Week 14

Quick Review: We are using JavaScript to develop Games.
Our platform is Phaser, and our editor is Brackets.

A sprite sheet is a series of images combined into a larger image. Usually the images are frames of an animation, thus a single image inside a sprite sheet is called frame.

From the Phaser Docs:
new Sprite(game, x, y, key, frame)

Sprites are the lifeblood of your game, used for nearly everything visual.

At its most basic a Sprite consists of a set of coordinates and a texture that is rendered to the canvas.

They also contain additional properties allowing for physics motion (via Sprite.body), input handling (via Sprite.input),

events (via Sprite.events), animation (via Sprite.animations), camera culling and more.

Tween:

new Tween(target, game, manager)

A Tween allows you to alter one or more properties of a target object over a defined period of time. This can be used for things such as alpha fading Sprites, scaling them or motion. Use Tween.to or Tween.from to set-up the tween values. You can create multiple tweens on the same object by calling Tween.to multiple times on the same Tween. Additional tweens specified in this way become "child" tweens and are played through in sequence. You can use Tween.timeScale and Tween.reverse to control the playback of this Tween and all of its children.

In our last exercise we saw how we can use a tween to animate our tiles…

Using tweens to animate tiles

While just showing the tile is an option, it's just a static image. No matter how cute your tiles look, they are just flat images if you don't animate them a bit.

That's why you are going to learn to create animations with Phaser tweens.

Tweens are a Phaser key feature. You will use a lot of tweens in the making of this game and more in general in the making of every game which requires animations.

Let's choose an animation speed, in milliseconds, which will be saved in

`gameOptions` object:

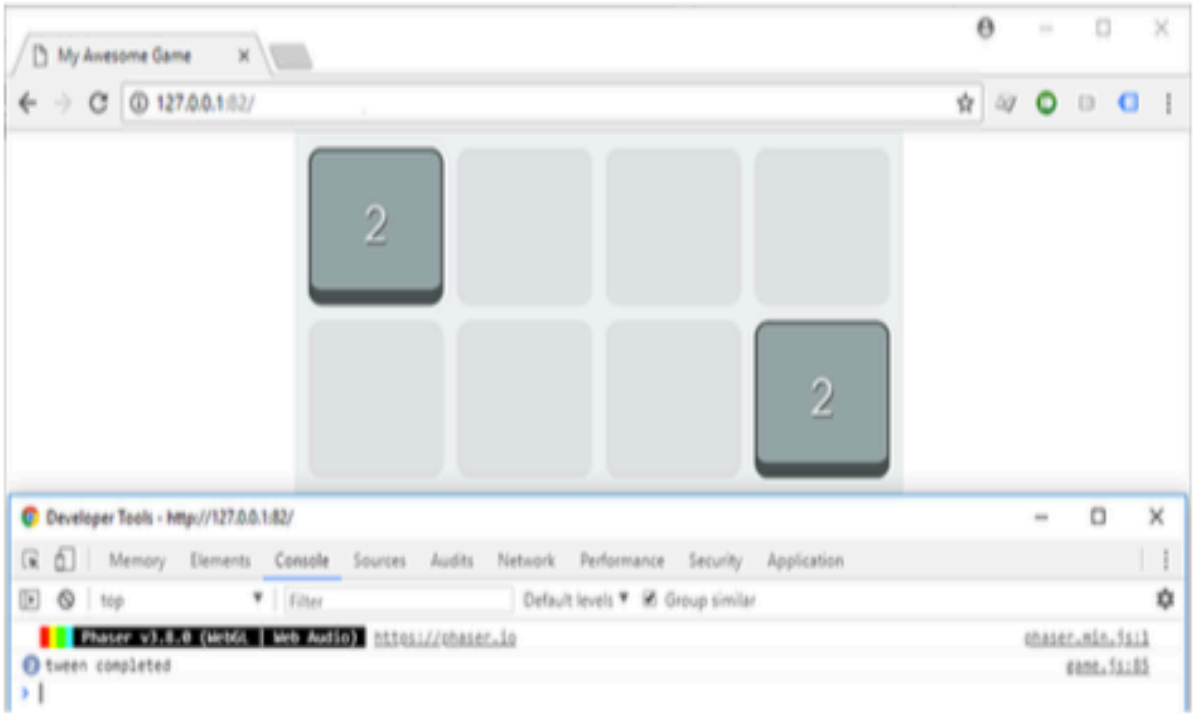
```
var gameOptions = {
    tileSize: 200,
    tileSpacing: 20,
    boardSize: {
        rows: 4,
        cols: 4
    },
    tweenSpeed: 2000
}
```

The animation will take two seconds to complete, which is way too much to just show a tile, but it's just to let you see what happens, and we are going to change it later. Then we need to add some lines to `addTile` method to create the tween:

```
addTile(){
    // same as before
    if(emptyTiles.length > 0){
        var chosenTile = Phaser.Utils.Array.GetRandom(emptyTiles);
        this.boardArray[chosenTile.row][chosenTile.col].tileValue = 1;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.visible = true;
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.setFrame(0);
        this.boardArray[chosenTile.row][chosenTile.col].tileSprite.alpha = 0;
        this.tweens.add({
            targets: [this.boardArray[chosenTile.row][chosenTile.col].tileSprite],
            alpha: 1,
            duration: gameOptions.tweenSpeed
        });
    }
}
```

Demo(1) - "Checking when tween end"

Now run the game and when tiles are completely visible you should see "tween completed" appear in the console.



What happened? We added two options to the tween, to add a callback function which prompts some text to the console and sets `canMove` property to `true`, and to set the callback scope to `this`.

> `onComplete` function is executed once the tween is completed.
>
> `callbackScope` sets the scope of `onComplete` callback function.

Now we can know when a tween finishes and act consequently.

Demo2 - Waiting for player input (Both key presses & mouse swipes).

Note: there are 4 directions for mouse swipes:
Up, down, left & right… notice the console for the time (milliseconds) + coordinates.

Waiting for player input

There's no game without player input, so here we go straight to the point: being a cross platform game we will allow players to interact with the game using keyboard, mouse or fingers.

We'll add two lines to `create` method of `playGame` class:

```
create(){
```

```
    // same as before
    this.input.keyboard.on("keydown", this.handleKey, this);
    this.input.on("pointerup", this.handleSwipe, this);
}
```

These two lines will listen respectively for the player to press a key, calling `handleKey` method which we are about to write, and to release the pointer – no matter if mouse pointer or the finger – calling `handleSwipe` method, which we are also about to write.

From these two lines you can see two different ways of handling inputs.

Talking about keyboard, we define a keyboard input the action of pressing down a key, so that the callback function will be execute as soon as the player presses a key, and not when the key is released.

Talking about pointers, the pointer input calls the callback function once the player releases it, because we want the game to be controlled by swiping, and a swipe ends when the player raises the finger or releases the mouse button.

> `input.keyboard.on("keydown", callback, context)` executes `callback` function in `context` scope when a keyboard key is pressed.
>
> `input.on("pointerup", callback, context)` executes `callback` function in `context` scope when a pointer – mouse pointer or finger – is released.

Let's have a look at handleKey method. At the moment we just need to know which key has been pressed down.

The game will be controlled with arrow keys, but we are going to add this feature later.

```
handleKey(e){
    var keyPressed = e.code
    console.log("You pressed key #" + keyPressed);
}
```

Like most callback functions, `handleKey` accepts the event itself as argument, so it's easy to save the code value of the key we just pressed in `keyPressed` variable.

> `code` property of a keyboard event returns the code of the key which fired the event.

Run the game, press some keys and this is what you should see in the console:

```
You pressed key #KeyR
You pressed key #KeyG
You pressed key #KeyC
You pressed key #KeyV
You pressed key #KeyS
You pressed key #KeyH
```

And this is is `handleSwipe` method:

```
handleSwipe(e){
    var swipeTime = e.upTime - e.downTime;
    var swipe = new Phaser.Geom.Point(e.upX - e.downX, e.upY - e.downY);
    console.log("Movement time:" + swipeTime + " ms");
    console.log("Horizontal distance: " + swipe.x + " pixels");
    console.log("Vertical distance: " + swipe.y + " pixels");
}
```

It's a bit more complicated because it's not just a matter of seeing which key has

been pressed: to check for a swipe we need to know how much time passed since the player started the input – by clicking the mouse or touching the screen – and the distance traveled during such time.

As usual we can find this information in some event properties.

> `downTime` property of `pointerup` event returns the timestamp taken when the input started, in milliseconds.
>
> `upTime` property of `pointerup` event returns the timestamp taken when the input ended, in milliseconds.
>
> `downX` and `downY` properties of `pointerup` event return respectively the horizontal and vertical coordinates where the input started, in pixels.
>
> `upX` and `upY` properties of `pointerup` event return respectively the horizontal and vertical coordinates where the input ended, in pixels.

At the end of the script, `swipeTime` variable contains the duration of the swipe, in milliseconds, while `swipe` variable contains a `Point` object representing the distance traveled, in pixels.

Launch the game and drag here and there with the mouse, you should see something like this:

```
Movement time:333.6199999998789 ms
```

```
Horizontal distance: 455.18324607329845 pixels
Vertical distance: -3.7696335078534275 pixels
Movement time:717.3829999999143 ms
Horizontal distance: 510.7853403141362 pixels
Vertical distance: 358.1151832460733 pixels
Movement time:583.9269999999087 ms
Horizontal distance: 619.1623036649214 pixels
Vertical distance: -261.98952879581145 pixels
>
```

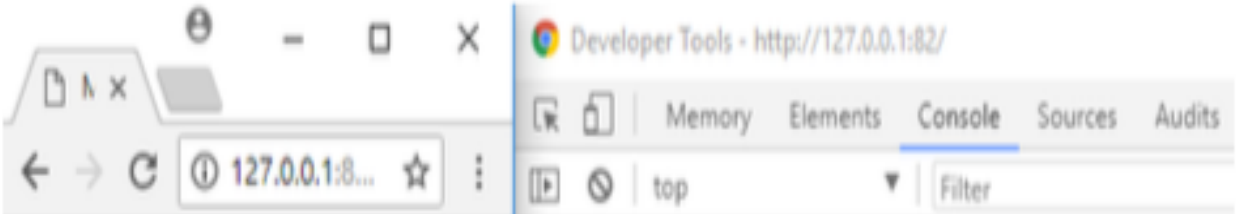
We have all required data to check if the player swiped, but there's another concept to cover about input events.

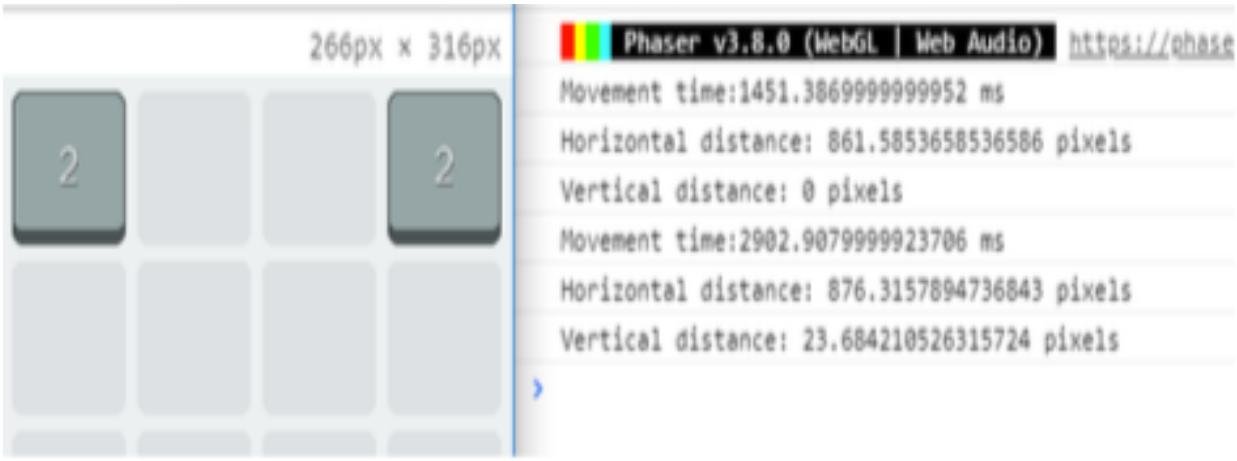
Understanding how pixels scale

We started talking about pixels, and how to measure distance in pixels, because pixels are a unit of measurement.

The problem is the game scales according to device resolution so different devices with different resolutions are supposed to have a different amount and density of pixels, and this should affect pixel measurements, right?

Wrong. Look at this picture:





We have a 266x316 pixels window, but if you move the pointer from side to side of you will always get a number around 900, which is the actual size of the game.

No matter the size of the canvas, you will be always working with game size.

This saves us from a lot of headache since we don't have to bear with screen resolutions or window sizes, we just have to set a game size and that's all.