Transitioning out of Scratch with WoofJS

Clarissa Littler

Updated as of March 16, 2019

1 What's WoofJS

WoofJS is a system for making small video games. It's meant for people who are bored with things like Scratch and are ready to start learning a general programming language, in this case JavaScript, rather than an educational language such as Scratch. That doesn't mean you shouldn't use it if you haven't used Scratch! It's still a great, simple, way to learn programming through making games.

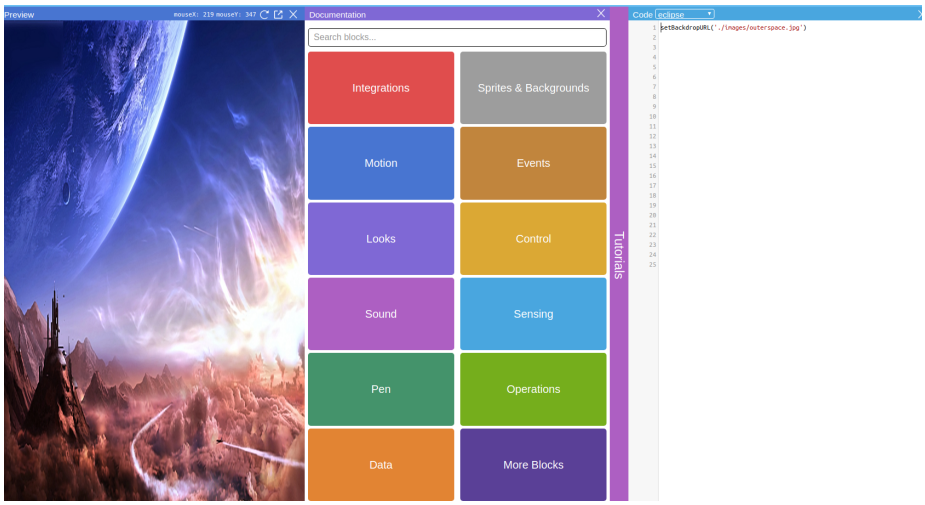
In this small tutorial we'll be giving a general introduction to WoofJS, explaining how it's different from Scratch, and then providing a walkthrough for a simple endless runner style game.

Feel free to follow as much or as little of this tutorial as you'd like!

I'm assuming *no* programming experience other than possibly Scratch. There's a lot of concepts we're packing in here and they make take time to absorb!

2 Introduction to WoofJS

The first thing you need to do is open `http://woofjs.com/create.html`. You should see something that looks like



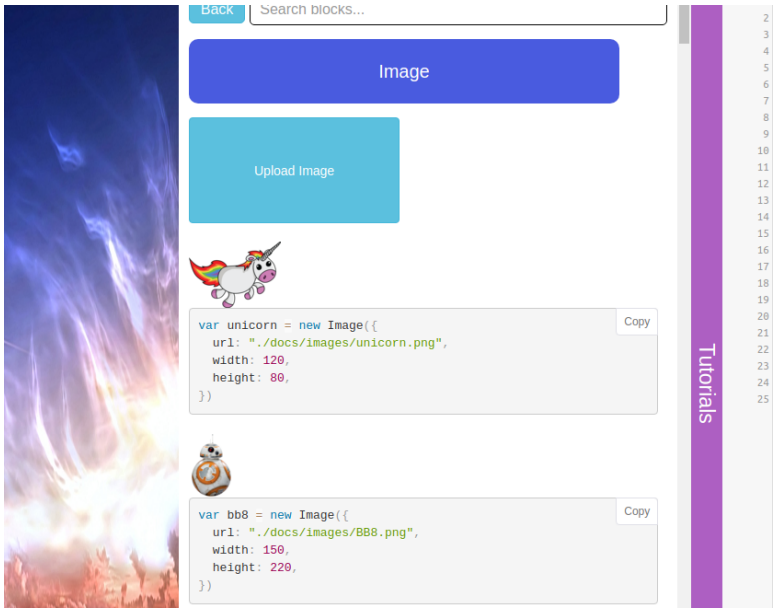
If you don't have an account you'll need to make one by clicking on the big button that says *Sign up* in the upper-left hand corner of the screen.

Now what? Let's talk about the three big sections you're looking at:

- The game is on the left hand side. This is where everything you code up can be seen and played. It's the stage from Scratch, basically!
- The right third of the screen is where you code. Unlike Scratch, there's only *one* area for code where the code for *all* sprites, *all* backgrounds, *everything* that happens in the game.
- The *middle* of the screen is the documentation. If you're familiar with Scratch it looks a lot like where you'd find the blocks to use, but you don't have to find things here in order to code your game. Rather, it's useful directions for *how* to make a game in WoofJS.

The documentation section has a really cool feature that we'll use to kick things off.

Click on the `Sprites and Backgrounds` section in the documentation then select the `Image` section. You should see something like



Now you can click on the button that says `copy` next to the weird-looking unicorn, then click on the code section just below the line that says

```
setBackdropURL('./images/outspace.jpg')
```

and press `Ctrl-V` to paste the code.

You should see the unicorn in the middle of your game and your code should look like

```
setBackdropURL('./images/outerspace.jpg')
var unicorn = new Image({
  url: "./docs/images/unicorn.png",
  width: 120,
  height: 80
})
```

Even here there's a few concepts we need to talk about!

First, that JavaScript is a language where everything proceeds *order* from the *top* of the code to the *bottom*. What does this code actually do, though?

The first line `setBackDropURL('./images/outerspace.jpg')` is your first example of a *function* in JavaScript. Functions are like custom blocks in Scratch.

They're like reusable chunks of code for easy use. Imagine if you saw a friend's shoe untied and you had to explain, in very painful detail, what to do to re-tie their laces every time this happened. That'd be terrible! No, instead you can say "dude, tie your shoe" and your friend *knows what that means and can do it.* In this case "it" is "make a background with the image provided between the quote marks". This is a built-in example, but you can provide your own links to images and it'll use those instead.

> Experiment to try: Do an image search and find an image you'd rather have as a background for your game. Get the link to the image and replace the part between quotes in the `setBackDropURL` function with the link you've found. If all went well, you should see the sci-fi landscape replaces with the image you chose!

The next thing to explain is the chunk of code that looks like

```
var unicorn = new Image({
    url: "./docs/images/unicorn.png",
    width: 120,
    height: 80
})
```

This is creating a new sprite from an image and then giving it a name, `unicorn`, for future reference. This time we're giving the function `Image` a piece of data that includes a bunch of different kinds of data separated by commas and enclosed in those curly-braces.

> Tip: If you're not familiar with where curly-braces are on the keyboard they're generally found by pressing `Shift` and the two keys to the right of the `p` key.

The word `var` that comes before `unicorn` is declaring that `unicorn` is going to be a new *variable*. Variables are far more common in JavaScript than they are in Scratch. You need them to not just hold data like how much time has elapsed or how much health your character has left but to even be *the names of the sprites*.

You can actually read what this data tells the image maker function: the image is found at the URL "./docs/images/unicorn.png", that the image should be 120 pixels wide and should be 80 pixels tall.

These aren't the only properties the image has. Like a Scratch sprite it has

a bunch of things you can change about it, like its x and y co-ordinates, the direction it's facing, or how transparent it is.

How do we access them, though? Add the following line of code just below where you made the unicorn.

```
unicorn.x = -100
```

You should see the weird little unicorn be further to the left than it was before. This is because you *set* the x-position of the unicorn to be -100.

What other properties can you modify? Here's an easy way to find out: type `unicorn`—or the name of any other variable—and then the `.`, the period on your keyboard, and then *stop typing*. You should see a little pop-up that shows you a bunch of possible properties that the sprite has. You can even navigate through this menu by pressing the up and down keys and hitting tab to fill in the code.

This isn't really interactive in any way, though, so in the next section we'll start making the unicorn respond to key-presses.

3 Starting your game: an endless runner

3.1 Movement

We're now ready to start writing our movement code! If this were Scratch, we'd drag out a forever block and put some if-blocks inside it. Since this is WoofJS, we're going to use a forever *function* and if-*statements* inside it.

We'll show the code and *then* explain what's happening. So go ahead and add the following to just below where you define the unicorn

```
forever(() => {
    if(keysDown.includes('UP')){
        unicorn.y = unicorn.y+10
    }
})
```

`forever` is a function that, well, runs "forever". Like a lot of the functions we've seen it takes some data—called an argument—between two parentheses. In this case, though, it takes *another function* as its argument. The part that reads as

```
() => {
    if(keysDown.includes('UP')){
        unicorn.y = unicorn.y+10
    }
}
```

is the function definition. You're making a function that takes *no arguments*, which is why it has nothing between its parentheses, and *then* it runs the code that comes to the right of the arrow. This code, if it has more than one line, has to be wrapped up in curly-braces.

What does this function *do*, though! Well, it uses an *if-statement* to check and see if the up-key is pressed. If it is, then it moves the unicorn up by 10 pixels.[1]

> Exercise: Only being able to move up is no good! Add in the code needed to move *down* as well.

> Extra credit: Make your sprite hang out at the left-edge of the screen by changing the line
>
> `unicorn.x = -100`
>
> to
>
> `unicorn.x = minX`
>
> `minX` is a variable defined by WoofJS to be the left-edge of the screen.

3.2 Making enemies

You can move up and down now, but there need to be enemies to dodge in our endless runner. If you're coming from Scratch you might be used to making a single enemy sprite and then cloning it repeatedly. That's not really how it works in WoofJS!

The equivalent technique is that we'll write a *function* that creates the enemy sprite and then call the enemy creation function repeatedly. We don't want to use `forever` this time, though, because that would make new enemies too

[1] All distances in WoofJS are implicitly in pixels

quickly. Instead, we'll use the `every` function which is like `forever` but only runs periodically, like "every second" or "every 0.1 seconds".

Copy the following code into your game

```
var enemies = []

var makeEnemy = () => {
    var e = new Image({
        url: "./docs/images/cupcake.png",
        width: 20,
        height: 20
    })
    e.x = maxX
    e.y = randomY()
    enemies.push(e)
}

every(0.2, "seconds", makeEnemy)

forever(() => {
    enemies.forEach(e => {
        e.x = e.x - 2
    })
})
```

what we're doing here is

- making a new variable called `enemies` and setting it equal to an *empty array*
  - arrays are like lists of things
  - this list of enemies is going to help us make sure that *every* enemy moves
- making our function to create our enemies
  - this function creates a new sprite image, puts it at the right-hand side of the screen, puts it at a random y position, then adds it to the `enemies` list
- creating a new `every` loop to make new enemies every 0.2 seconds
  - Extra credit: why do we put `makeEnemy` instead of `makeEnemy()` as the third argument to `every`
- creating a new `forever` loop that *for each* enemy in the `enemies` list, changes its x-position by -2

3.3 Making enemies *collide*

We still haven't added the ability for the enemy to collide with the player character, though. We really should!

First, we need to add an `hp` property to our player. Add the line

```
unicorn.hp = 10
```

just below where you defined the `unicorn` variable.

Now, we need to add some extra code into the movement for the enemies to check and see if they're touching the unicorn. Thankfully that's pretty easy! When it hits, we'll reduce the player's hp by 1, *remove* the enemy from the list so it's not moved anymore, then *delete* it to take it out of the game.

```
forever(() => {
    enemies.forEach(e => {
        e.x = ex. - 2 - (timer/10)
        if (e.touching(unicorn)) {
            unicorn.hp = unicorn.hp - 1
            enemies.remove(e)
            e.delete()
        }
    })
})
```

3.4 Adding a score and death

To add a score you'll need to

- Add a new property
- Add a new `every` function that ticks up every second to add 1 to the score

To add *death* you'll need to add a check to your player movement `forever` loop to see if your `hp` hits zero. If it does, call the `freeze` function to end the game.

```
// new property
unicorn.score = 0

// modified movement loop
```

```
forever(() => {
    if(keysDown.includes('UP')){
        unicorn.y = unicorn.y + 10
    }
    if(keysDown.includes('DOWN')){
        unicorn.y = unicorn.y - 10
    }
    //The new part!
    if(unicorn.hp <= 0) {
        freeze()
    }
})

every(1, 'second', () => {
    unicorn.score++
    timer++
})
```

and then adding the text for the score to the screen is pretty simple

```
var Sccore = new Text({
    text: () => unicorn.score,
    size: 30,
    x:maxX-20,
    y:maxY-20,
    color: "white"
})
```

> Extra credit: why is the "text" for the score a function? What happens if you take it out of the parentheses and arrow so that it just reads as `text: unicorn.score`?

3.5 Making it fairer: firing shots

This is going to be a lot like the way that we made our enemies move and collide. We'll

- Add a new `projectiles` variable that will be used to keep track of the things we shoot
- Add a function to make projectiles and add them to the list

- Add a new `forever` loop that checks to see if you've pressed the spacebar to fire a projectile
- Add a new `forever` loop that moves all the projectiles and checks if they've touched the enemy and, if it *has*, then delete the enemy, delete the projectile, and increase the score by 10

```
var projectiles = []
var makeProjectile = () => {
    var p = new Rectangle({
        width: 20,
        height: 10,
        ccoclor: "pink",
        x: unicorn.x,
        y: unicorn.y
    })
    projectiles.push(p)
}

var shotTimer = 0

forever(() => {
    if(keysDown.includes('SPACE') && shotTimer <= 0) {
        makeProjectile()
        shoot.startPlaying()
        shotTimer = 20
    }
    else {
        shotTimer = shotTimer - 1
    }
})

forever(() => {
    projectiles.forEach(p => {
        p.x = p.x + 5
        var touched = false
        enemies.forEach(e => {
            if(p.touching(e)) {
                enemies.remove(e)
                projectiles.remove(p)
                e.delete()
                touched = true
                unicorn.score = unicorn.score + 10
            }
        })
        if (touched) {
```

```
            p.delete()
        }
    })
})
```

> Extra credit: Why did we split up the forever loop for moving up and down versus the loop for firing shots? If you're not sure then try combining them and seeing what happens!

3.6 Music and Sound Effects

We haven't even touched sounds in our game! Read through the sounds documentation and add some sound effects for

- firing your laser
- getting hit
- blowing up an enemy

4 A second project: platforming in WoofJS

To Be Written...