Side-Scrolling Game Workshop

This workshop for in-class lab for the "JavaScript Advanced" course @ SoftUni.

This document defines a complete walkthrough of creating a **Side-scrolling Game** with JavaScript and HTML.

You will need a text editor or IDE and a browser. In this lesson we will use the latest version of Visual Studio Code. You can download the **resources** which contain the images we will be using for our game.

If you are interested in what exactly a Side-scrolling video game is, you can read this article.

I. Project Overview

We will play with the wizard who flies in the sky. Bugs will appear from the right and head towards us. We score points by shooting them with fireballs. If we hit them, they die. If a bug hits us the game is over. In the end of this workshop our game will look like the screenshot below.

II. Base Skeleton

First, we create **index.html** for in game screens and then - **styles.css** for game styles. Our JavaScript logic will be kept in **main.js**. Don't forget to include our newly created files in **index.html**. We include **styles.css** in the html head tag and **main.js** just above the closing body tag. At this point we have our files created and linked in **index.html**. Ok, but where is our game? Inside the body tag we need a couple of div elements which will contain our game screens.

- Our first step is to create a **div** with **class game-section.** This is our game screen, containing all other screens. Inside **game-section** create 4 **divs**:

1. One with **class game-start**. This screen will contain our 'Press to Play!' button.
2. Another with **class game-score**. This will be our score counter.
3. Yet another with **class game-area.** This will be our main game screen.
4. And lastly one with **class game-over.** We will keep this screen hidden until the game is over.

We are done with our base skeleton for the game! In the next section of this lesson we begin styling our div elements and adding our game logic in the **main.js** file.

At this point **index.html** should look something like this:

index.html × | main.js | styles.css

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="styles.css">
    <title>Wizard Game</title>
</head>

<body>
    <!-- game screen -->
    <div class="game-section">

        <!-- Game start screen -->
        <div class="game-start">Press to Play!</div>

        <!-- score -->
        <div class="game-score"></div>

        <!-- main game area -->
        <div class="game-area"></div>

        <!-- game over -->
        <div class="game-over"></div>

    </div>
    <!-- game screen ends -->

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

</html>
```

III. Creating the Game

Adding Base Styles

After we created our base skeleton in section II, we will add some base styles to our document.

We will start by adding some for our whole **body**. Open **styles.css**, make all **font sizes 18px** and use the **`Consolas`** font family**.**

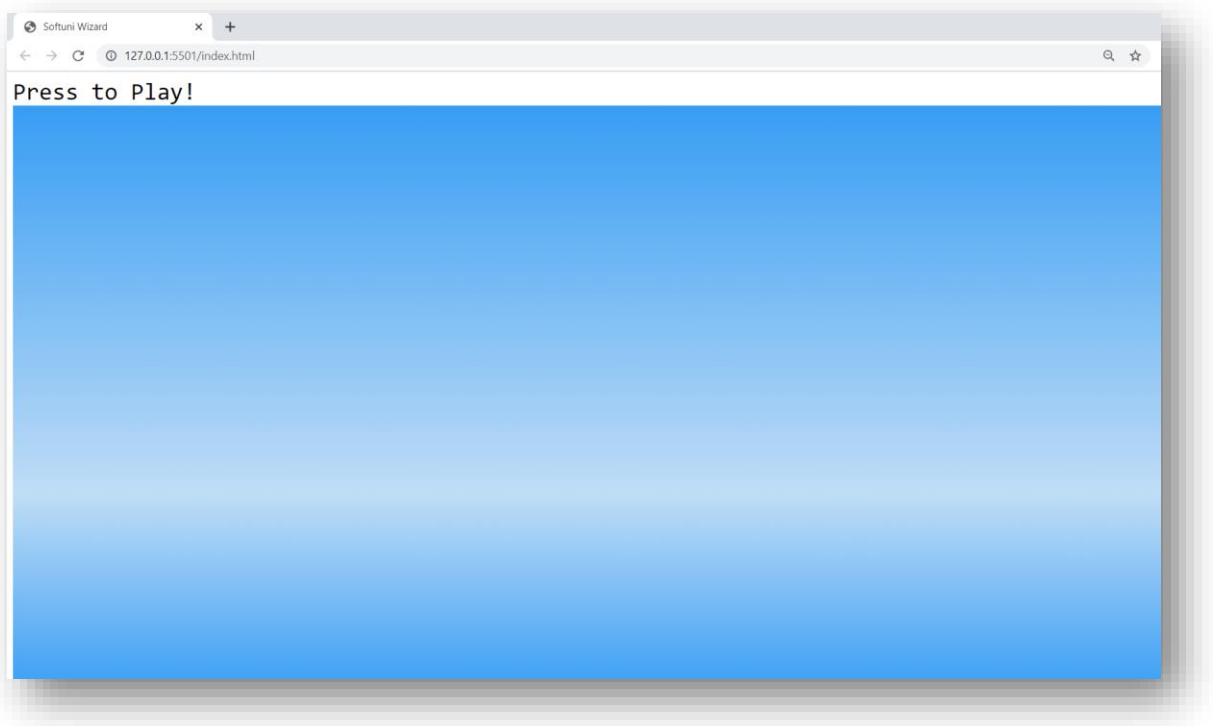
```
body {
    font-size: 32px;
    font-family: consolas;
}
```

Now it is time to add some styles to our div elements.

We start with our **game-area**. This screen needs to be blue since it represents the sky, for more realistic color we can use linear gradient of different shades of blue as our **background color.** You can use given colors from above or go your own style. To make positioning of our other game screens absolute compared to this one, we use **relative position**. Our **game-area** needs to be full-screen, so we set **width to 100%** and **height** to **93vh.** The div's **default margin** will not be used, so we set it to **0.**

```
.game-area {
    position: relative;
    background: rgb(61, 160, 244);
    background: linear-gradient(0deg, rgba(61, 160, 244, 1) 0%,
                rgba(192, 221, 246, 1) 33%,
                rgba(55, 157, 244, 1) 100%);
    width: 100%;
    height: 93vh;
    margin: 0;
}
```

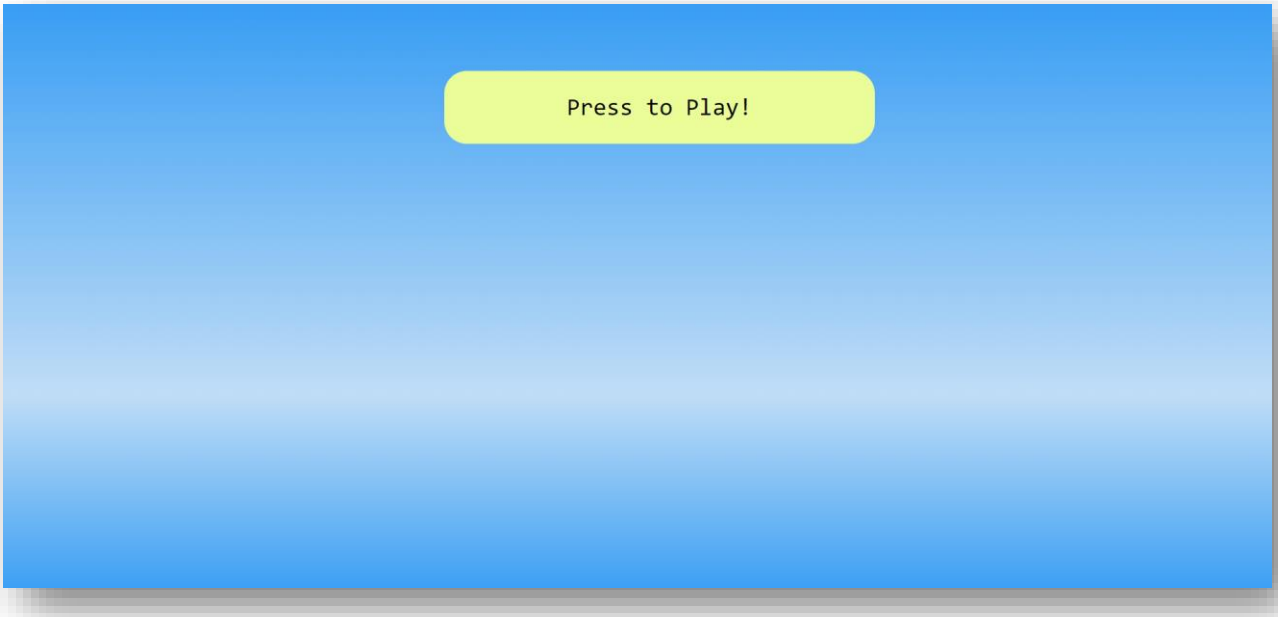
Time to check what we've achieved so far. Open **index.html** in the browser. You will see the blue **game-area** class with the **game-start** section above it.



Next, let's add styles to the **game-start** screen. In **styles.css** we need **absolute position** compared to our **game-area**'s relative position. We want the **game-start** screen to be above our game-area div, so we add **z-index** of **1**. We reduce **width** of the div element by **30%** and we make its **background color green yellow using `rgb(235, 253, 153)`.** Next, we adjust its position to be **100px** from the **top** of its container and **35%** from its **left** side. Now that we have our div box centered in the upper half of the screen, let's style our text inside. **Align** it in the **center** and add **30px padding** so that we have more space around 'Press to Play!'.

```css
.game-start {
    position: absolute;
    width: 30%;
    background-color: rgb(235, 253, 153);
    border-radius: 30px;
    top: 100px;
    left: 35%;
    z-index: 1;
    text-align: center;
    padding: 30px;
}
```

If we open **index.html** in the browser, we should see our changes. We will add an action to our 'Press to Play!' button in the next section of this lesson.



Time to add styles to our **game-score** class where we will keep track of our score. Currently we have just an empty div tag. Let's fix this! Open **index.html**. Hard-code some text inside the div with class **game-score**. We will fix that later.

```html
<!-- score -->
<div class="game-score">1500 pts.</div>
```

Now, let's add some style to our **game-score** class. Open **styles.css** and make its **background black** and its **text white** and **centered.**

```css
.game-score {
    background-color: black;
    color: white;
    text-align: center;
}
```

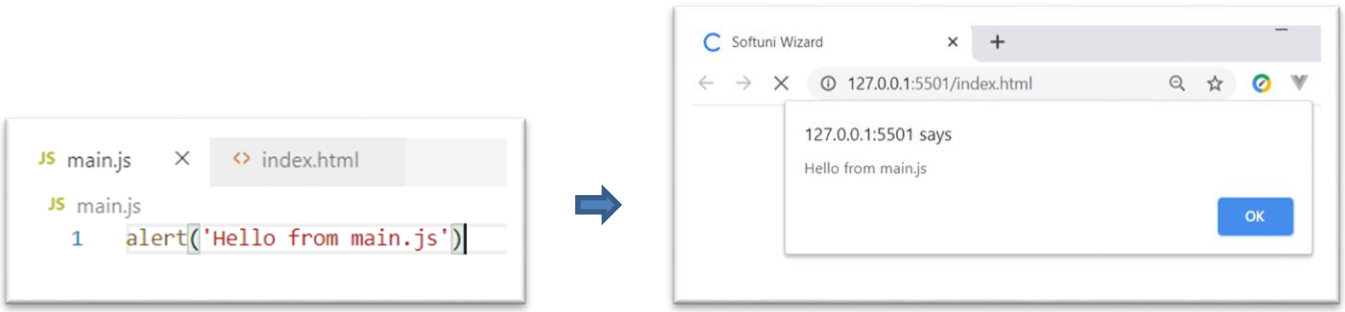
Let's open the browser again. We can see our **game-score** screen has appeared at the top of the page.

We will continue adding styles later. Now let's switch to JavaScript!

1. Player Movement

We already added some base styles to our game screens in the previous step. Now it's time to add scripts to make that content dynamic.

First, we need to assure **main.js** and **index.html** are correctly linked. Open **main.js** and create an alert message.



Now open the browser and check if we have the message. Ok, we are there! You can remove the hello message from **main.js**.

We begin by **selecting** the html elements with JavaScript.

```
1  // select game screens
2  const gameStart = document.querySelector('.game-start');
3  const gameArea = document.querySelector('.game-area');
4  const gameOver = document.querySelector('.game-over');
5  const gameScore = document.querySelector('.game-score');
6
```

It is time to make our game start when 'Press to Play!' is clicked. We need to **attach an event listener** which will fire once our **gameStart** selector receives a click.

```
7  //game start listener
8  gameStart.addEventListener('click', onGameStart);
9
```

We need to define our **`onGameStart`** function. It will be responsible for our game starting. To check if everything is OK, we will display an alert message after the **game-start** div element is clicked. Refresh the browser. After clicking "Press to Play!" you should see the alert message.

```
10 //game start function
11 function onGameStart() {
12     alert('Game started')
13 }
```

You can remove the alert message now. We want to hide the **game-start** screen after the button is clicked. To do that we first add a new class property **hide**. Open **styles.css** and add **display none** to our new class **hide.**

```
31 .hide {
32     display: none;
33 }
```

Go back to **main.js**. Inside our **`onGameStart`** function we need to attach the new class **hide** to our **game-start** screen.

```
10 //game start function
11 function onGameStart() {
12     gameStart.classList.add('hide');
13 }
14
```

Refresh the page and now after "Press to Play!" is clicked, the **game-start** screen should disappear!

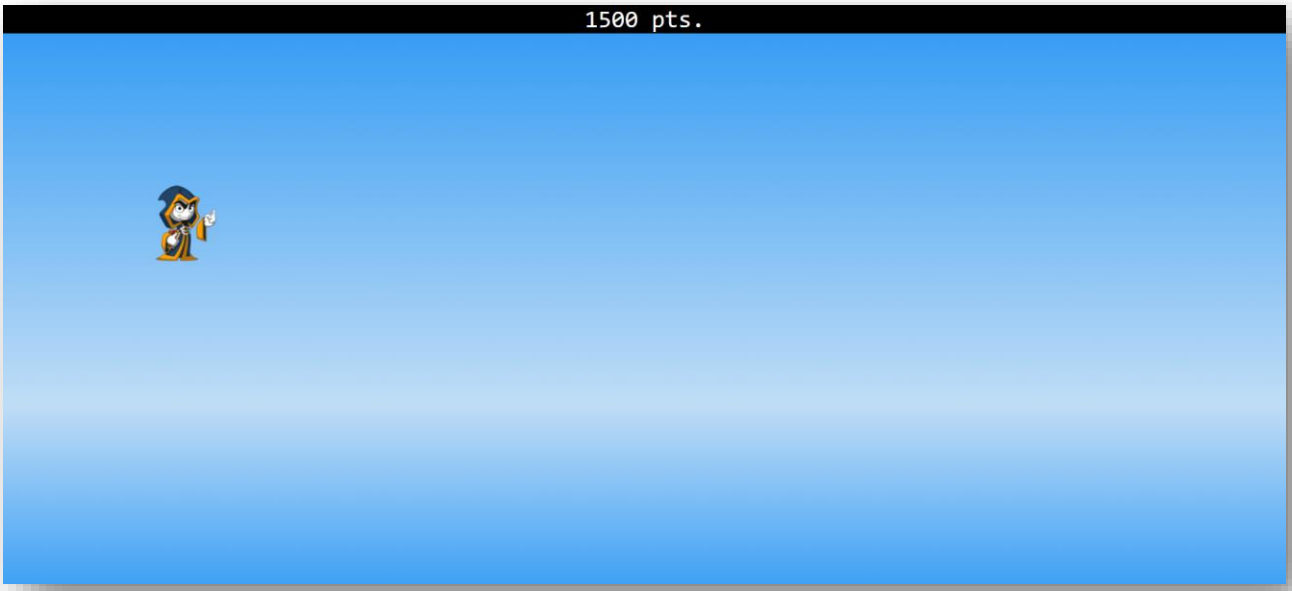
Now we proceed with creating our initial scene and rendering our wizard! You can find the image into the resources. But how to render our wizard? In our **`onGameStart`** function we simply create a new div element with class wizard. We will make that div with the size and background of our picture. We should add absolute position here as well. Before we attach our div with class wizard to the DOM, we need to set initial position - where the picture appears on the screen. For now, let's hard-code 200px top and 200px left, which means our picture will render 200px from the top of its container and 200px from its left border. Later we'll make these values dynamic. Open main.js and initialize our **wizard** inside **`onGameStart`**.

```
    //render wizard
    const wizard = document.createElement('div');
    wizard.classList.add('wizard');
    wizard.style.top = '200px';
    wizard.style.left = '200px';
    gameArea.appendChild(wizard);
}
```

Open **styles.css** and add styles to the **wizard** class.

```
.wizard {
    z-index: 100;
    position: absolute;
    width: 82px;
    height: 100px;
    background-image: url(./images/wizard.png);
    background-size: cover;
}
```

Now it's time to open the browser. After clicking '**`Press to Play!`**', our wizard should appear!



Our next step is to make the wizard fly around on the screen. To do that first we need to handle the user input. We will add **global event listeners** which will track the keys pressed by the user. We will register two events. The first one is **key down** which will be handled by our **onKeyDown** function. The same for **key up** which will be handled by our **onKeyUp** function. Open **main.js**. Add the two event listeners.

```
//global key listeners
document.addEventListener('keydown', onKeyDown);
document.addEventListener('keyup', onKeyUp);
```

Now we have our events registered, but we need somewhere to store the pressed buttons. Let's define an empty object **keys** where we will record the pressed buttons.

```
15  let keys = {};
```

Now define the **onKeyUp** and **onKeyDown** functions which will handle the events.

Note: The event **(e)** is passed by default.

```
30  //key handlers
31  function onKeyDown(e) {
32      keys[e.code] = true;
33      console.log(keys);
34  }
35  function onKeyUp(e) {
36      keys[e.code] = false;
37      console.log(keys);
38  }
```

If we console.log our **keys** object, refresh the browser, press some keys on the keyboard and open the **console** we will see our object **keys** holds the keys pressed so far. When the ***ArrowUp*** key is pressed and it is down our **onKeyDown** function registers that key and sets its value to ***true***. And the opposite, when the key is released our **onKeyUp** function registers the key is released and sets its value to ***false*****.**

```
▶ {ArrowUp: true}                    main.js:33
▶ {ArrowUp: false}                   main.js:37
```

Now we can track whether our keys are pressed or not.

The next step is to define the **game loop**. It will track changes and update the frames on every iteration. We will be using the built-in function **`window.requestAnimationFrame()`**. You can read more about it here.

Define our **`gameAction`** function. This will be the main function which will loop while the game is running. We will pass it to **`window.requestAnimationFrame()`** as a callback. By doing that we will have our **`gameAction`** in an infinite loop.

```
32  function gameAction() {
33      console.log('action');
34      window.requestAnimationFrame(gameAction);
35  }
```

Inside our **`onGameStart`** function we need to pass **`gameAction`** to **`window.requestAnimationFrame()`**

```
18  function onGameStart() {
19      gameStart.classList.add('hide');
20
21      //render wizard
22      const wizard = document.createElement('div');
23      wizard.classList.add('wizard');
24      wizard.style.top = '200px';
25      wizard.style.left = '200px';
26      gameArea.appendChild(wizard);
27
28      //game infinite loop
29      window.requestAnimationFrame(gameAction);
30  }
```

If you open your browser and check the console, you will see we have an infinite loop!

```
3013 action                                                    main.js:32
```

Now let's define our controllers and register the user input. We will play with the arrow keys.

```
32  // game loop function
33  function gameAction() {
34
35      // Register user input
36      if (keys.ArrowUp) {
37
38      }
39
40      if (keys.ArrowDown){
41
42      }
43      if (keys.ArrowLeft) {
44
45      }
46
47      if (keys.ArrowRight) {
48
49      }
50
51      window.requestAnimationFrame(gameAction);
52  }
```

In the scope above we will apply logic which affects the behavior of our wizard. To do that we will need another two global variables. The first one is an object which we will call **player**. It holds initial values for our player. The second one, also object, we will call **game**. It holds values for our game. On every iteration of our game loop we will update its values. Open **main.js** and let's define our new objects.

```
let keys = {};
let player = {};
let game = {};
```

Inside **game** we will define the initial **speed** of our game. Let's set it to **2** for now.

```
let game = {
    speed: 2
};
```

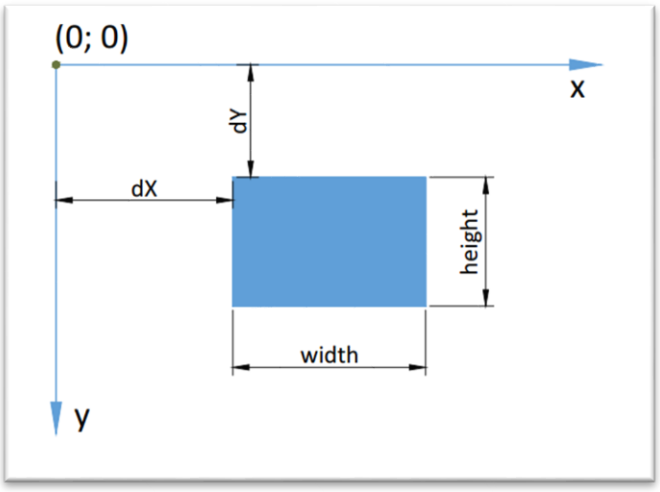
It is time to change the initial position values which we hard-coded in our **`onGameStart`** function earlier. We will set them in our new **player** object.

```
let player = {
    x: 150,
    y: 100
};
```

In the **`onGameStart`** function we assign the initial values which we just defined for our player's starting coordinates. We also need to concatenate the string "**px**" to those values.

```
    wizard.style.top = player.y + 'px';
    wizard.style.left = player.x + 'px';
```

Before we continue and make our wizard move, we need to look more closely at the coordinate system our monitor uses. The **0.0** coordinate is in the upper left corner of the screen. If we want to move our wizard **up**, for example, we need to decrease the **Y** coordinate.



Let's get back to **`gameAction()`** and make our wizard move up. First, we need to define our wizard again because she was defined in another **`onGameStart`** scope. To move her up we need to decrease the **Y** coordinate by subtracting the value of our **`game.speed`** property. After that we assign the new position value to our wizard div element. On the next **iteration** of our **game loop**, **`gameAction()`** will **re-render** the wizard on her new position.

```
// game loop function
function gameAction() {
    const wizard = document.querySelector('.wizard');

    // Register user input
    if (keys.ArrowUp) {
        player.y -= game.speed;
    }

    if (keys.ArrowDown) {

    }
    if (keys.ArrowLeft) {

    }

    if (keys.ArrowRight) {

    }

    wizard.style.top = player.y + 'px';

    window.requestAnimationFrame(gameAction);
}
```

We can move our wizard up. Now just apply the same logic to the other directions we need. If we want to move down, we increase the **Y** coordinate. If we want to move left, we decrease the **X** coordinate. To move right we must increase the **X** coordinate. Do not forget to apply the new values to our **wizard's top** and **left** coordinates**.**

```
    // Register user input
    if (keys.ArrowUp) {
        player.y -= game.speed;
    }

    if (keys.ArrowDown) {
        player.y += game.speed;
    }
    if (keys.ArrowLeft) {
        player.x -= game.speed;
    }
```

```
54
55      if (keys.ArrowRight) {
56          player.x += game.speed;
57      }
58
59      // Apply movement
60      wizard.style.top = player.y + 'px';
61      wizard.style.left = player.x + 'px';
62
63      window.requestAnimationFrame(gameAction);
64  }
```

At this point we can move our wizard around the screen, but it's very slow and boring. Time to introduce a new property in our **game** object. We will call it **`movingMultiplier`**.

```
22  let game = {
23      speed: 2,
24      movingMultiplier: 4
25  };
```

We will apply that multiplier to the **`game.speed`** property. By having more properties in our **game** object, we will have more control over every aspect of the game, especially if we need to refactor the code later.

```
46      // Register user input
47      if (keys.ArrowUp) {
48          player.y -= game.speed * game.movingMultiplier;
49      }
50
51      if (keys.ArrowDown) {
52          player.y += game.speed * game.movingMultiplier;
53      }
54      if (keys.ArrowLeft) {
55          player.x -= game.speed * game.movingMultiplier;
56      }
57
58      if (keys.ArrowRight) {
59          player.x += game.speed * game.movingMultiplier;
60      }
```

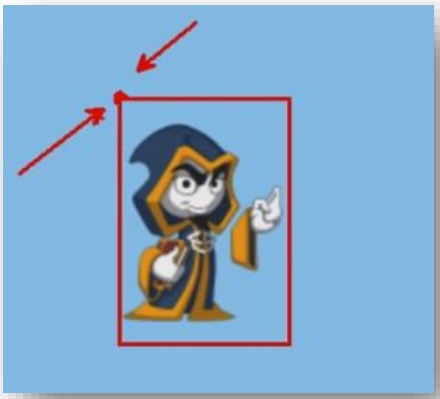
Now it is way better, but we have a bug - our wizard can move **outside** the game area and we don't want that. In our if statements we must check whether the current player coordinates are within the game area. Because our wizard position is **relative** compared to the **absolute** position of the game-area screen, in the **styles.css** file, the 0.0 coordinate is in the upper-left corner of the **game-area** div. Using that knowledge, we will move our wizard up **only** if her **Y** coordinate is a positive value.

```
16      // Register user input
17      if (keys.ArrowUp && player.y > 0) {
18          player.y -= game.speed * game.movingMultiplier;
19      }
```

The same logic will be applied when moving left - only if our wizard's **X** coordinate is a positive value.

```
54      if (keys.ArrowLeft && player.x > 0) {
55          player.x -= game.speed * game.movingMultiplier;
56      }
```

So far so good - we can't move outside our top and left sides. To restrict movement outside the bottom and right sides we need to refactor our code a little.



This is because the 0.0 coordinates of our wizard are at the upper left corner. To limit movement outside the bottom side we need to calculate the wizard's div height. The same needs doing for the right side. We need to calculate the wizard's div width. We will be using the built-in functions HTMLElement.offsetWidth and HTMLElement.offsetHeight. Let's add two new properties to our **player** object - width and height, which we will initially set to 0. Moving back to **`onGameStart()`**, right after the point where we append our wizard to the DOM, we assign new values to our player's height and width which are equal to our current wizard div element's dimensions.

```
17  let player = {
18      x: 150,
19      y: 100,
20      width: 0,
21      height: 0
22  };
```

```
38      gameArea.appendChild(wizard);
39
40      player.width = wizard.offsetWidth;
41      player.height = wizard.offsetHeight;
```

Now we will restrict the movement at the bottom side of the screen. Go back to **`gameAction()`.** We must check if the current value of the player's **Y** plus **`player.height`** is **less** than **`gameArea.offsetHeight`**:

```
56      if (keys.ArrowDown && player.y + player.height < gameArea.offsetHeight) {
57          player.y += game.speed * game.movingMultiplier;
58      }
```

The same logic should be applied to restrict the movement on the right side. This time we must check the **offsetWidth:**

```
63      if (keys.ArrowRight && player.x + player.width < gameArea.offsetWidth) {
64          player.x += game.speed * game.movingMultiplier;
65      }
```

Open the game in your browser and check the results. Our wizard is now locked in the game-area div and it can't move outside of it.

Apply Gravitation

Our wizard flies in the skies. He can easily overcome gravity, but it still affects him! Our next step is to apply the effect of gravity in our game.

In our **`gameAction()`** we will define a new variable **`isInAir`**. It will check if the wizard is above the bottom of the **game-area** screen. If that's ***true***, it will apply gravity equal to our **game.speed** property.

```
47  // game loop function
48  function gameAction() {
49      const wizard = document.querySelector('.wizard');
50
51      // Apply gravitation
52      let isInAir = (player.y + player.height) <= gameArea.offsetHeight
53      if (isInAir) {
54          player.y += game.speed;
55      }
```

We can refactor our code a bit. Now we can move down only if our wizard **`isInAir`.**

```
62      if (keys.ArrowDown && isInAir) {
63          player.y += game.speed * game.movingMultiplier;
64      }
```

Start the game again. Gravitation is now applied, and the wizard is slowly falling.

The Game Scores

It is time to start counting our score. So far, we have two global objects **game** and **player**. While these two contain mostly configuration variables we need another object which will keep track of the changes in our dynamic content. In our **main.js** file's **global scope**, create a **scene** object which will count our score.

```
29  let scene = {
30      score: 0
31  };
```

Go back to **index.html.** It's time to remove the hard-coded values we added before and replace them with a **span** element with class **points**. This element will help us update the state of the score more easily. Inside, add an initial value of **0.**

```
17          <!-- score -->
18          <div class="game-score"><span class="points">0</span> pts.</div>
```

We will update the score with JavaScript, so let's create a **`gamePoints`** selector. We know our class **`points`** lives inside the **`gameScore`** selector, so instead of searching through the entire document, we can find the **points** class using its property:

```
5   const gameScore = document.querySelector('.game-score');
6   const gamePoints = gameScore.querySelector('.points');
```

Ok, we have our **points** class selected. Now we must increment the counter. Inside **`gameAction()`** we will increment our score by 1 which means on every iteration or frame of our infinite loop, we will grant one point.

```
56      //Increment score count
57      scene.score++;
```

At this point we have our html element selected and our score incrementation sorted out as well. The last thing is to apply the changes we've made and update the DOM element. In our **`gameAction()`**, just before the end of the function we will apply our changes.

```
85      // Apply score
86      gamePoints.textContent = scene.score;
87
88      window.requestAnimationFrame(gameAction);
89  }
```

If we start the game again our score will increment by one for every frame.

2. Fireballs

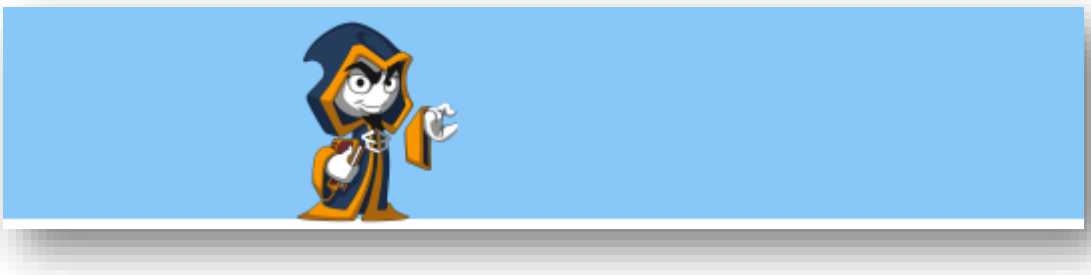
Our wizard shoots fireballs! To do that we have to implement a new controller. Shooting fireballs happens via the **space bar**. Go to **`gameAction()`** and add a new if statement which will check for **space** in the **register user input** section. After **space** is pressed, we need to change our wizard picture. That's how we will simulate animation. To do this we simply need to add a new **class** to our wizard div - **wizard-fire**. We will handle that change in **style.css**.

```
81      if (keys.Space) {
82          wizard.classList.add('wizard-fire');
83      } else {
84          wizard.classList.remove('wizard-fire');
85      }
```

Now, after **space** is pressed the wizard div has a new class property - **wizard-fire.** Open **styles.css** and add styles to our new class. We simply need to **change** the picture and make it **cover** the div.

```
45  .wizard-fire {
46      background-image: url(./images/wizard-fire.png);
47      background-size: cover;
48  }
```

Open the game in the browser now and hold **space**.



Let's add the actual fireball now. Define a new function which will handle fireball behavior. Inside the user input section, where we check the space bar, invoke our new function **`addFireBall()`.**

```
81      if (keys.Space) {
82          wizard.classList.add('wizard-fire');
83          addFireBall();
84      } else {
85          wizard.classList.remove('wizard-fire');
86      }
```

The first step is to render our fireball on the screen. To do that we simply have to make a new div element, like we did with our wizard and add a **fire-ball** class to it. After that we must append it to the DOM. We already have a **gameArea** selector, so let's make use of it.

Note: our function must be declared in the global scope!

```
 98  function addFireBall() {
 99      let fireBall = document.createElement('div');
100      fireBall.classList.add('fire-ball');
101
102      gameArea.appendChild(fireBall);
103  }
```

Open **styles.css** and add styles to our new class **fire-ball.** It will simply load the fireball picture from the recourses. We add **positon absolute** and the default **width** and **height** which are **40px**, which ensures our picture will cover the newly created div.

```
50  .fire-ball {
51      position: absolute;
52      width: 40px;
53      height: 40px;
54      background-image: url(./images/fire-ball.png);
55      background-size: cover;
56  }
```

If we open the browser and check what we've achieved, we will notice two things. Firstly, we do have a fireball, but its position is incorrect. Our wizard should shoot fireballs from her hand. Let's do this. We will need the current player coordinates. We will pass our **player** object to the **addFireBall** function. Go back to the **register input section** and pass the player object.

```
81      if (keys.Space) {
82          wizard.classList.add('wizard-fire');
83          addFireBall(player);
84      } else {
85          wizard.classList.remove('wizard-fire');
86      }
```

We can now use the player object, which holds the current **X,Y** coordinates, in **addFireBall()**. Again, we will be using the **style** properties to render the fireball in the correct place.

To adjust the height at which the fireball will render, we must use **`fireBall.style.top`.** It will be equal to the **`player.y`** coordinate plus **`player.height`** which is the height of the wizard div element in **px**. We must divide it by **3** because we want the fireball to render where the wizard's hand is and, in the end, we simply adjust the height with **5px**. Don't forget to concatenate "**px**" after the expression.

To adjust the fireball **X** coordinate we simply want our player's current **X** plus her **width.**

And the last thing is to assign the **`fireball.x`** coordinate to **`fireBall.style.left`**. Don't forget to concatenate "**px**" after the expression.

In the end your function will look like the one on the screenshot below:

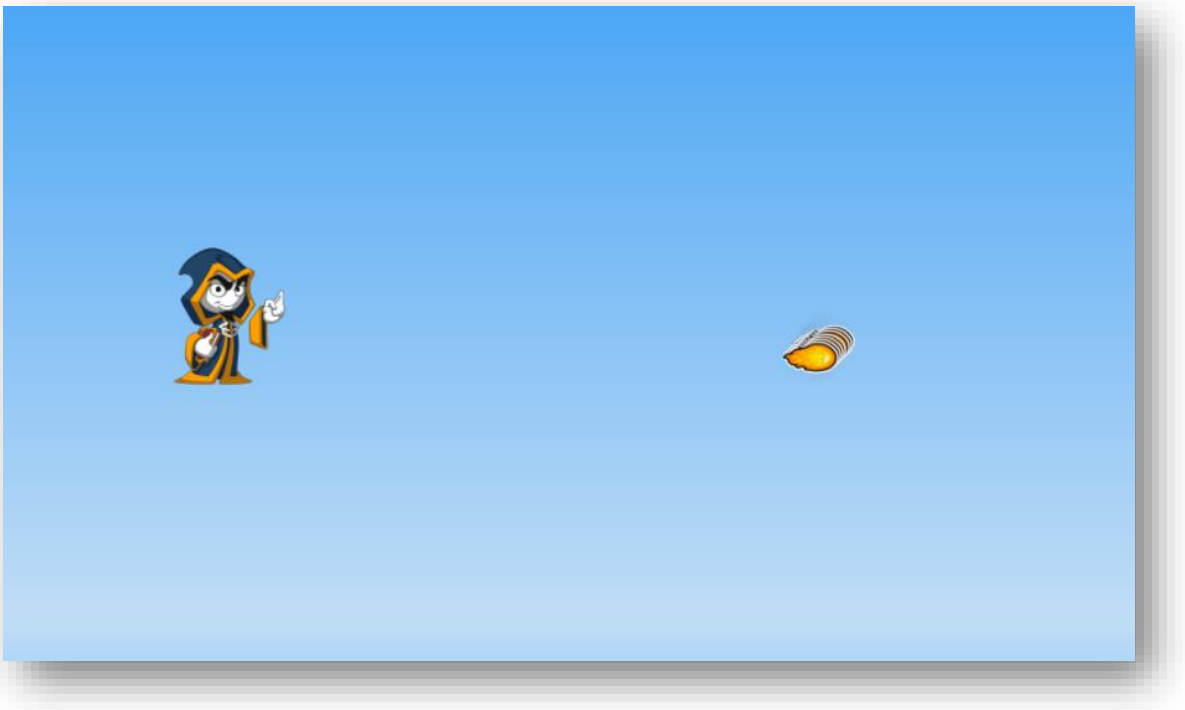
```
98  function addFireBall(player) {
99      let fireBall = document.createElement('div');
100
101     fireBall.classList.add('fire-ball');
102     fireBall.style.top = (player.y + player.height / 3 - 5) + 'px';
103     fireBall.x = player.x + player.width;
104     fireBall.style.left = fireBall.x + 'px';
105     gameArea.appendChild(fireBall);
106 }
```

Open your browser and check the results. We have our fireball rendered at the correct place. We have a little bug. Our wizard is firing too many fireballs. We will fix that later.

Let's get our fireballs flying. To do this we have to modify their position. They will fly only ahead so we need to modify only their **X** coordinate. Go to **`gameAction()`**. We must make a **selector** for all fireballs present on the screen. After we have selected them, we foreach though them and modify their **X** coordinate. For now, we will increment their **X** by our base **`game.speed`**. Then we assign the new **X** coordinate to our **`fireball.style.left`** property. Now on every frame we increment the **`fireball.x`** coordinate.

```
59      // Modify fireballs positions
60      let fireBalls = document.querySelectorAll('.fire-ball');
61      fireBalls.forEach(fireBall => {
62          fireBall.x += game.speed;
63          fireBall.style.left = fireBall.x + 'px';
64      });
```

Open the game in your browser and shoot some fireballs.



Our fireballs are moving but now they exit out of the game area. Same problem we had with our wizard earlier so we will use a similar solution. In the **fireBalls foreach** we must add an if statement. If the current **fireBall.x** coordinate plus **fireBall.offsetWidth** is greater than our **gameArea.offsetWidth,** we must remove the fireball from the screen. To do that we invoke the fireball's parent element and **remove** the child which is our fireball.

```
59      // Modify fireballs positions
60      let fireBalls = document.querySelectorAll('.fire-ball');
61      fireBalls.forEach(fireBall => {
62          fireBall.x += game.speed;
63          fireBall.style.left = fireBall.x + 'px';
64
65          if (fireBall.x + fireBall.offsetWidth > gameArea.offsetWidth) {
66              fireBall.parentElement.removeChild(fireBall)
67          }
68      });
```

You've probably noticed our wizard is faster than his fireballs. We don't want that so an adjustment to fireball speed is required. Introduce a new property in the **game** object. We will call it **`fireBallMultiplier`** with a value of **5.**

```
25  let game = {
26      speed: 2,
27      movingMultiplier: 4,
28      fireBallMultiplier: 5
29  };
```

```
62      fireBalls.forEach(fireBall => {
63          fireBall.x += game.speed * game.fireBallMultiplier;
64          fireBall.style.left = fireBall.x + 'px';
65
66          if (fireBall.x + fireBall.offsetWidth > gameArea.offsetWidth) {
67              fireBall.parentElement.removeChild(fireBall)
68          }
69      });
```

Now we simply multiply our base **game.speed** by **game.fireBallMultiplier.**

We must fix one more thing. Our wizard is firing way more fireballs than we want. We are already using the built-in function **window.requestAnimationFrame()**. We can pass a **timestamp** to our main **gameAction()**. Let's pass it and we will **console.log** the timestamp to see what is happening.

```
53  // game loop function
54  function gameAction(timestamp) {
55      const wizard = document.querySelector('.wizard');
56
57      console.log(timestamp);
```

```
164844.404                                                    main.js:57
164861.07                                                     main.js:57
```

We have the time elapsed in milliseconds. We can use this. Go back to the **player** object. We will add a new property **lastTimeFiredFireball** with initial value of **0.** With that property we will keep the timestamp of our last fireball.

```
18  let player = {
19      x: 150,
20      y: 100,
21      width: 0,
22      height: 0,
23      lastTimeFiredFireball: 0
24  };
```

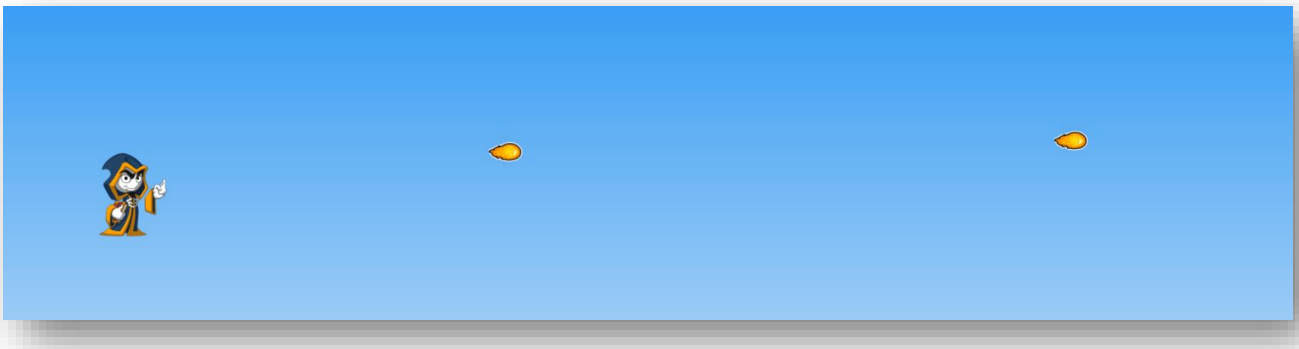
We will need another property this time in our **game** object. Let's call it **fireInterval.** This is the minimum time interval between our fireballs. We set it to **1000ms** which means 1 sec.

```
26  let game = {
27      speed: 2,
28      movingMultiplier: 4,
29      fireBallMultiplier: 5,
30      fireInterval: 1000
31  };
```

Go back to the **register input section**. We must refactor our if statement. Now we want to shoot a fireball only if the **timestamp** minus **`player.lastTimeFiredFireball`** is greater than **`game.fireInterval`**(1 second). If that's ***true*** we must assign a new value to **`player.lastTimeFiredFireball`** which equals the current **timestamp.**

```
95      if (keys.Space && timestamp - player.lastTimeFiredFireball > game.fireInterval) {
96          wizard.classList.add('wizard-fire');
97          addFireBall(player);
98          player.lastTimeFiredFireball = timestamp;
99      } else {
100         wizard.classList.remove('wizard-fire');
101     }
```

Restart the game in the browser. We can only shoot one fireball per second. That's what we wanted!



Adding Clouds

As you know our wizard is flying through the sky. It's time to make that sky looks more realistic by adding clouds to the background. They will be moving from the right side of the screen to the left. In that way we will simulate that our wizard is flying forward. Good, let's get started!

We will need a new property in our **scene** object which will be called **`lastCloudSpawn`**. We don't want the clouds to be spawned all the time like it was with the fireballs in the beginning. In that property we will keep the last time a cloud was spawned. We will use the **timestamp** here as well.

```
33  let scene = {
34      score: 0,
35      lastCloudSpawn: 0
36  };
```

Go to the **game** object. We will need a new property there which we will call **`cloudSpawnInterval`**. Here we set the time it takes for a new cloud to be spawned. We set that value to **3000ms** which is 3 sec.

```
26  let game = {
27      speed: 2,
28      movingMultiplier: 4,
29      fireBallMultiplier: 5,
30      fireInterval: 1000,
31      cloudSpawnInterval: 3000
32  };
```

So far, we have the timestamp of the last cloud we spawned and the time interval between the clouds. In our **`gameAction()`** we will create a new **`div`** element like we did before which we will style with **CSS**. We add a class **`cloud`** to that div element.

```
61      //Increment score count
62      scene.score++;
63
64      // Add clouds
65         let cloud = document.createElement('div');
66         cloud.classList.add('cloud');
```

It is time to add some styles. Open **styles.css** and simply repeat what we did with the other elements. Set a **background image** to the div and **cover** it with that image. Set default values to **width** and **height** which are the sizes of our picture. With the **z-index** we ensure that the clouds will be behind the wizard, not over her.

```
58  .cloud {
59      z-index: 0;
60      position: absolute;
61      width: 200px;
62      height: 200px;
63      background-image: url(./images/cloud.png);
64      background-size: cover;
65  }
```

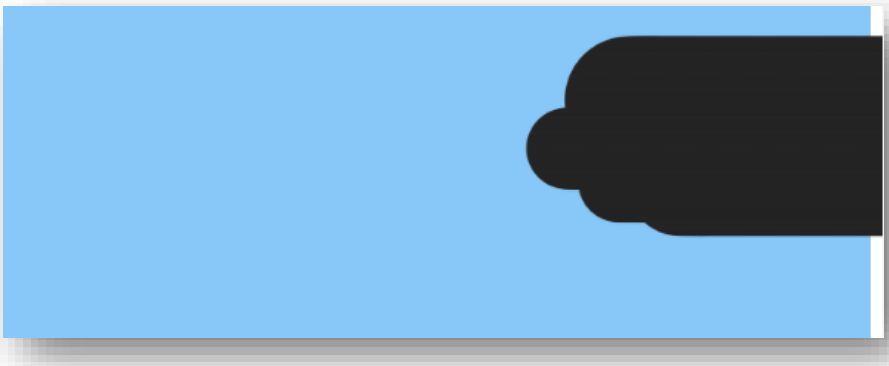
We have our **cloud** div generated with JavaScript and we have styles for it. The next step is to set the initial positon where they spawn and attach them to the DOM. Ok let's do this. We want clouds to spawn on the right side of the screen, so we set the **`cloud.X`** to be equal to our **`gameArea.offsetWidth`**. After that we assign that **`cloud.X`** to the **`cloud.style.left`** property. In the end we append our **cloud div** element to the DOM.

```
64      // Add clouds
65      let cloud = document.createElement('div');
66      cloud.classList.add('cloud');
67      cloud.x = gameArea.offsetWidth;
68      cloud.style.left = cloud.x + 'px';
69
70      gameArea.appendChild(cloud);
```

If we start the game now, we won't see the clouds because we must make them move. We must do the same we did with our fireballs. After the **Add clouds section** we will make another one which we will call **Modify cloud position**. Here we will use the same logic we did with the fireballs. We **select** all clouds that are currently on the screen. We will foreach through them and set the **cloud.x** coordinate to decrease by our base **game.speed** on every frame. In the end we must remove the cloud if it reaches the left part of the screen. We create a statement- if **cloud.x** plus **cloud.offsetWidth** of the div element is zero or less, we remove the element from the DOM.

```
72      // Modify cloud positions
73      let clouds = document.querySelectorAll('.cloud');
74      clouds.forEach(cloud => {
75          cloud.x -= game.speed;
76          cloud.style.left = cloud.x + 'px';
77
78          if (cloud.x + clouds.offsetWidth <= 0) {
79              cloud.parentElement.removeChild(cloud);
80          }
81      });
```

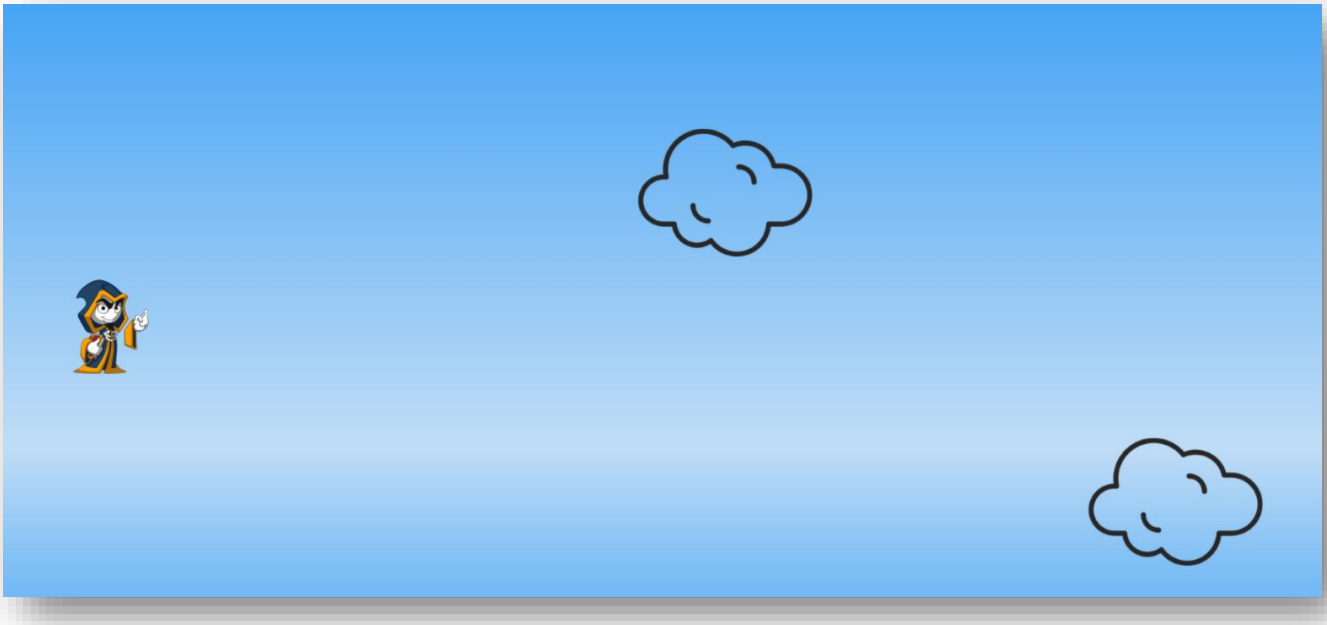
If we start the game now, we will notice that we have the same problem we had with the fireballs - the clouds are spawning on every frame.



This time we will use our **`game.cloudSpawnInterval`**, **`scene.lastCloudSpawn`** and current **`timestamp`**. We will use **`Math.random()`** to make spawn intervals more unpredictable. Now we must make an if statement which will check whether the current **timestamp** minus **`scene.lastCloudSpawn`** is greater than **`game.cloudSpawnInterval`** plus **20000** multiplied by **`Math.random()`.** The cloud will now spawn only if that condition is ***true***. The last thing we are going to do here is to use the **`Math.random()`** again, this time to make the cloud's spawn height more unpredictable.

```
64      // Add clouds
65      if (timestamp - scene.lastCloudSpawn > game.cloudSpawnInterval + 20000 * Math.random()) {
66          let cloud = document.createElement('div');
67          cloud.classList.add('cloud');
68          cloud.x = gameArea.offsetWidth - 200;
69          cloud.style.left = cloud.x + 'px';
70          cloud.style.top = (gameArea.offsetHeight - 200) * Math.random() + 'px';
71          gameArea.appendChild(cloud);
72          scene.lastCloudSpawn = timestamp;
73      }
```

You can start the game again. This time everything is working just fine.



The Bugs – Our Mortal Enemies

The one thing that our code wizard hates most of all are **bugs** and her sole purpose is killing them. Now is the time to implement those bugs in our game. They will spawn on the right side of the screen and move to the left. In the next step of this lesson we will implement the **collision**. After we've done that, we will be able to kill them with fireballs. First things first. Let's render the bugs on the screen.

We will need two new properties: **lastBugSpawn** in the **scene** object and **bugSpawnInterval** in the **game** object.

```
26  let game = {
27      speed: 2,
28      movingMultiplier: 4,
29      fireBallMultiplier: 5,
30      fireInterval: 1000,
31      cloudSpawnInterval: 3000,
32      bugSpawnInterval: 1000
33  };
```

```
34  let scene = {
35      score: 0,
36      lastCloudSpawn: 0,
37      lastBugSpawn: 0
38  };
```

Inside **`gameAction()`** we make a new section **Add bugs** using the same logic we did with the other elements. We create a new div element with JavaScript, we style it and append it to the DOM. Then we reduce the spawn interval. In that way we will have more bugs than we can kill. Our game will be more interesting.

```
66      // Add bugs
67      if (timestamp - scene.lastBugSpawn > game.bugSpawnInterval + 5000 * Math.random()) {
68          let bug = document.createElement('div');
69          bug.classList.add('bug');
70          bug.x = gameArea.offsetWidth - 60;
71          bug.style.left = bug.x + 'px';
72          bug.style.top = (gameArea.offsetHeight - 60) * Math.random() + 'px';
73          gameArea.appendChild(bug);
74          scene.lastBugSpawn = timestamp;
75      }
```

Open **styles.css** and add styles to our new class **bug**

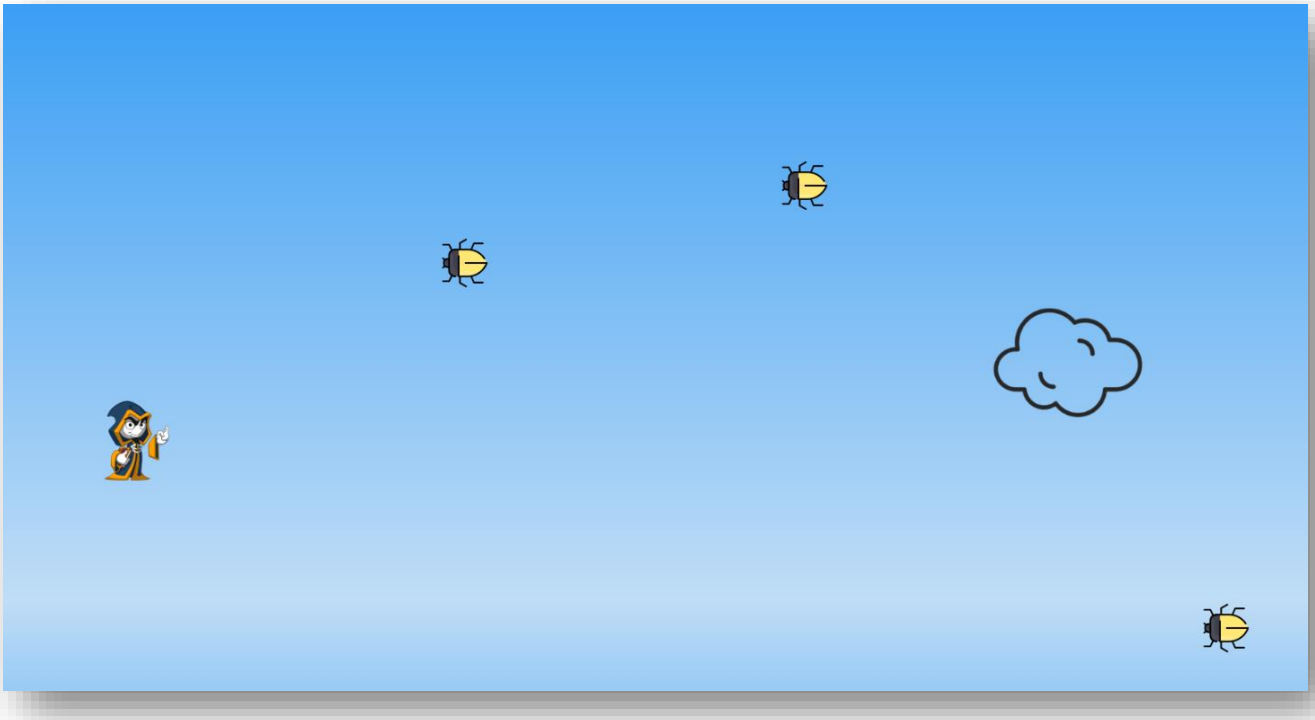
```
.bug {
    position: absolute;
    width: 60px;
    height: 60px;
    background-image: url(./images/bug.png);
    background-size: cover;
}
```

Open the game in the browser. Our bugs are spawning correctly.

Now let's make them move. We will make a **modify bug position** section and apply the same logic we used with the clouds.

```
88      // Modify bug positions
89      let bugs = document.querySelectorAll('.bug');
90      bugs.forEach(bug => {
91          bug.x -= game.speed * 3;
92          bug.style.left = bug.x + 'px';
93          if (bug.x + bugs.offsetWidth <= 0) {
94              bug.parentElement.removeChild(bug);
95          }
96      });
```

Start the game again. Our bugs are now moving.



Collision Detection

Collisions are an important part of every game and ours is no exception. If you'd like to read more about collision and collision detection you can check out this article. In layman's terms, we need to detect if two of our elements overlap.

First, we need to define a new function in the **global** scope of our **main.js** file which we will call **isCollision.** With this function we will determinate if two of our elements overlap. Here we will use another built-in function Element.getBoundingClientRect(). Our function will need two parameters, **firstElement** and **secondElement**. This two elements are DOM elements, so we can call the built-in function. First let's just console.log and see the result of that function.

```
163  function isCollision(firstElement, secondElement) {
164      let firstRect = firstElement.getBoundingClientRect();
165      let secondRect = secondElement.getBoundingClientRect();
166      console.log(firstRect);
167  }
```

We must invoke our function from somewhere. For now, just to **test**, we will invoke it with the space bar.

```
if (keys.Space && timestamp - player.lastTimeFiredFireball > game.fireInterval)
    wizard.classList.add('wizard-fire');
    addFireBall(player);
    player.lastTimeFiredFireball = timestamp;

    isCollision(wizard, wizard);
```

Start the game, hit space and see the result of the console.log.

I will eat ice-cream this afternoon wuillpliy

```
▼DOMRect {x: 158, y: 190, width: 82, height: 100, top: 190, …}
   bottom: 290
   height: 100
   left: 158
   right: 240
   top: 190
   width: 82
   x: 158
   y: 190
  ▶ __proto__ : DOMRect
```

What we see are the current **coordinates** of our wizard. We will use that to detect if there is collision between wizard and bugs or fireballs and bugs.

Let's get back to **`isCollision()`**. We simply check if the first element is **outside** the second element. To make that true we **inverse** the logic.

```
161 function isCollision(firstElement, secondElement) {
162     let firstRect = firstElement.getBoundingClientRect();
163     let secondRect = secondElement.getBoundingClientRect();
164
165     return !(firstRect.top > secondRect.bottom ||
166         firstRect.bottom < secondRect.top ||
167         firstRect.right < secondRect.left ||
168         firstRect.left > secondRect.right);
169 }
```

If our function returns ***true,*** we have **collision.** Let's test this. Go to **`gameAction`** and after the user input make a new section - **Detect collision.** We will test our function with our wizard and the bugs first. We **selected** our bugs earlier**.** We can foreach over them and for every bug we will check whether we have collision with our wizard. This will be updated on every frame. Now we will check if we have a collision message in the browser console.

```
151     // Collision detection
152     bugs.forEach(bug => {
153         if (isCollision(wizard, bug)) {
154             console.log('collision');
155         }
156     });
```

Open the game in your browser and check if we have the test message when we move though bugs.

```
139 collision
```

So far so good. We know we've been hit by a bug. Now the game must end so let's implement our **gameOverAction** function. Go to the **scene** object. There we must add a new Boolean property called **isActiveGame** with default value **true.** This will help us determine when to stop the game and show the **game-over** screen.

```
35  let scene = {
36      score: 0,
37      lastCloudSpawn: 0,
38      lastBugSpawn: 0,
39      isActiveGame: true
40  };
```

In the **global** scope define our **gameOverAction()**. It simply changes the state of **scene.isActiveGame** from **true** to **false** and removes the **hide** class property from the **game-over** screen.

```
169 function gameOverAction() {
170     scene.isActiveGame = false;
171     gameOver.classList.remove('hide');
172 }
```

Now we need mechanics to stop the game. We can do this by adding an if statement inside our **gameAction()** and checking the state of **scene.isActiveGame:** if it's **true** we will invoke the next iteration of our endless game loop, if it's **false** our animations will stop.

```
166     if (scene.isActiveGame) {
167         window.requestAnimationFrame(gameAction);
168     }
```

Now we need to remove console.log in the **Collision detection** section and invoke our **gameOverAction()**

```
152     // Collision detection
153     bugs.forEach(bug => {
154         if (isCollision(wizard, bug)) {
155             gameOverAction();
156         }
157     });
```

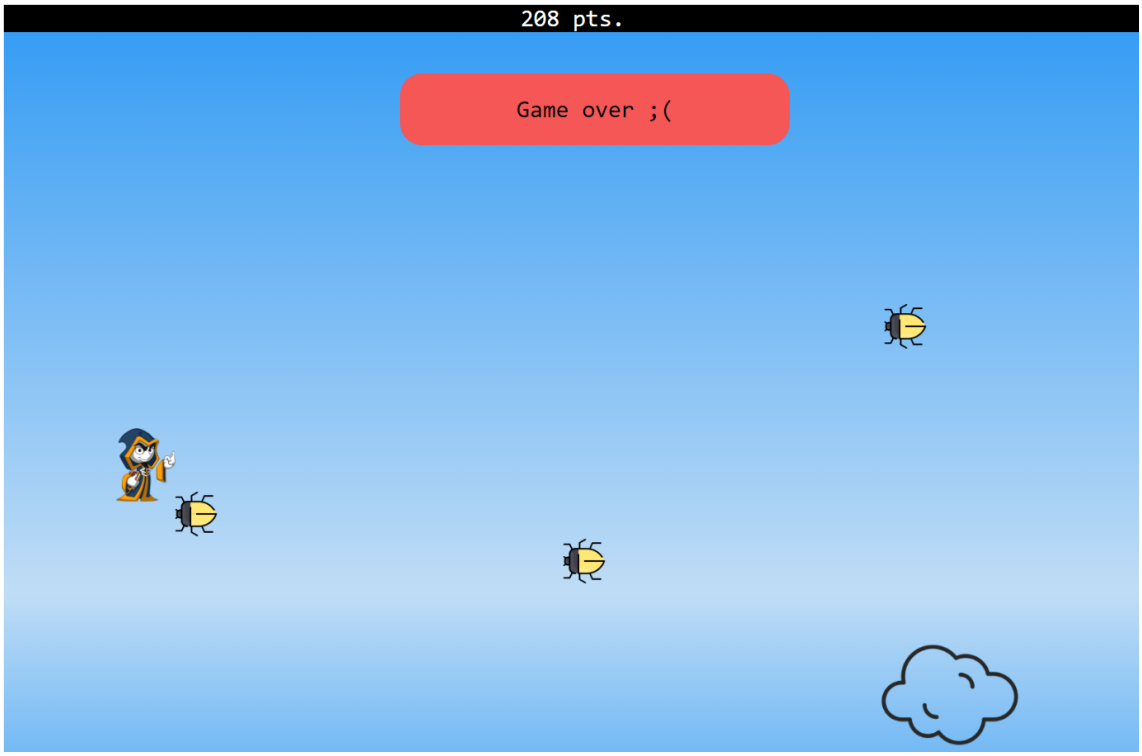
Go back to **index.html**. We will add a class **hide** to that screen which means it will stay hidden until we remove the **hide** class from our **`gameOverAction()`**

```
23          <!-- game over -->
24          <div class="game-over hide">Game Over ;(</div>
```

Finally we need to add styles to that **game-over** screen. Actually we can reuse the ones from the game-start screen and just change the color from green to red.

```
25  .game-over {
26      position: absolute;
27      width: 50%;
28      background-color: red;
29      top: 100px;
30      left: 25%;
31      z-index: 1;
32      text-align: center;
33      padding: 30px;
34  }
```

It's time to start the game. We have a game over screen now!



The last thing is to detect collision between our fireballs and bugs. Go to the **collision detection** section. Inside the loop we will make another one. This time we will foreach all the **fireballs** on the screen and check if there is collision between them and the **bugs**. For now we will console.log a test message.

```
152     // Collision detection
153     bugs.forEach(bug => {
154         if (isCollision(wizard, bug)) {
155             gameOverAction();
156         }
157 
158         fireBalls.forEach(fireBall => {
159             if (isCollision(fireBall, bug)) {
160                 console.log('bug killed!');
161             }
162         })
163     });
```

Open the game in the browser and try to shoot some bugs.

```
12 bug killed!                                                    main.js:160
```

Ok we have it! We shot a bug and now we want a bonus score! Go to the **game** object once again and add a new property named **bugKillBonus** with default value of **2000**.

```
26  let game = {
27      speed: 2,
28      movingMultiplier: 4,
29      fireBallMultiplier: 5,
30      fireInterval: 1000,
31      cloudSpawnInterval: 3000,
32      bugSpawnInterval: 1000,
33      bugKillBonus: 2000
34  };
```

Now when we hit a bug with a fireball our score will increase by 2000.

```
153     // Collision detection
154     bugs.forEach(bug => {
155         if (isCollision(wizard, bug)) {
156             gameOverAction();
157         }
158 
159         fireBalls.forEach(fireBall => {
160             if (isCollision(fireBall, bug)) {
161                 scene.score += game.bugKillBonus;
162             }
163         })
164     });
```

The last thing here is to remove the bug and fireball from the DOM. We've done that before. Nothing new here.

```
153     // Collision detection
154     bugs.forEach(bug => {
155         if (isCollision(wizard, bug)) {
156             gameOverAction();
157         }
158
159         fireBalls.forEach(fireBall => {
160             if (isCollision(fireBall, bug)) {
161                 scene.score += game.bugKillBonus;
162                 bug.parentElement.removeChild(bug);
163                 fireBall.parentElement.removeChild(fireBall);
164             }
165         })
166     });
```

IV. Sky is the Limit*

Our game is super awesome now, but it is far from hyper mega awesome! Try to implement bonuses, different levels, lives. Use your imagination to its full potential!