Tutorial 6

Author: Kia Kalani
Contributors: Ali Lezzaik, Alireza Teimoori

Introduction:

The main focus of this tutorial is to make the enemies and the bullet.

Additionally, by the end of this tutorial you should expect to be able to make a basic space invader game and if necessary expand the components of it.

Bullets

In order to make this game more aesthetically appealing, we will be using images to load the bullets. Therefore we are going to make a simple `Bullet` class that extends our `GameObject` that we created earlier on.

Same as the player, we would have x, y, width and height components for this object.

The code is straightforward for making the bullet. For updating the bullets, we essentially subtract 20 from the bullet y position so it goes up.

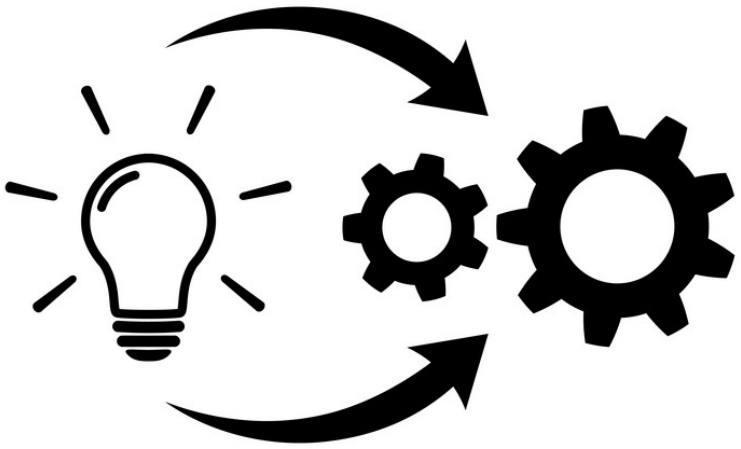
Therefore, in terms of code, it would look similar to the following:

```
class Bullet extends GameObject{

    constructor(x,y) {
        super(x,y);
        this.image = loadImage('img/bullet.jpg');
        this.width = 10;
        this.height = 30;
    }

    render() {
        image(this.image, this.x, this.y, this.width, this.height);
    }

    update() {
        this.y -= 20;
    }
}
```

A simple Game List Implementation

By having the bullet class done, we are going to move onto implementing it inside the player class. Before doing so, we need a list implementation in order to have multiple bullets inside the game.

Although the implementation we are going to use is not the most efficient, it will be sufficient enough for a simple game such as space invader. We would essentially use an array and push and remove elements to it. Inside the constructor, we define an empty array.



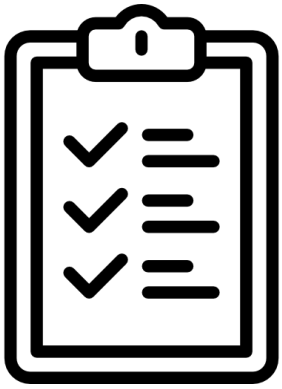
Additionally, for adding an element we are going to make a call to the push method. However, for removing an element, we would deal with it the same as we would deal with the array lists.

Essentially, we store the object into a variable and shift everything down by one. In terms of code, it would look like the following:

```
    remove(index) {
        let len = this.__items.length;
        let element = this.__items[index];
        for (let i = index;i<len-1;i++) {
            this.__items[i] = this.__items[i+1];
        }
        this.__items.pop();
        return element;
    }
```

Finally for getting the size of the elements, we would return the length of the items inside the array.

Creating Bullets Within the Game



By having our `GameList` class, we can now get into defining the bullets functionality within the player. We would now create a `GameList` for bullets. This can be done inside the constructor.

By having that defined, now we would have to render all of the bullets inside the gameList element we have defined.
We essentially loop for all of the bullets and render them like the following code:

```
for (let i =0;i< this.bullets.size();i++) {
    this.bullets.get(i).render();
}
```

Additionally, we apply the same concept to update the bullet positions. However, the bullets still do not exist.

So we would want to have a bullet appear whenever we release our space key.

In order to do so, we make a small modification to the switch case we had for the player movement. we are going to add the following code to it.

```
case " ":
    this.bullets.add(new Bullet(this.x+ (this.width/2), this.y));
    break;
```

This way, a bullet would appear in the middle of the player and would start moving up.

Everything seems fine; however, there is one slight issue. The bullets would never get removed and keep going infinitely. In other words, we have to destroy the bullets when they go above the point `height` of them. That way, the game would not end up crashing due to running out of memory.

For doing that we are going to modify the `update` method of the player and add the following to it.

```
for (let i = this.bullets.size()-1; i >-1;i--) {
    if (this.bullets.get(i).y < -30) {
        this.bullets.remove(i);
    }
}
```

You may wonder why a loop in the opposite way? The answer is simple. If we move from the front and remove a bullet, a bullet can be removed and the items would shift one down.

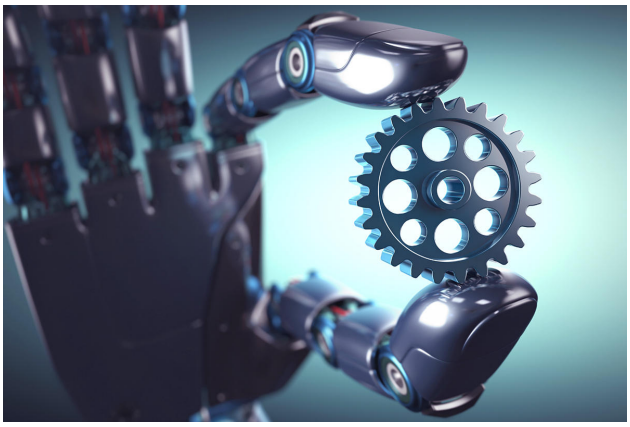
That is a problem because when they shift down, some of those bullets would be disregarded and not removed.

At this point we have our bullet fully functional. Now we need to get to other components of the game.

Mobs

Mobs or the so called invaders are the objects we are going to be making within the game. Mobs are just like the bullets but they come from the opposite side and have four components of x, y, width, and height. We have almost gone into the whole idea of making them.

Therefore, we are going to skip from the methodology of making them. Instead, we need to figure out a way to make the mobs come from the other side in a random time passage.



The simplest way is to use the current time of the system in milliseconds. However, for implementation of it, we are going to create a class called `MobGenerator`.

Now we do need the components of the `GameObject` for this class but there is no need for the x and y position.

This class needs three main components:

1. The first one is a `GameList` that would be containing all of the mobs.
2. The second one is the current time
3. The third one is in milliseconds how many seconds we want to wait before generating a new mob.

The current time in milliseconds would be derived from the following method:

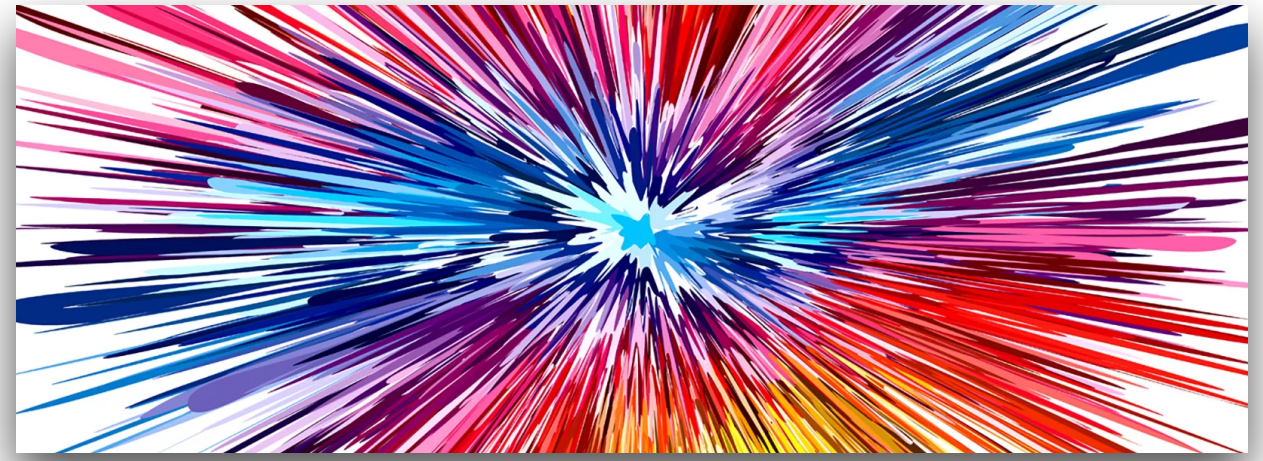
```
getCurrentTimeMiliSeconds() {
    return new Date().getTime();
}
```

We first have to render and update the mobs the same way we have done inside the `player` class through `render` and `update` methods.

Afterwards, we would need to make a method for adding mobs to the mobs list. This can be done by comparing the current time with the time we had saved inside the constructor and if it is more than the milliseconds we had defined as a random value, we would essentially update the value of the time inside the constructor with the current time and update the random number of milliseconds with a new random value and add a mob in a random position inside the canvas in terms of x and `-height` for the y coordinate. In terms of code, it can look similar to the following:

```
addMob() {
    if ( this.getCurrentTimeMiliSeconds() - this.now >= this.howManySeconds) {
        this.now = this.getCurrentTimeMiliSeconds();
        this.howManySeconds = nextInt(1, 5)*1000;
        this.mobs.add(new Mob(nextInt(1,MENUWIDTH) - 80, -80));
    }
}
```

Now we can add this method inside the update method to make sure this class would keep adding mobs by the passage of time.

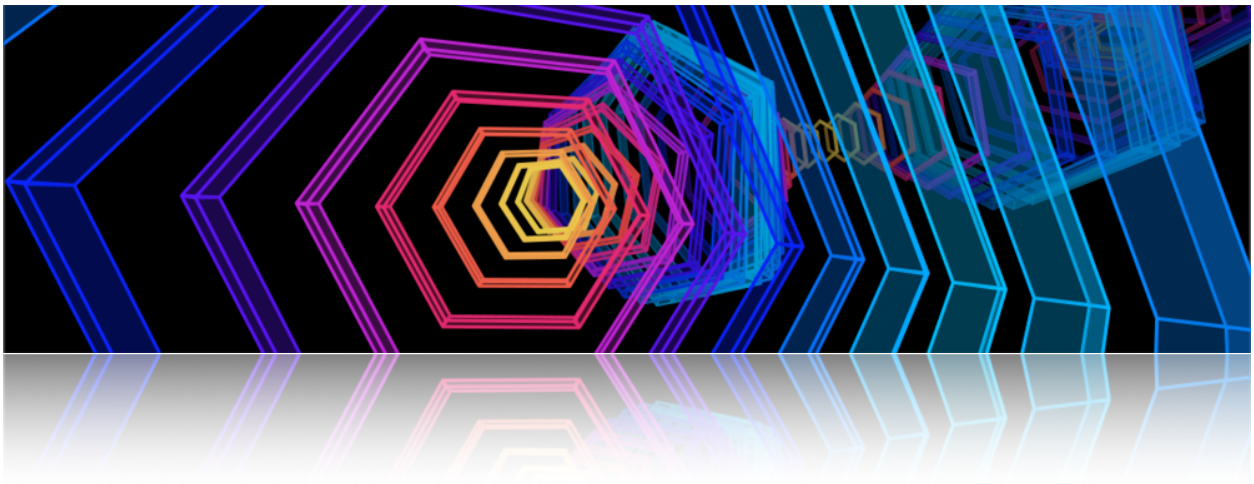


Now we will define an instance of this class inside the `GamePage` menu and do all of the events related to it. By running the `index.html` you notice that the game has mobs and you can shoot at them. However, we still have not implemented any method to handle the case when bullets collide with the mobs.

Making Bullets Hit Mobs

For making the bullets removed and the mobs removed by bullet collision and increasing points, we need to make a method within the player class called `bulletHitMob`.

This method would take the collision detector and the mobs as parameters. It will remove the bullets and the mobs that have collided with each other and adds 5 points to the player's score.



The idea of reverse looping has been specified before. We would just do nested loops to check for the collision and remove the collided elements.
The code would end up looking like the following:

```
bulletHitMob(mobs, collDetector) {
    for (let i = mobs.size() -1; i > -1;i--) {
        let mob = mobs.get(i);
        for (let j = this.bullets.size() -1; j > -1; j--) {
            let bullet = this.bullets.get(j);
            if (collDetector.collidedRects(mob.x,mob.y, mob.width, mob.height,
bullet.x, bullet.y, bullet.width, bullet.height)) {
                mobs.remove(i);
                this.bullets.remove(j);
                this.score += 5;
            }
        }
    }
}
```

Now we would call this method inside the update method of the `GamePage` and pass in the mobs list from the mob generator and the collision detector of the menu as arguments.

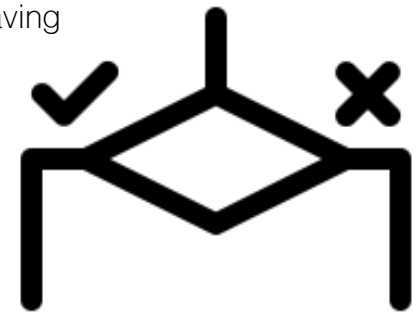
Running the game again, the player can now shoot and increase his score accordingly. However, now this game has no losing condition.

Making a Losing Condition

We can implement a life element for the player and make the player lose by losing all of the lives. However, that would be an additional element that could be implemented accordingly and we would just give the player 1 life in the whole game.

The simple way of checking for the losing condition is exactly the same as checking for the bullets condition. However, this time there would be a single loop that would remove the mob and change the scene to `GameOver` if the mob collides with the player.

Conclusion



Using `p5js` for making games is a convenient idea due to having bunch of built in functionalities that can be used to make really impressive contents. Although making a game such as space invader is relatively simple and does not need to much experience in programming, is a good practice for improvement of JavaScript skills and teaching different built ins that `p5js` provides.

That sums up the last tutorial!

For gaining access to the source code, please refer to: https://github.com/kiakalani/P5JSTutorial/tree/main/tutorial6/code