 50,
    };

    this.color = color;
    this.isAttacking;
  }
```

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.

```
// Method to draw the sprite as a red rectangle on the canvas
draw() {
  c.fillStyle = this.color;
  c.fillRect(this.position.x, this.position.y, this.width, this.height);

  //attack box
  if (this.isAttacking) {
    c.fillStyle = "green";
    c.fillRect(
      this.attackBox.position.x,
      this.attackBox.position.y,
      this.attackBox.width,
      this.attackBox.height
    );
  }
}
```

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

```
// Checks for a succesfull hit
function rectangularCollison({ rectangle1, rectangle2 }) {
  return (
    rectangle1.attackBox.position.x + rectangle1.attackBox.width >=
      rectangle2.position.x &&
    rectangle1.attackBox.position.x <=
      rectangle2.position.x + rectangle2.width &&
    rectangle1.attackBox.position.y + rectangle1.attackBox.height >=
      rectangle2.position.y &&
    rectangle1.attackBox.position.y <= rectangle2.position.y + rectangle2.height
  );
}

```

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

```
// Update the sprite's position and redraw it
update() {
  this.draw();

  // Updating the attackbox position
  this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
  this.attackBox.position.y = this.position.y;

  this.position.x += this.velocity.x;

  // Update the sprite's position based on its velocity
  this.position.y += this.velocity.y;

  // If the sprite is about to move beyond the canvas floor
  if (this.position.y + this.height + this.velocity.y >= canvas.height) {
    this.velocity.y = 0;
  } else {
    this.velocity.y += graivty;
  }
}
attack() {
  this.isAttacking = true;
  console.log(this.isAttacking);
  setTimeout(() => {
    this.isAttacking = false;
    console.log(this.isAttacking);
  }, 100);
}
}
```

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.

```
// Event listener  handle the start of the movement
window.addEventListener("keydown", (event) => {
  console.log(event.key);
  // Check which key was pressed and mark it as pressed
  switch (event.key) {
    case "d":
      keys.d.pressed = true;
      player.lastKey = "d";
      break;
    case "a":
      keys.a.pressed = true;
      player.lastKey = "a";
      break;

    case "w":
      player.velocity.y = -20;
      break;

    case " ":
      console.log("attack");
      player.attack();

      break;

    case "ArrowRight":
      keys.ArrowRight.pressed = true;
      enemy.lastKey = "ArrowRight";

      break;
    case "ArrowLeft":
      keys.ArrowLeft.pressed = true;
      enemy.lastKey = "ArrowLeft";
      break;

    case "ArrowUp":
      enemy.velocity.y = -20;
      break;

    case "ArrowDown":
      console.log("attack");
      enemy.attack();

      break;
  }
});
```

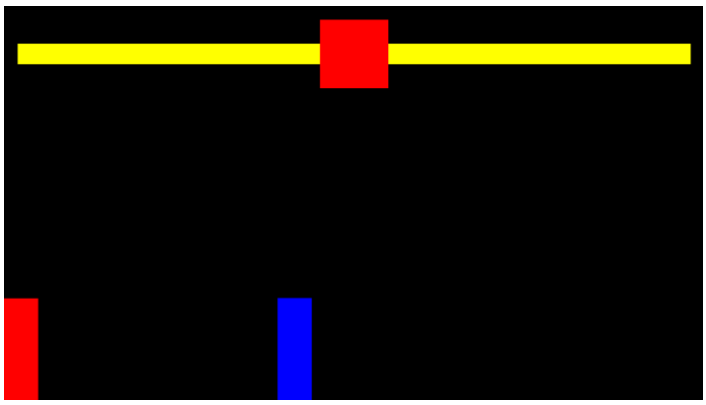
Attack Window: The attack box will appear for a brief moment, simulating an attack motion. This is achieved by setting ‚isAttacking' to true upon pressing the attack hotkey and then using setTimeout to revert it to false.

```
  // Update the sprite's position and redraw it
  update() {
    this.draw();

    // Updating the attackbox position
    this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
    this.attackBox.position.y = this.position.y;

    this.position.x += this.velocity.x;

    // Update the sprite's position based on its velocity
    this.position.y += this.velocity.y;

    // If the sprite is about to move beyond the canvas floor
    if (this.position.y + this.height + this.velocity.y >= canvas.height) {
      this.velocity.y = 0;
    } else {
      this.velocity.y += graivty;
    }
  }
  attack() {
    this.isAttacking = true;
    console.log(this.isAttacking);
    setTimeout(() => {
      this.isAttacking = false;
      console.log(this.isAttacking);
    }, 100);
  }
}
```

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:

```
index.html > body > script
<head>
  <style>
    /* Ensures that padding and borders are included in the total width and height of elements */
    * {
      box-sizing: border-box;
    }
  </style>
</head>
<body>
  <!-- Container div  wraps the entire game interface including the health bars and the canvas -->
  <div style="position: relative; display: inline-block">
    <!-- div that contains the health bars and the timer, positioned absolutely to overlay the canvas -->
    <div
      style="
        position: absolute;
        display: flex;
        width: 100%;
        align-items: center;
        padding: 20px; /* Adds space around the health bars and timer for visual spacing */
      "
    >
      <!-- Players health bar, taking up the full available width initially -->
      <div style="background-color: yellow; height: 30px; width: 100%"></div>
      <!-- Timer square in the center, with a fixed size and no flex shrink to maintain its size -->
      <div
        style="
          background-color: red;
          width: 100px;
          height: 100px;
          flex-shrink: 0; /* Prevents the timer from shrinking when other elements grow or shrink */
        "
      ></div>
      <!-- Enemy's health bar, also taking up the full width available initially -->
      <div
        id="enemyHealth"
        style="background-color: yellow; height: 30px; width: 100%"
      ></div>
    </div>
    <!-- The game canvas where the actual game is rendered -->
    <canvas></canvas>
  </div>
  <!-- Link to the game's JavaScript file -->
  <script src="index.js"></script>
</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:

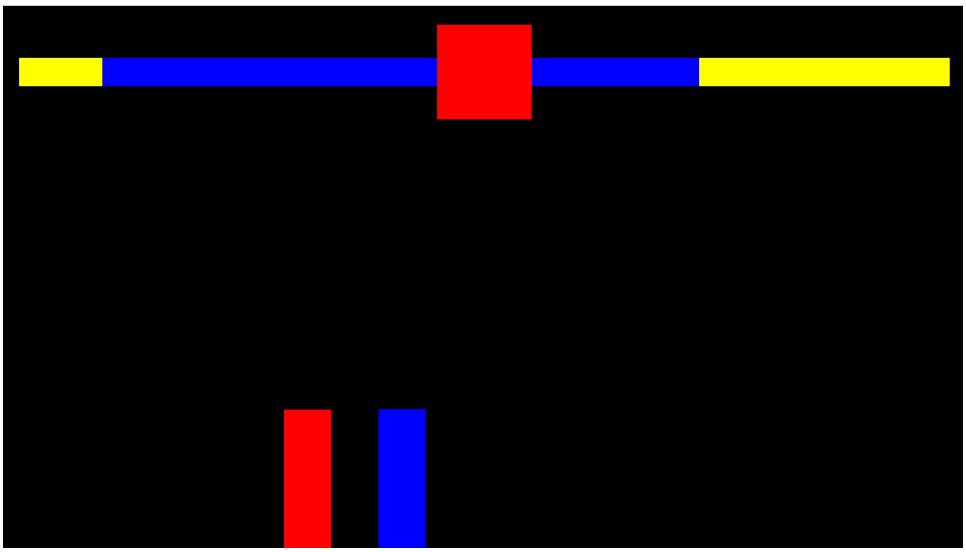
- Health Property: A health property is added to the Sprite class constructor to track the health of the sprites (player and enemy).

```
class Sprite {
  constructor({ position, velocity, color = "red", offset }) {
    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: {
        x: this.position.x,
        y: this.position.y,
      },
      offset,
      width: 100,
      height: 50,
    };

    this.color = color;
    this.isAttacking;
    this.health = 100;
  }

```

- 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.

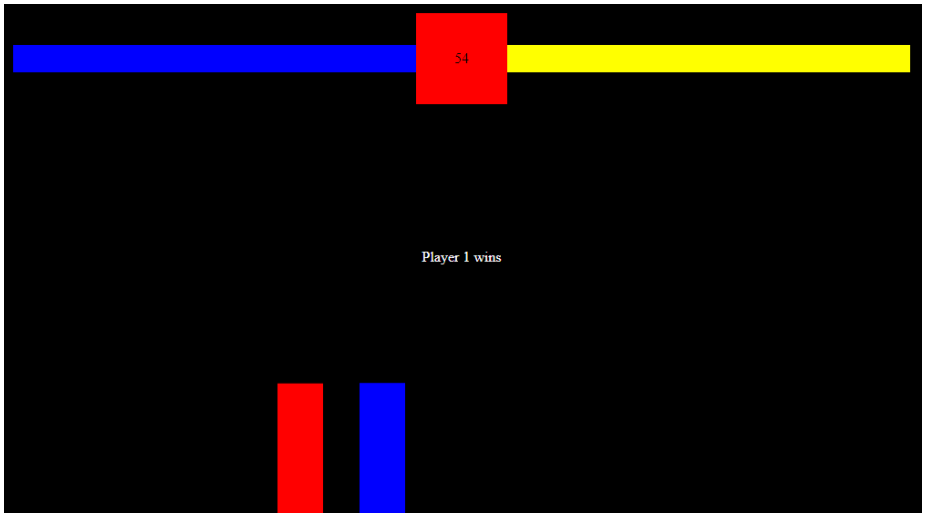
```
  //detect for collision between player and enemy
  if (
    rectangularCollison({
      rectangle1: player,
      rectangle2: enemy,
    }) &&
    player.isAttacking
  ) {
    player.isAttacking = false;
    enemy.health -= 20;
    document.querySelector("#enemyHealth").style.width = enemy.health + "%";
  }

  //detect for collision between enemy and player
  if (
    rectangularCollison({
      rectangle1: enemy,
      rectangle2: player,
    }) &&
    enemy.isAttacking
  ) {
    player.health -= 20;
    enemy.isAttacking = false;
    document.querySelector("#playerHealth").style.width = player.health + "%";
    console.log("hit");
  }
}

```

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.

```
<!-- Timer square in the center, with a fixed size and no flex shrink to maintain its size -->
<div
  id="timer"
  style="
    background-color: red;
    width: 100px;
    height: 100px;
    flex-shrink: 0; /* Prevents the timer from shrinking when other elements grow or shrink */
    display: flex;
    align-items: center;
    justify-content: center;
  "
>
  10
</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*:*

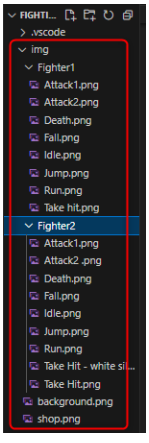
- The ‚decreaseTimer()' function is invoked once when the game starts to begin the countdown.
- The game's existing collision detection logic, which decrements health upon successful attacks, now also checks the timer. If the timer reaches zero, the game transitions to the end state, showing the result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

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 arent 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 += graivty;
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
```

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:

```
// Class representing the images
class Sprite {
  constructor({ position, imageSrc }) {
    this.position = position;
    this.width = 50;
    this.height = 150;
    this.image = new Image();
    this.image.src = imageSrc;
  }

  draw() {
    c.drawImage(this.image, this.position.x, this.position.y);
  }

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

// Class representing the fighters in the game
class Fighter {
  constructor({ position, velocity, color = "red", offset }) {
    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: {
        x: this.position.x,
        y: this.position.y,
      },
      offset,
      width: 100,
      height: 50,
    };

    this.color = color;
    this.isAttacking;
    this.health = 100;
  }

  // Method to draw the sprite as a red rectangle on the canvas
  draw() {
    c.fillStyle = this.color;
    c.fillRect(this.position.x, this.position.y, this.width, this.height);

    //attack box
    if (this.isAttacking) {
      c.fillStyle = "green";
      c.fillRect(
        this.attackBox.position.x,
        this.attackBox.position.y,
        this.attackBox.width,
        this.attackBox.height
      );
    }
  }

  // Update the sprite's position and redraw it
  update() {
    this.draw();

    // Updating the attackbox position
    this.attackBox.position.x = this.position.x + this.attackBox.offset.x;
    this.attackBox.position.y = this.position.y;

    this.position.x += this.velocity.x;

    // Update the sprite's position based on its velocity
    this.position.y += this.velocity.y;

    // Update the sprite's position based on its velocity
    this.position.y += this.velocity.y;

    // If the sprite is about to move beyond the canvas floor
    if (this.position.y + this.height + this.velocity.y >= canvas.height - 96) {
      this.velocity.y = 0;
    } else {
      this.velocity.y += graivty;
    }
  }
  attack() {
    this.isAttacking = true;
    console.log(this.isAttacking);
    setTimeout(() => {
      this.isAttacking = false;
      console.log(this.isAttacking);
    }, 100);
  }
}
```

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:

```
// Class representing the images
class Sprite {
  constructor({ position, imageSrc, scale = 1, framesMax = 1 }) {
    this.position = position;
    this.width = 50;
    this.height = 150;
    this.image = new Image();
    this.image.src = imageSrc;
    this.scale = scale;
    this.framesMax = framesMax;
    this.framesCurrent = 0;
    this.framesElapsed = 0;
    this.framesHold = 10; // How many frames to hold before transitioning to the next frame
  }

  draw() {
    c.drawImage(
      this.image,
      this.framesCurrent * (this.image.width / this.framesMax), // The x coordinate where to start clipping
      0, // The y coordinate where to start clipping
      this.image.width / this.framesMax, // The width of the clipped image
      this.image.height, // The height of the clipped image
      this.position.x, // The x position where to place the image on the canvas
      this.position.y, // The y position where to place the image on the canvas
      (this.image.width / this.framesMax) * this.scale, // The width of the image to be drawn on the canvas, scaled
      this.image.height * this.scale // The height of the image to be drawn on the canvas, scaled
    );
  }

  update() {
    this.draw();

    this.framesElapsed++;
    if (this.framesElapsed % this.framesHold === 0) {
      if (this.framesCurrent < this.framesMax - 1) {
        this.framesCurrent = this.framesCurrent + 1; // Move to the next frame
      } else this.framesCurrent = 0; // Reset to the first frame
    }
  }
}
```

‚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.

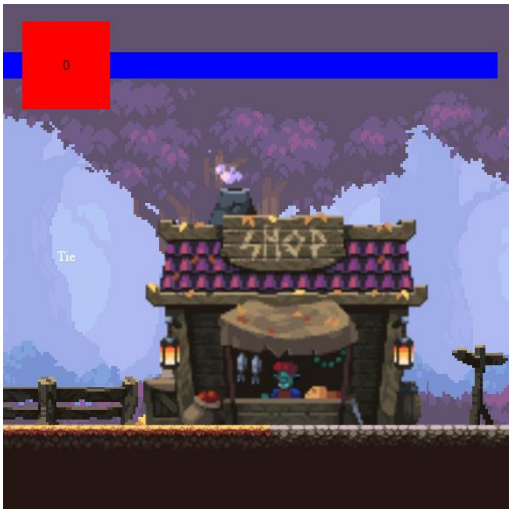
Update method

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 occured:

- 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.

```
// Class representing the images
class Sprite {
  constructor({
    position,
    imageSrc,
    scale = 1,
    framesMax = 1,
    offset = { x: 0, y: 0 },
  }) {
    this.position = position;
    this.width = 50;
    this.height = 150;
    this.image = new Image();
    this.image.src = imageSrc;
    this.scale = scale;
    this.framesMax = framesMax;
    this.framesCurrent = 0;
    this.framesElapsed = 0;
    this.framesHold = 5; // How many frames to hold before transitioning to the next frame
    this.offset = offset;
  }

  draw() {
    c.drawImage(
      this.image,
      this.framesCurrent * (this.image.width / this.framesMax), // The x coordinate where to start clipping
      0, // The y coordinate where to start clipping
      this.image.width / this.framesMax, // The width of the clipped image
      this.image.height, // The height of the clipped image
      this.position.x - this.offset.x, // The x position where to place the image on the canvas
      this.position.y - this.offset.y, // The y position where to place the image on the canvas
      (this.image.width / this.framesMax) * this.scale, // The width of the image to be drawn on the canvas, scaled
      this.image.height * this.scale // The height of the image to be drawn on the canvas, scaled
    );
  }
  animateFrames() {
    this.framesElapsed++;
    if (this.framesElapsed % this.framesHold === 0) {
      if (this.framesCurrent < this.framesMax - 1) {
        this.framesCurrent = this.framesCurrent + 1;
        // console.log("Frames current ", this.framesCurrent," Frames max ", this.framesMax)
      } else this.framesCurrent = 0;
    }
  }

  update() {
    this.draw();

    this.animateFrames();
  }
}
```

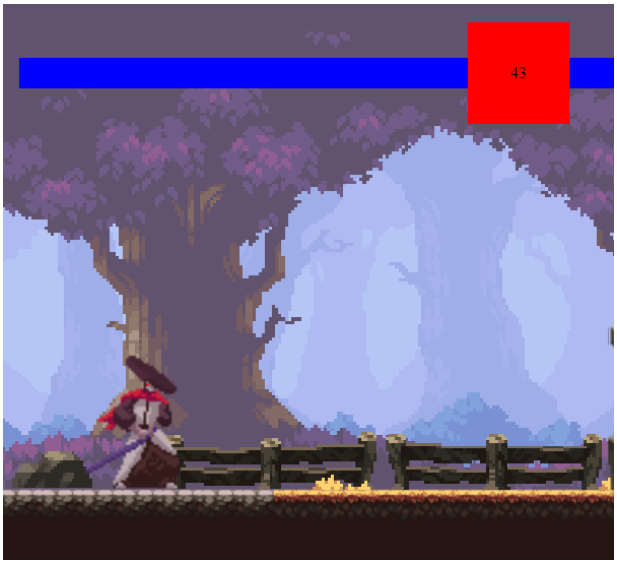
```
// Class representing the fighters in the game
class Fighter extends Sprite {
  constructor({
    position,
    velocity,
    color = "red",
    imageSrc,
    scale = 1,
    framesMax = 1,
    offset = { x: 0, y: 0 },
  }) {
    super({
      position,
      imageSrc,
      scale,
      framesMax,
      offset,
    });

    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: {
        x: this.position.x,
        y: this.position.y,
      },
      offset,
      width: 100,
      height: 50,
    };

    this.color = color;
    this.isAttacking;
    this.health = 100;
    this.framesCurrent = 0;
    this.framesElapsed = 0;
    this.framesHold = 10; // How many frames to hold before transitioning to the next frame
  }

  // Method to draw the sprite as a red rectangle on the canvas

  // Update the sprite's position and redraw it
  update() {
    this.draw();
    this.animateFrames();
```

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.

```
// Create the player sprite with initial position and velocity
const player = new Fighter({
  position: {
    x: 0,
    y: 0,
  },
  velocity: {
    x: 0,
    y: 10,
  },

  imageSrc: "./img/Fighter2/idle.png",
  framesMax: 8,
  scale: 2.5,
  offset: {
    x: 215,
    y: 157,
  },
});
```

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.

```
class Fighter extends Sprite {
  constructor({
    position,
    velocity,
    color = "red",
    imageSrc,
    scale = 1,
    framesMax = 1,
    offset = { x: 0, y: 0 },
    sprites,
  }) {
    super({
      position,
      imageSrc,
      scale,
      framesMax,
      offset,
    });

    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: {
        x: this.position.x,
        y: this.position.y,
      },
      offset,
      width: 100,
      height: 50,
    };

    this.color = color;
    this.isAttacking;
    this.health = 100;
    this.framesCurrent = 0;
    this.framesElapsed = 0;
    this.framesHold = 10; // How many frames to hold before transitioning to the next frame
    this.sprites = sprites; // Initialize the sprites property with all the sprites needed for the animation

```

Initializing Sprites Object:

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

```
// Create the player sprite with initial position and velocity
const player = new Fighter({
  position: {
    x: 0,
    y: 0,
  },
  velocity: {
    x: 0,
    y: 10,
  },

  imageSrc: "./img/Fighter2/idle.png",
  framesMax: 8,
  scale: 2.5,
  offset: {
    x: 215,
    y: 157,
  },
  sprites: {
    idle: {
      imageSrc: "img/Fighter2/Idle.png",
      framesMax: 8,
    },
    run: {
      imageSrc: "img/Fighter2/Run.png",
      framesMax: 8,
    },
  },
});
```

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 ensure 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 loade.

```
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.

```
switchSprite(sprite) {
  // Checks for the sprite that matches the case
  switch (sprite) {
    case "idle":
      if (this.image !== this.sprites.idle.image) {
        this.image = this.sprites.idle.image;
        this.framesMax = this.sprites.idle.framesMax;
        this.framesCurrent = 0; // Reset the current frame to 0 to start the animation from the beginning
      }
      break;

    case "run":
      if (this.image !== this.sprites.run.image) {
        this.image = this.sprites.run.image;
        this.framesMax = this.sprites.run.framesMax;
        this.framesCurrent = 0;
      }
      break;

    case "jump":
      if (this.image !== this.sprites.jump.image) {
        this.image = this.sprites.jump.image;
        this.framesMax = this.sprites.jump.framesMax;
        this.framesCurrent = 0;
      }
      break;
    case "fall":
      if (this.image !== this.sprites.fall.image) {
        this.image = this.sprites.fall.image;
        this.framesMax = this.sprites.fall.framesMax;
        this.framesCurrent = 0;
      }
      break;
  }
 }
}
```

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.

```
//  adjust player's velocity based on the pressed keys
if (keys.a.pressed && player.lastKey === "a") {
  player.velocity.x = -5;
  player.switchSprite("run");
} else if (keys.d.pressed && player.lastKey === "d") {
  player.velocity.x = 5;
  player.switchSprite("run");
} else {
  player.switchSprite("idle");
}

//jumping and falling
if (player.velocity.y < 0) {
  player.switchSprite("jump");
} else if (player.velocity.y > 0) {
  player.switchSprite("fall");
}
```

Adding the Attack Animation

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

```
  sprites: {
    idle: {
      imageSrc: "img/Fighter2/Idle.png",
      framesMax: 8,
    },
    run: {
      imageSrc: "img/Fighter2/Run.png",
      framesMax: 8,
    },
    jump: {
      imageSrc: "img/Fighter2/Jump.png",
      framesMax: 2,
    },
    fall: {
      imageSrc: "img/Fighter2/Fall.png",
      framesMax: 2,
    },
    attack1: {
      imageSrc: "img/Fighter2/Attack1.png",
      framesMax: 6,
    },
  },
});
```

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.

```
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 includes 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) {
```

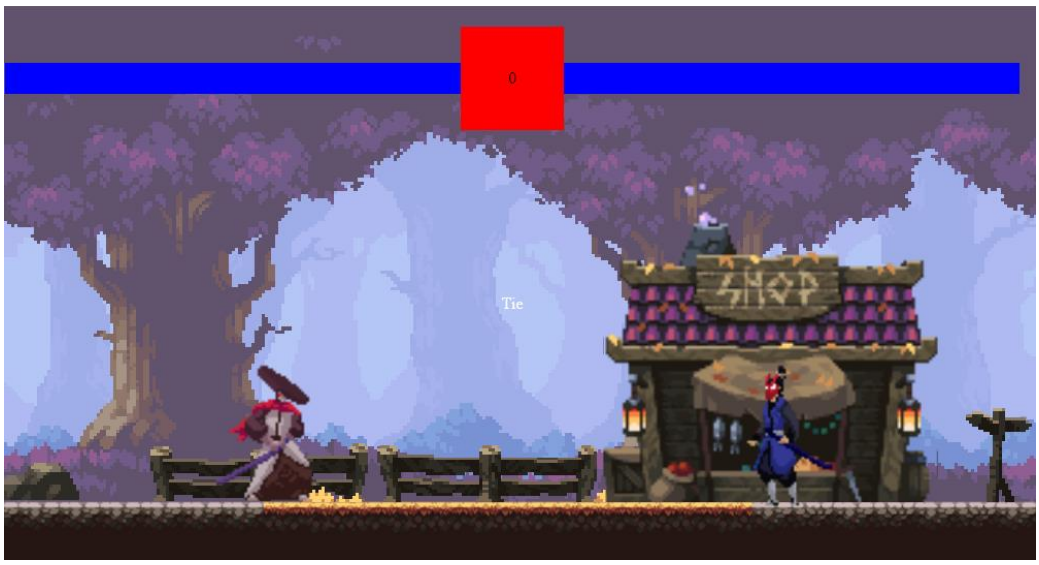
This is how the attack animation will look like:

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

```
// Create the enemy sprite with initial position and velocity
const enemy = new Fighter({
  position: {
    x: 400,
    y: 100,
  },
  velocity: {
    x: 0,
    y: 0,
  },
  offset: {
    x: -50,
    y: 0,
  },
  imageSrc: "./img/Fighter1/idle.png",
  framesMax: 4,
  scale: 2.5,
  offset: {
    x: 215,
    y: 167,
  },
  sprites: {
    idle: {
      imageSrc: "img/Fighter1/Idle.png",
      framesMax: 4,
    },
    run: {
      imageSrc: "img/Fighter1/Run.png",
      framesMax: 8,
    },
    jump: {
      imageSrc: "img/Fighter1/Jump.png",
      framesMax: 2,
    },
    fall: {
      imageSrc: "img/Fighter1/Fall.png",
      framesMax: 2,
    },
    attack1: {
      imageSrc: "img/Fighter1/Attack1.png",
      framesMax: 4,
    },
  },
});
```

```
//  adjust player's velocity based on the pressed keys
if (keys.a.pressed && player.lastKey === "a") {
  player.velocity.x = -5;
  player.switchSprite("run");
} else if (keys.d.pressed && player.lastKey === "d") {
  player.velocity.x = 5;
  player.switchSprite("run");
} else {
  player.switchSprite("idle");
}

//jumping and falling
if (player.velocity.y < 0) {
  player.switchSprite("jump");
} else if (player.velocity.y > 0) {
  player.switchSprite("fall");
}

//  adjust enemy's velocity based on the pressed keys
if (keys.ArrowLeft.pressed && enemy.lastKey === "ArrowLeft") {
  enemy.velocity.x = -5;
  enemy.switchSprite("run");
} else if (keys.ArrowRight.pressed && enemy.lastKey === "ArrowRight") {
  enemy.velocity.x = 5;
  enemy.switchSprite("run");
} else {
  enemy.switchSprite("idle");
}

//jumping and falling - enemy
if (enemy.velocity.y < 0) {
  enemy.switchSprite("jump");
} else if (enemy.velocity.y > 0) {
  enemy.switchSprite("fall");
}
```

The canvas will now showcase both of the fighters:

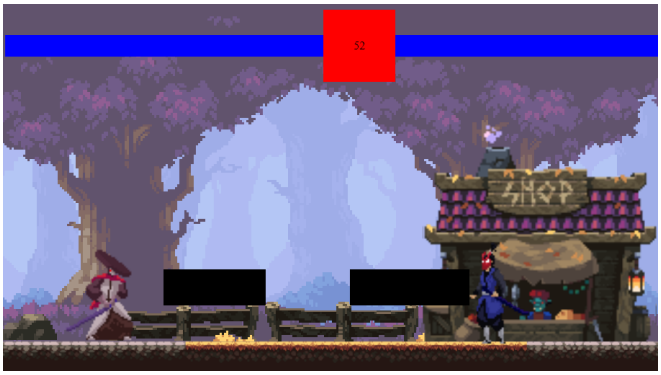


To continue, the attack box was correctly placed and synchronized with the animation frames. For this, several steps were taken to improve the game mechanics:

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.

```
//draw attack box to position it
c.fillRect(
  this.attackBox.position.x,
  this.attackBox.position.y,
  this.attackBox.width,
  this.attackBox.height
);
```



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.

```
// Class representing the fighters in the game
class Fighter extends Sprite {
  constructor({
    position,
    velocity,
    color = "red",
    imageSrc,
    scale = 1,
    framesMax = 1,
    offset = { x: 0, y: 0 },
    sprites,
    attackBox = { offset: {}, width: undefined, height: undefined },
  }) {
    super({
      position,
      imageSrc,
      scale,
      framesMax,
      offset,
    });

    this.position = position;
    this.velocity = velocity;
    this.width = 50;
    this.height = 150; // Height of the sprtie to calculate ground collision
    this.lastKey;

    this.attackBox = {
      position: {
        x: this.position.x,
        y: this.position.y,
      },
      offset: attackBox.offset,
      width: attackBox.width,
      height: attackBox.height,
    };

    this.color = color;
    this.isAttacking;
    this.health = 100;
    this.framesCurrent = 0;
    this.framesElapsed = 0;
    this.framesHold = 5; // How many frames to hold before transitioning to the next frame
    this.sprites = sprites; // Initialize the sprites property with all the sprites needed for the animation

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

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

```
const enemy = new Fighter({
  position: {
    x: 400,
    y: 100,
  },
  velocity: {
    x: 0,
    y: 0,
  },
  offset: {
    x: -50,
    y: 0,
  },
  imageSrc: "./img/Fighter1/idle.png",
  framesMax: 4,
  scale: 2.5,
  offset: {
    x: 215,
    y: 167,
  },
  sprites: {
    idle: {
      imageSrc: "img/Fighter1/Idle.png",
      framesMax: 4,
    },
    run: {
      imageSrc: "img/Fighter1/Run.png",
      framesMax: 8,
    },
    jump: {
      imageSrc: "img/Fighter1/Jump.png",
      framesMax: 2,
    },
    fall: {
      imageSrc: "img/Fighter1/Fall.png",
      framesMax: 2,
    },
    attack1: {
      imageSrc: "img/Fighter1/Attack1.png",
      framesMax: 4,
    },
  },
  attackBox: {
    offset: {
      x: -166,
      y: 50,
    },
    width: 166,
    height: 50,
  },
});
```

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.

```
//detect for collision between player and enemy
if (
  rectangularCollison({
    rectangle1: player,
    rectangle2: enemy,
  }) &&
  player.isAttacking &&
  player.framesCurrent === 4
) {
  player.isAttacking = false;
  enemy.health -= 20;
  document.querySelector("#enemyHealth").style.width = enemy.health + "%";
}
```

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.

```
//if player misses
if (player.isAttacking && player.framesCurrent === 4) {
  player.isAttacking = false;
}

//if enemy misses
if (enemy.isAttacking && enemy.framesCurrent === 2) {
  enemy.isAttacking = false;
}
```

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:

```
const enemy = new Fighter({
  position: {
    x: 400,
    y: 100,
  },
  velocity: {
    x: 0,
    y: 0,
  },
  offset: {
    x: -50,
    y: 0,
  },
  imageSrc: "./img/Fighter1/Idle.png",
  framesMax: 4,
  scale: 2.5,
  offset: {
    x: 215,
    y: 167,
  },
  sprites: {
    idle: {
      imageSrc: "img/Fighter1/Idle.png",
      framesMax: 4,
    },
    run: {
      imageSrc: "img/Fighter1/Run.png",
      framesMax: 8,
    },
    jump: {
      imageSrc: "img/Fighter1/Jump.png",
      framesMax: 2,
    },
    fall: {
      imageSrc: "img/Fighter1/Fall.png",
      framesMax: 2,
    },
    attack1: {
      imageSrc: "img/Fighter1/Attack1.png",
      framesMax: 4,
    },
    takeHit: {
      imageSrc: "img/Fighter1/Take hit.png",
      framesMax: 3,
    },
    death: {
      imageSrc: "img/Fighter1/Death.png",
      framesMax: 7,
    },
  },
  attackBox: {
    offset: {
      x: -166,
      y: 50,
    },
    width: 166,
    height: 50,
```

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.

```
takeHit() {
  this.health -= 20;

  if (this.health <= 0) {
    this.switchSprite("death");
  } else this.switchSprite("takeHit");
}
```

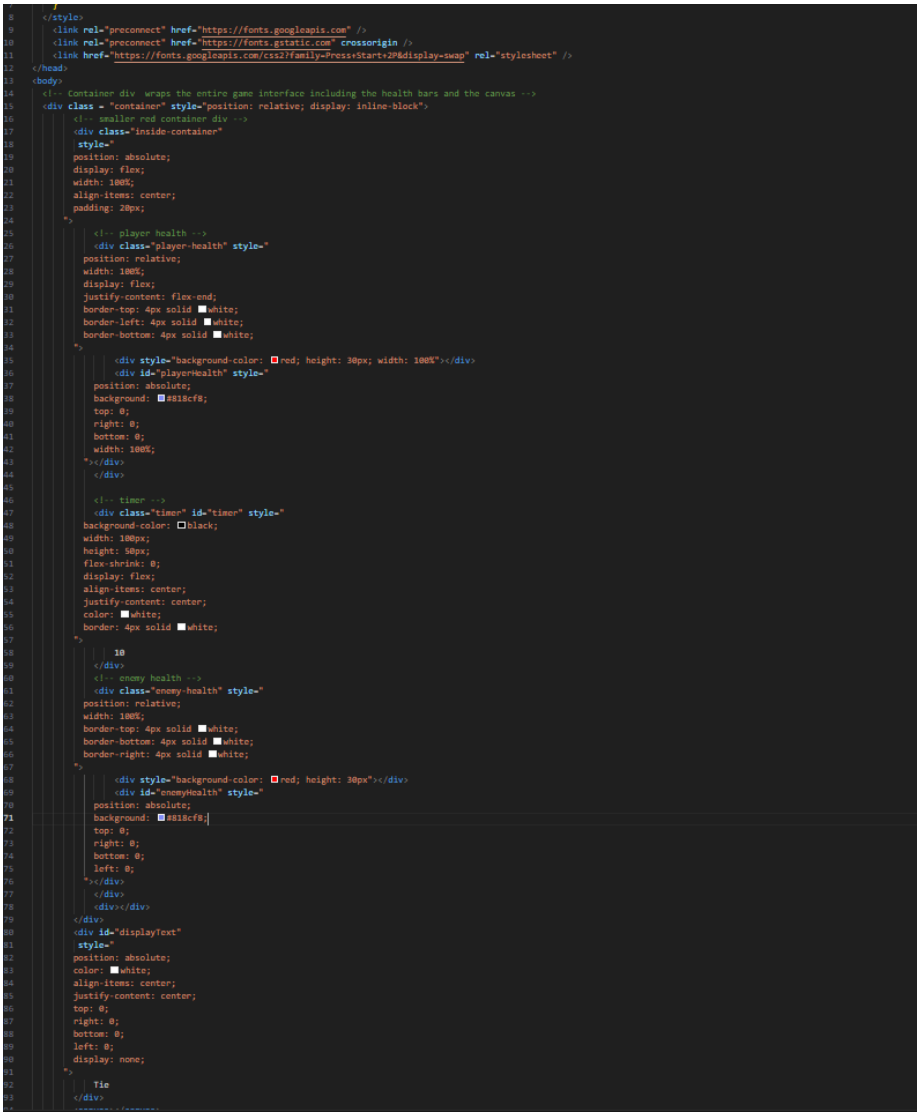
The ‚switchSprite()' method in the Fighter class has been updated to include cases for "takeHit" and "death" animations. Additional checks prevent the animation from looping indefinitely and ensure that the "death" animation only plays once by setting a new dead property to true. This property was added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

```
switchSprite(sprite) {
  //if the player dies you cant continue
  if (this.image === this.sprites.death.image) {
    if (this.framesCurrent === this.sprites.death.framesMax - 1)
      this.dead = true;
    return;
  }
  //prevents the animation from looping the attack
  if (
    this.image === this.sprites.attack1.image &&
    this.framesCurrent < this.sprites.attack1.framesMax - 1
  )
    return;

  //override when fighter gets hit
  if (
    this.image === this.sprites.takeHit.image &&
    this.framesCurrent < this.sprites.takeHit.framesMax - 1
  )
    return;
  // Checks for the sprite that matches the case
  switch (sprite) {
    case "takeHit":
      if (this.image !== this.sprites.takeHit.image) {
        this.image = this.sprites.takeHit.image;
        this.framesMax = this.sprites.takeHit.framesMax;
        this.framesCurrent = 0;
      }
      break;
    case "death":
      if (this.image !== this.sprites.death.image) {
        this.image = this.sprites.death.image;
        this.framesMax = this.sprites.death.framesMax;
        this.framesCurrent = 0;
      }
      break;
```

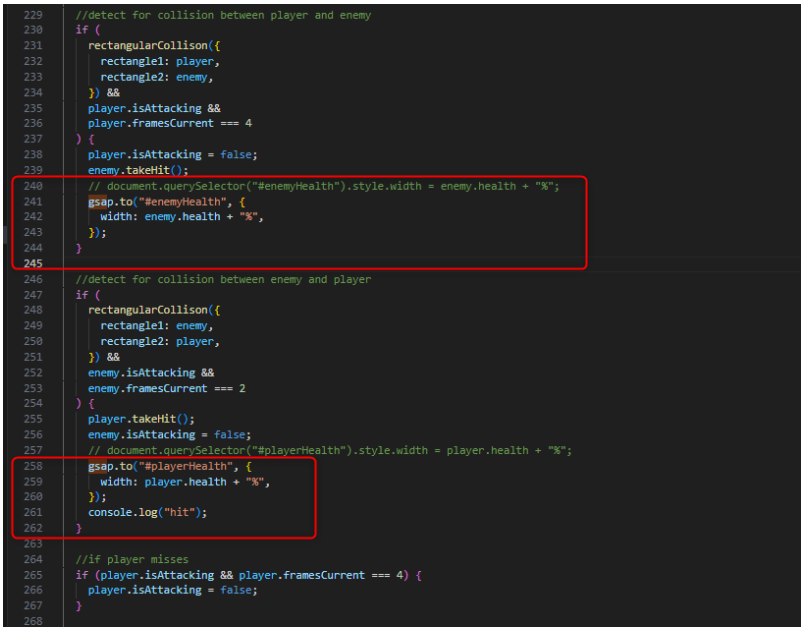
Moreover, the global event listeners for keydown events now check If a fighter is dead. If it is, the game no longer responds to movement and commands for that fighter, preventing any actions.

```
// Event listener  handle the start of the movement
window.addEventListener("keydown", (event) => {
  console.log(event.key);
  if (!player.dead) {
    // Check which key was pressed and mark it as pressed
    switch (event.key) {
      case "d":
        keys.d.pressed = true;
        player.lastKey = "d";
        break;
      case "a":
        keys.a.pressed = true;
        player.lastKey = "a";
        break;

      case "w":
        player.velocity.y = -20;
        break;

      case " ":
        console.log("attack");
        player.attack();

        break;
    }
  }

  if (!enemy.dead) {
    switch (event.key) {
      case "ArrowRight":
        keys.ArrowRight.pressed = true;
        enemy.lastKey = "ArrowRight";

        break;
      case "ArrowLeft":
        keys.ArrowLeft.pressed = true;
        enemy.lastKey = "ArrowLeft";
        break;

      case "ArrowUp":
        enemy.velocity.y = -20;
        break;

      case "ArrowDown":
        console.log("attack");
        enemy.attack();

        break;
    }
  }
});
```

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,".



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



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:


0
Tie
SHOP