Project 1.5 Tutorial

EMDA 203

Instructions

1. Download the `Project_1_5_Tutorial.zip` archive from the Moodle site's Project_1_5 assignment section, and unzip the archive.
2. Open your `Users` folder (your P drive) and find your `EMDA_203_Starterkit` folder. Once there, open up the `builds` folder, and delete the old `Project_1_5` folder.
3. Copy the entire unzipped `Project_1_5_Tutorial` folder you just unzipped into your `builds` folder.
4. To avoid confusion, delete the zip archive and the unzipped copy of the new folder from your `Downloads` folder (or wherever you downloaded them).
5. Now, open up `Project_1_5_Tutorial_Code_2018.html` in Chrome and TextWrangler. As you work through the project, answer the questions below.

Overview

Our main goal this week is to add collision detection to our game. Before we get into that, however, let's start by reorganizing and reworking some of our previous code.

Renaming

To begin with, we've renamed the following functions to be more descriptive.

- Renamed `handleKeyboardInput()` to `moveHero()`
- Renamed `targetBoudaryCheck()` to `boundaryCheckTargets()`
- Renamed `heroBoundaryCheck()` to `boundaryCheckHero()`

`init()`

Is nothing sacred?? We've split our old `init()` function into two functions: `init()` and `initGame()`. Now, we can call them separately as circumstances require. Our old `init()` function retains its same catchy name, but now initializes only those things that truly need to happen once – namely, the `Stage` and the `Ticker`. Our `initGame()` function will initialize our gameplay variables and display objects. We'll call `initGame()` whenever we want to restart the game. Now, we can have a game over condition with an option to restart the game without reloading the entire page.

1. Where is `initGame()` called for the first time?

`initGame()`

`initGame()` does two things. It sets/re-sets all of our game-play variables and it sets/re-sets all of our objects that will be displayed on the Stage. Our game-play variables are all new this week and are declared in global scope: `score`, `gameClock`, and `gameOver`. Note that `score` and `gameClock` contain numbers while `gameOver` is a *boolean* that is initialized as false.

2. What might the `gameOver` variable be used for, and why set it to false?

Our display objects are our hero, our background, and our targets along with a new object that will display the player's score. In order to make our code easier to read, we've lumped our old code into two new functions: `makeBackground()` and `makeHero()` to go along with our `makeTargets()` function.

3. Do you see how writing these new functions makes `initGame()` easier to read? Just nod your head, and say yes. Thanks.

Our final display object, `scoreText`, contains an instance of the `createjs.Text` class. The Text class lets us display text on the Stage.

4. Can you guess what the three parameters are for the new `createjs.Text` call on line XXX? Check your guess against the createjs API to see if you're right. Do a web search for "createjs.Text" to check your answer.

5. Can you position the Text object to a better part of the screen?

6. Can you change the Text object's color? Web search "hex color" to spec a new color.

makeTargets()

A *switch statement* lets us check multiple *if* conditions in a single statement. Previously, we used if( ){ }else if( ){ }else{ } which can be difficult to read. Our previous code:

```
if(whichTarget === 0){
	targetsArray[i] = new createjs.Bitmap("images/target01.png");
}else if(whichTarget === 1){
	targetsArray[i] = new createjs.Bitmap("images/target02.png");
}else{
	targetsArray[i] = new createjs.Bitmap("images/target03.png");}
```

We've replaced it with a *switch statement* to check whether `whichTarget` contains 0,1 or 2:

```
switch(whichTarget){
	case 0:
		targetsArray[i] = new
createjs.Bitmap("images/target01.png");
		break;
	case 1:
		targetsArray[i] = new
	createjs.Bitmap("images/target02.png");
		break;
	case 2:
		targetsArray[i] = new
	createjs.Bitmap("images/target03.png");
		break;
	default:
```

```
            console.log("Uh-oh, we've got a problem at the switch!");
            break;
}// end switch
```

collisionGnome.js

Whenever we want to check if a display object has collided with another display object, we need to outfit each display object with something known as a *collider*. A *collider* sets up properties on an object that can be checked against another object's *collider* properties to see if a collision has occurred. The simplest colliders set up an object known as a *bounding box* based on an object's height and width. If one bounding box overlaps another bounding box… collision! Our `collisionGnome` library lets us assign colliders with the `collisionGnome.addCollider(sprite, hitboxRatio)` method. For the sprite parameter, we pass the thing we want to attach a collider to. A hitboxRatio of 1 will give us a hitbox that matches the pixel dimensions of our Bitmap. We can see a visual representation of an object's bounding box by calling `collisionGnome.setDebug(true)`.

7. What objects are assigned colliders?

8. Where are the colliders assigned?

9. Where is the setDebug(true) method called?

10. Why is it useful to show the bounding boxes as part of the debugging process?

11. When you move the character around and collide with the targets why do the bounding boxes remain?

`checkCollisions()` Part I: Counting Down a *for* Loop

The purpose of checkCollisions() is to check if our hero has collided with one or more of our targets. To do this, we'll walk through the targetsArray using a for loop. Previously, in our for loops, we've initialized our `i` at `0` and used `i++` to increase the value of `i` by `1` after running the for loop's block statement. Now, we'll learn how to initialize `i` as the `length` of the `targetsArray` and count *down* after each loop by using `i--`. Our new count down for loop will look like this:

```
    for(var i = targetsArray.length-1; i>=0; i--){ … }
```

First of all, we initialize `i` as `targetsArray.length-1`. Why? Because `targetsArray.length-1` is equal to the *index* of the last thing in our array. The *last index* in an array is *always* one less than the *length* of the array. Secondly, our condition to evaluate will be `i >= 0;` Why? Because we started with last thing in the array, we want to end with the first thing in the array stored at index `0`. When `i === 0`, our condition will be `true`.
When `i === -1`, it will be `false`.

`checkCollisions()` Part II: `collidesWith()`

The `collidesWith()` function from collisionGnome compares an object's bounding box position with another object's box. If our hero collides with the target at index i of the targetsArray, collidesWith() returns true. If there is no collision, we get false. Whenever our hero hits a target, we want three things to happen. Each one of these things requires a separate line of code.

- We want the score variable to increase by `1`. (line 2XX)
- We want the target that's been hit to disappear from the stage. (line 2XX)
- We want to *completely* remove the hit target's data from the `targetsArray`. (line 2XX)

The score increase is pretty straightforward. To remove an object from the stage, we simply call the Stage method, `removeChild()` with the object to be removed passed as an argument in the parentheses: `myStage.removeChild(targetsArray[i]);`

To remove our target from the `targetsArray`, we'll use the `Array.splice()` method. The syntax for splice is:

```
arrayName(index, numberOfThingsToRemove)
```

Our checkCollisions() code reads:

```
targetsArray.splice(i, 1);
```

12. How would you translate the line above?

`updateScoreText(), checkGameOver(), handleGameOver()`

13. What does `updateScoreText()` do?

14. Describe how `checkGameOver()` is triggered and what it does once called.

15. How does `handleGameOver()` use the `gameClock` variable?

16. How does `handleGameOver()` re-set the game?