Day 1 of KAPLAY Workshop - Learning the Basics of JavaScript

- Based off of JSLegend's great tutorial video
- Using kaplay.js and JavaScript to build the game
- Will use Kaplay Playground as our IDE (since it requires no downloads, which works for the library's computers that can't download external software)
- Don't use semicolons in the code! It's one less thing to explain
- Have a whiteboard with definitions/syntax on it

What We'll Be Learning Today

1. Basics of programming: variables, math, strings, if-statements, etc.
2. Beginnings of kaplay (moving player character around)

Set the Ground Rules

1. There are no stupid questions
2. Raise your hand at any point to interrupt (doesn't matter if I'm talking)
3. Don't be afraid to ask the person next to you!
    1. Just be mindful of how loud you're talking
4. Be respectful towards others (just don't be mean)
    1. If someone doesn't understand something, try to explain it to them nicely or ask me for help

The purpose of this class is to introduce you to programming concepts. Not everything will stick right away, and that's okay! Follow along as best you can, just type out the code I type on my screen and you'll make it through.

Introductions

> Ask everybody what their name is and if they have programmed before. If so, what have they made?

I'll start by saying something like:

> My name is Mr. Josh and I started programming when I was 14/15. I had tried to learn to code before, but I didn't start understanding until high school by building games like the one

> we'll build.

Go in order.

What is Programming? How do Computers Work?

> Ask the class: what is programming?

At a basic level, programming is about giving the computer instructions to perform a task. It's similar to how you might tell someone to do something for you, but there's a key difference. To demonstrate this, I want you all to tell me how to close the door to this room.

> If at any point I can't close the door, I'll come back to where I'm standing. They have to start over.

Some example instructions:

1. Turn to face towards the door
2. Walk in that direction until you are in front of the door

This is the difference between you and a computer: you implicitly understood the steps needed to complete the task without being told, whereas a computer needs to be explicitly told what to do. It's your job as the programmer to break down the task into steps the computer can complete. Remember: a computer will do exactly what you tell it to do, and nothing more. This is one of the hardest parts to understand about programming, and is something I still get tripped up on to this day. However, this is also the fun part.

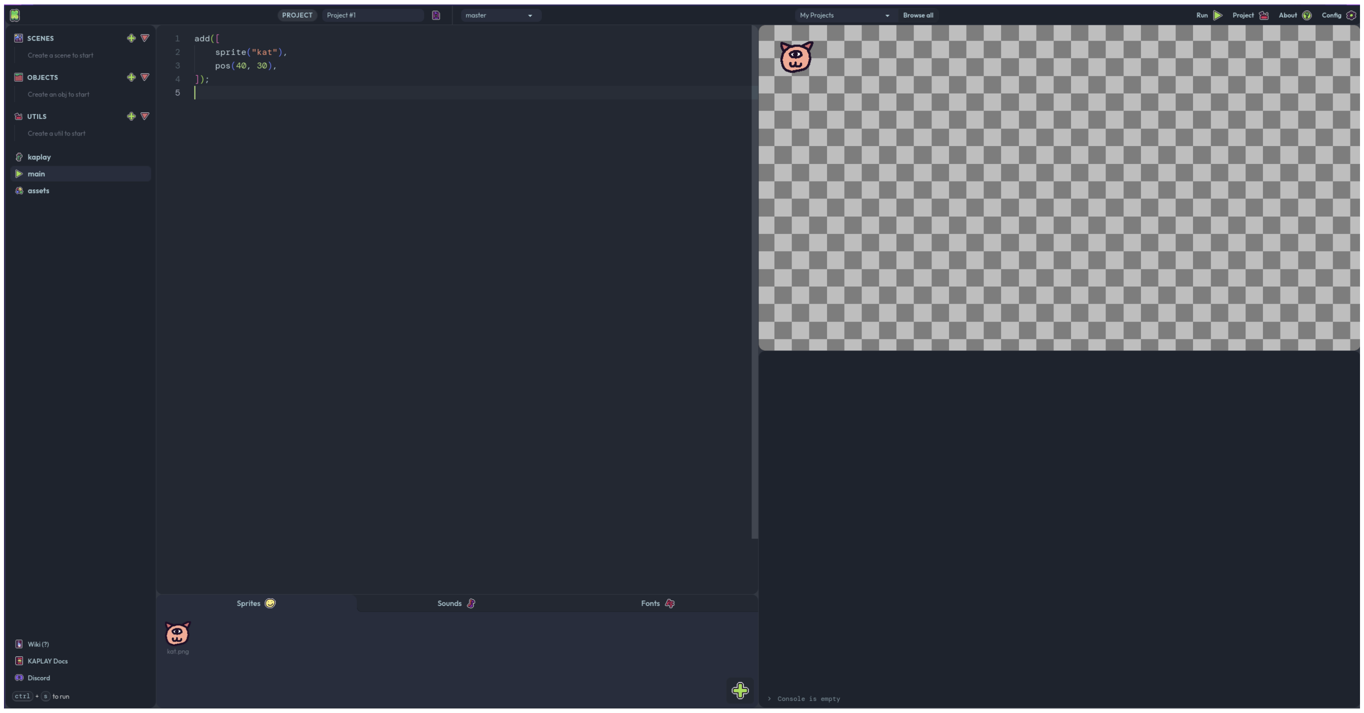
Let's get started.

Opening Kaplayground

This is where we'll doing all of our work. Open the following URL in your browser:

```
https://play.kaplayjs.com/
```

When you open it, you should see this:



```
add([
    sprite("kat"),
    pos(40, 30),
]);
```

Going around the screen:

- The large section on the left is our code. This is where we'll be working in for the most part.
- In the top right is our game. It currently doesn't do anything now because we haven't programmed any behavior yet. Right now, we'll ignore this, but will use this more later today and next week.
- In the bottom right is the console. Any text we print out will show up here. This is where we'll spend most of our time today.
  If the console isn't big enough, expand it to be bigger so you can see multiple lines.

From here on out, I want you all to follow along by typing the same code I type and running it. I'll show you how right now.

Setting up Kaplayground

1. Set up the wifi
2. Open the browser (Chrome? Edge?)
3. Hit the "Save project" in the top part of the screen to enable auto-save

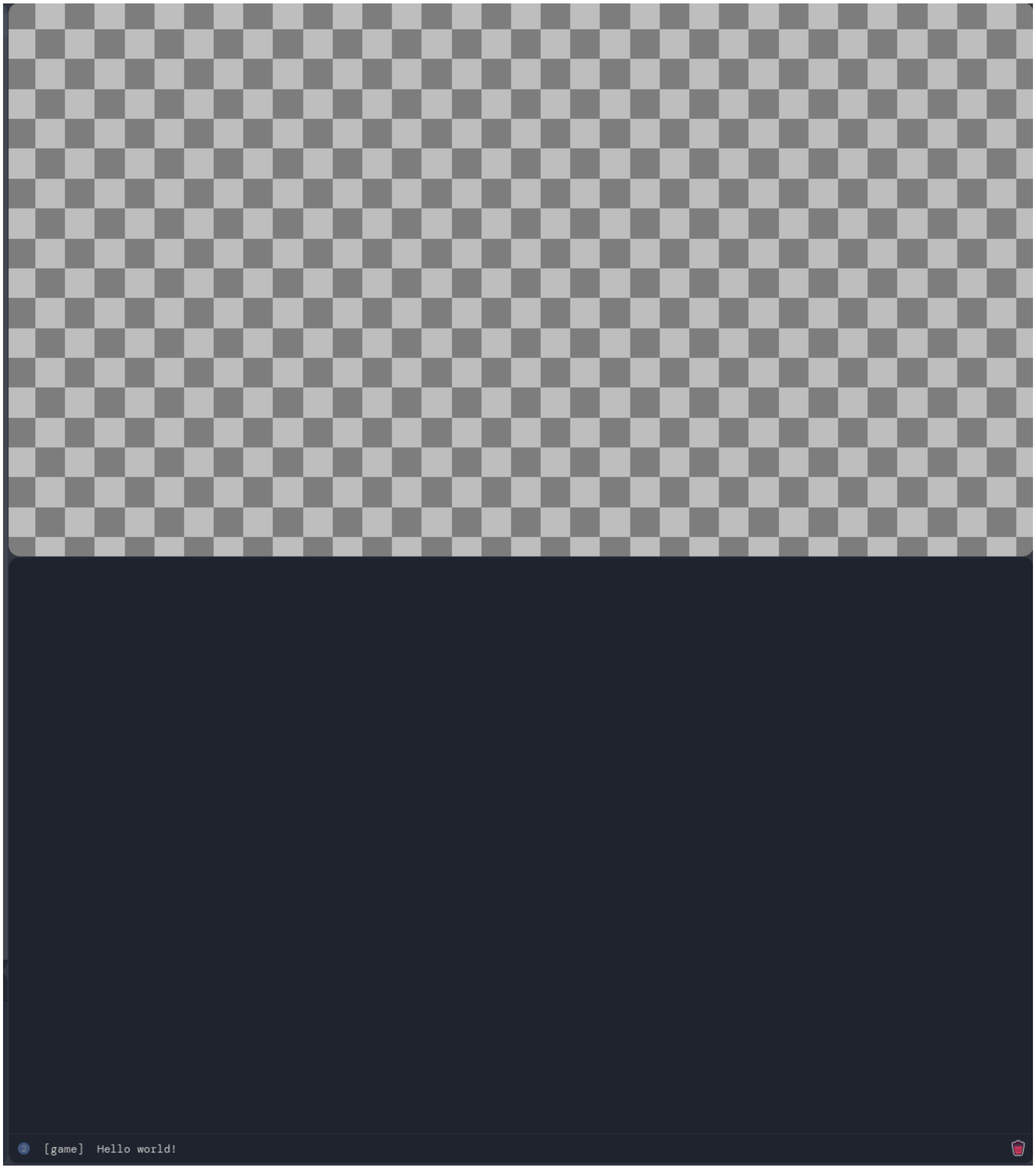
Hello World

In the code editor, select everything and delete it. Then, replace it with:

```
console.log("Hello world!")
```

When you save it (clicking the "Run" button in the top right or by using the `Ctrl + S` keyboard shortcut), you should see two things:

1. The cat image in the game area is gone
2. The text "Hello world!" appeared in the console



```
[game]  Hello world!
```

Congratulations, you wrote your first line of code! Let's break it down:

1. `console.log` is a function that we can call to show text in the console
2. The `(` and `)` indicate that we are calling the function
3. The item in between are the "parameters" we are passing to the function
4. `"Hello world!"` is a string of text
5. Putting `"Hello world!"` between the `(` and `)` means that we want to print that text to the console

We will talk more about functions later, but all you need to know for now is that they run other code without you seeing it.

Doing Math

You can also use JavaScript to do math for you:

```
console.log(2 + 2)
console.log(2 - 2)
console.log(2 * 2)
console.log(2 / 2)
```

You should see:

```
4
0
4
1
```

Additionally, you can "add" strings together too:

```
console.log("Hello " + "world" + "!")
```

> Ask them: what does this print out?

```
Hello world!
```

This process is called "concatenation" or "combining" of strings.

Understanding Execution Order

In the example above, notice how we had code on multiple lines. When we ran them, the results appeared in the order they appeared in. This is because the computer reads and runs code from top to bottom.

Variables

Let's say I want to print the same text multiple times:

```
console.log("Hello world!")
console.log("Hello world!")
console.log("Hello world!")
```

This works, but what if you want to change it? You'd have to change it in each place! Luckily, there's something we can use to help us: variables. Variables tell the computer that a certain name should be equal to a certain value. Think of them like a box to put something in. You label the box so you can find it later.

Let's modify our code above to see what happens:

```
let message = "Hello world!"
console.log(message)
console.log(message)
console.log(message)
```

In JavaScript, you define a variable with the `let` keyword followed by a name for the variable. Once you define a variable, you can use it. However, can you use it before you define it? Let's try:

```
console.log(message)
let message = "Hello world!"
console.log(message)
console.log(message)
console.log(message)
```

Notice how nothing is printed to the console. That's because it failed to run. Kaplayground won't show it, but it will fail with some text like:

> Uncaught ReferenceError: can't access lexical declaration 'message' before initialization

Remember, the computer reads code from top to bottom. If you haven't told it the name of a variable, how could it know to use it?

Changing a Variable

Variables are useful for reusing a value, but they can also be changed (that's why they're called *variables*). Let's see that in action:

```
let myUpdatedVariable = "first"
console.log(myUpdatedVariable)
myUpdatedVariable = "second"
console.log(myUpdatedVariable)
```

This should produce the output:

```
first
second
```

Even though we passed the same *name* to `console.log`, each line produces different output. That's because they're pulling from the same box.

Also, notice how I capitalized the first letters of "updated" and "variable" in the variable name. You'll see this often when variables contain multiple words. We can't have spaces between each word, so we use capitalization to signify the start of a new word. This is called "camel casing" since the capitalized letter looks like a camel's hump.

> For each of the examples, have some code on the screen already.

Exercise 1: Math

First, write a program with three variables that contain a number (you can pick the numbers): `a`, `b`, and `c`. Then, write a program that prints out:

1. The sum of each
2. The difference of each (in alphabetical order)
3. The product of each
4. The quotient of each

Sample code:

```
let a = 1
let b = 2
let c = 3

console.log(a + b + c)
console.log(a - b - c)
console.log(a * b * c)
console.log(a / b / c)
```

If you chose `a = 1`, `b = 2`, and `c = 3`, your output would be:

```
6
-4
6
0.16666666666666
```

Try playing around with it: what happens when you set `c = 0`?

> The last number will be `Infinity` because you can't divide by zero.

Exercise 2: Incrementing a Value

> Have them guide me in writing this solution. This isn't something I've taught them, but I want to see the gears turning.

Write a program that:

- Creates a variable `x` set to some number (you can pick)
- Prints out `x`
- Add 1 to `x`
- Prints out `x` again

If you chose 0 as the starting value of `x`, it should print:

```
0
1
```

If you chose 10 as the starting value of `x`, it should print the following *without changing the code*, only the value of `x`:

```
10
11
```

How would you do that?

Sample code:

```
let x = 10
console.log(x)
x = x + 1
console.log(x)
```

Conditionals and Booleans

So far, we've done some math, concatenated some strings, and printed out their results. This is great, but all we have done is run code from the top to bottom. Sometimes, we need computers to only run some code when some condition is met. This is where if-statements come in.

Let's say we want to write a program that checks whether you are in front of the door or not. If you are, you open it. Otherwise, nothing happens. Let's show the code for that:

```
let isInFrontOfDoor = true
if (isInFrontOfDoor) {
    console.log("Open the door!")
}
```

> Talk about camel case for variables.

Notice the value `true` there. That's a **boolean**. A boolean is one of two values:

- `true`
- `false`
  When `true` is passed to an `if` statement, it runs the code inside the curly braces. If it's `false`, the computer skips over that code.

Now, try changing `isInFrontOfDoor` to `false` instead. What happens? Nothing, because it skips over the code in the curly braces.

`else` Statements

What if we want to do something if the condition is false, though? Let's try adding that too:

```
let isInFrontOfDoor = true
if (isInFrontOfDoor) {
    console.log("Open the door!")
} else {
    console.log("Keep walking")
}
```

The `else` statement lets us do something when the above `if` statement's condition isn't true.

Comparisons

Let's modify the code that we just worked on. Instead of storing whether we're in front of the door or not, let's store how many steps away from the door we are. First, we'll create our variable:

```
let stepsAwayFromDoor = 50
```

If you are 0 steps away from the door, you are right in front of it. Otherwise, you still need to keep going. Let's write some code that checks that:

```
let stepsAwayFromDoor = 50

if (stepsAwayFromDoor === 0) {
    console.log("Open the door!")
} else {
    console.log("Keep walking")
}
```

The computer compares `stepsAwayFromDoor` with `0`. If `stepsAwayFromDoor` is equal to 0, then that expression "evaluates" to `true`. Otherwise, it's false and runs the `else` block. The `===` symbol is a "comparison operator" and there are many of them:

- Less than/less than and equal to: `>` and `>=`
- Greater than/greater than and equal to: `<` and `<=`
- Equals: `===`
- Not equals: `!==`

> Draw out the equivalent symbols they'd see in their math classes and explain why we don't use those in programming.

Try changing the `===` in the above code example to the different operations and see how it changes the behavior.

`else if` statements

Let's modify the code above to say different things if you are really far away:

- If you are over 100 steps away
- If you are between 100 and 10 steps away

- If you are

We could add an `else if` statement into the mix. Let's see that:

```
let age = 26

if (age > 18) {
    console.log("Older than 18")
} else if (age === 18) {
    console.log("Exactly 18")
} else {
    console.log("Younger than 18")
}
```

Try changing age again to see what gets printed. First, we check if you're older than 18. If you aren't, then we check if you're 18 exactly. Finally, if you are younger than 18, we run the `else` block.

Getting Started with Kaplay

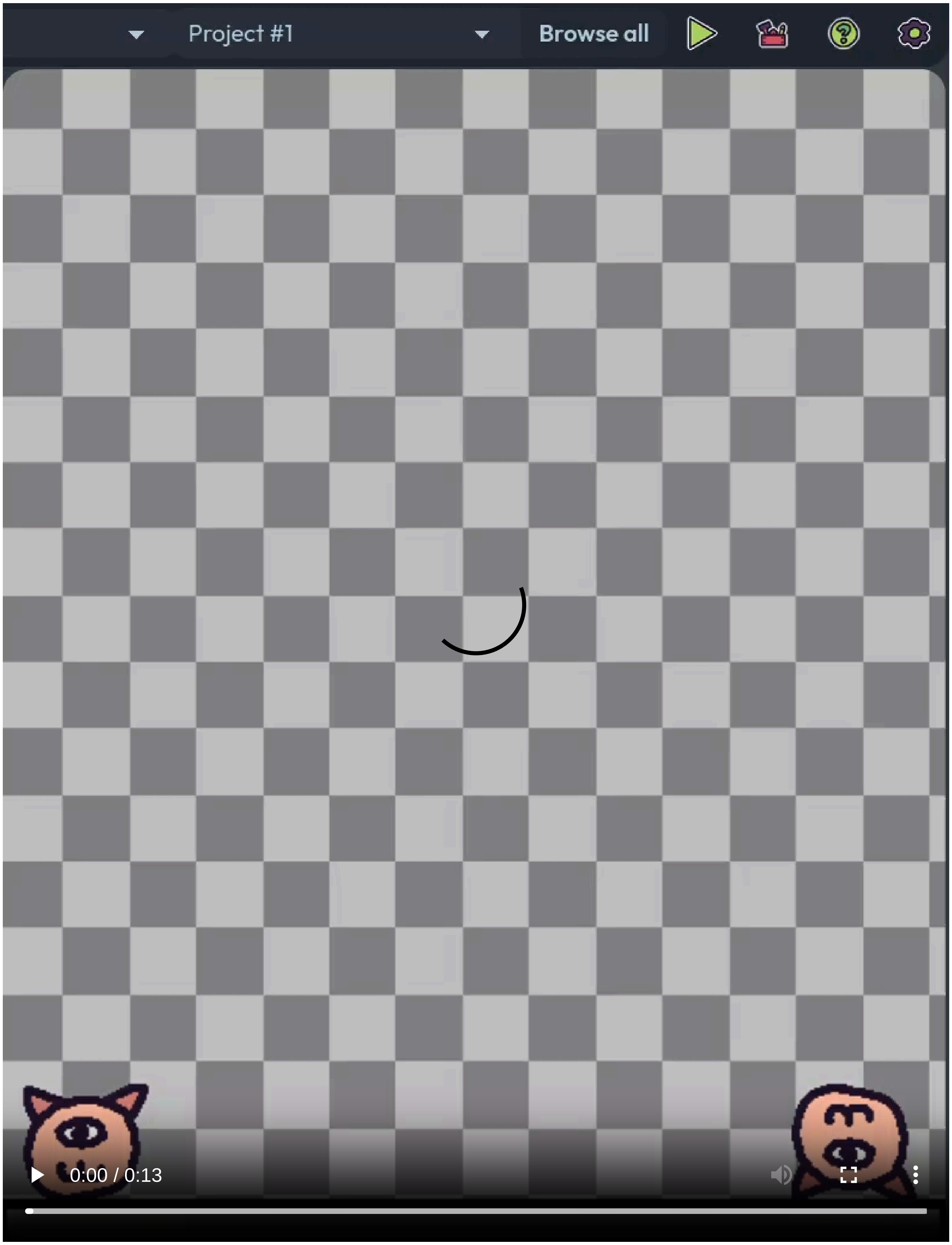
There's more to coding and making a game than we talked about so far. However, going over everything in the format we have is boring. Let's start making a game! We won't make a full game today, but we'll lay the foundation for a small one.

Full Code

> Paste the code below to show them what we'll make, but remove everything to start from scratch.

```
let kat = add([
    sprite("kat"),
    pos(10, center().y),
    area(),
    body(),
    "player"
])

setGravity(1000)

let platform = add([
    rect(width(), height()),
    pos(0, height() - 20),
    area(),
    body({ isStatic: true }),
    outline(5),
])

kat.onKeyPress("space", function () {
    if (kat.isGrounded()) {
        kat.jump()
    }
})

kat.onKeyDown("left", function () {
    kat.move(-250, 0)
})

kat.onKeyDown("right", function () {
    kat.move(250, 0)
})

let goal = add([
    sprite("kat", { flipY: true }),
    area(),
    body(),
    pos(width() - 75, height() - 100)
])

goal.onCollide("player", function () {
    let bubble = add([
        anchor("center"),
        pos(center()),
        rect(400, 100, { radius: 8 }),
        outline(4, BLACK),
    ])
    bubble.add([
        anchor("center"),
        text("You did it!", {
            size: 26,
        }),
        color(BLACK),
    ])
})
```

Here's a video of the game in action:



We can:

- Move using the left and right arrow keys

- Jump using the spacebar
- Collide with the upside down "kat" to win the game

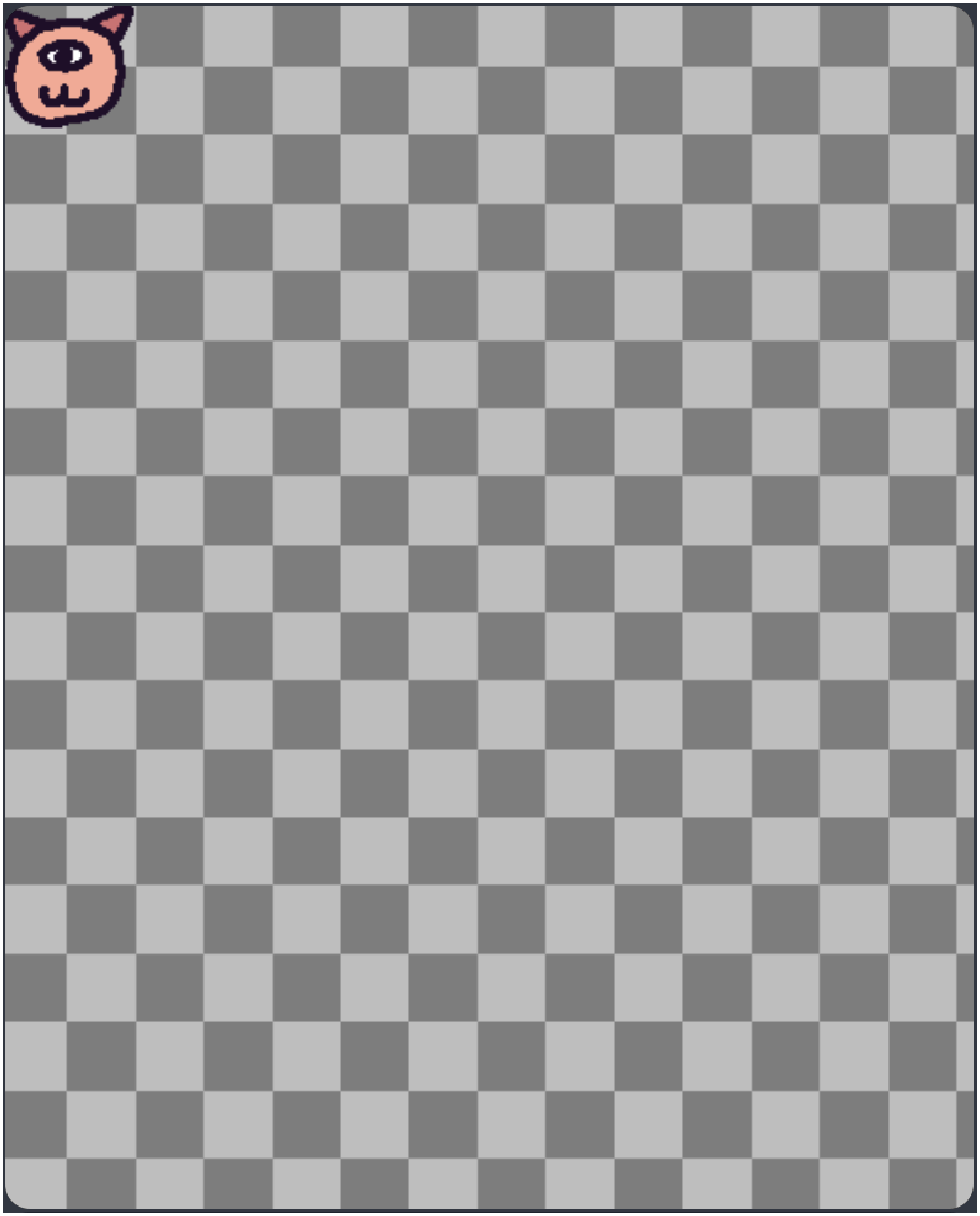
This is a simple game, but getting to this point is more complicated than you might think. I'll walk you through how to get here. This will give us a good foundation to build on top of next week.

Draw a Sprite to the Screen

Let's start from a clean slate and remove all of the code we've written so far. First, let's add the cat back to the screen:

```
let kat = add([
    sprite("kat")
])
```

We should see the cat (or "kat") on the screen from before:



Let's break this code down:

- `add` creates a "game object" and puts it into our game
- `add` is a function (like `console.log`) and takes a **list** as a **parameter**

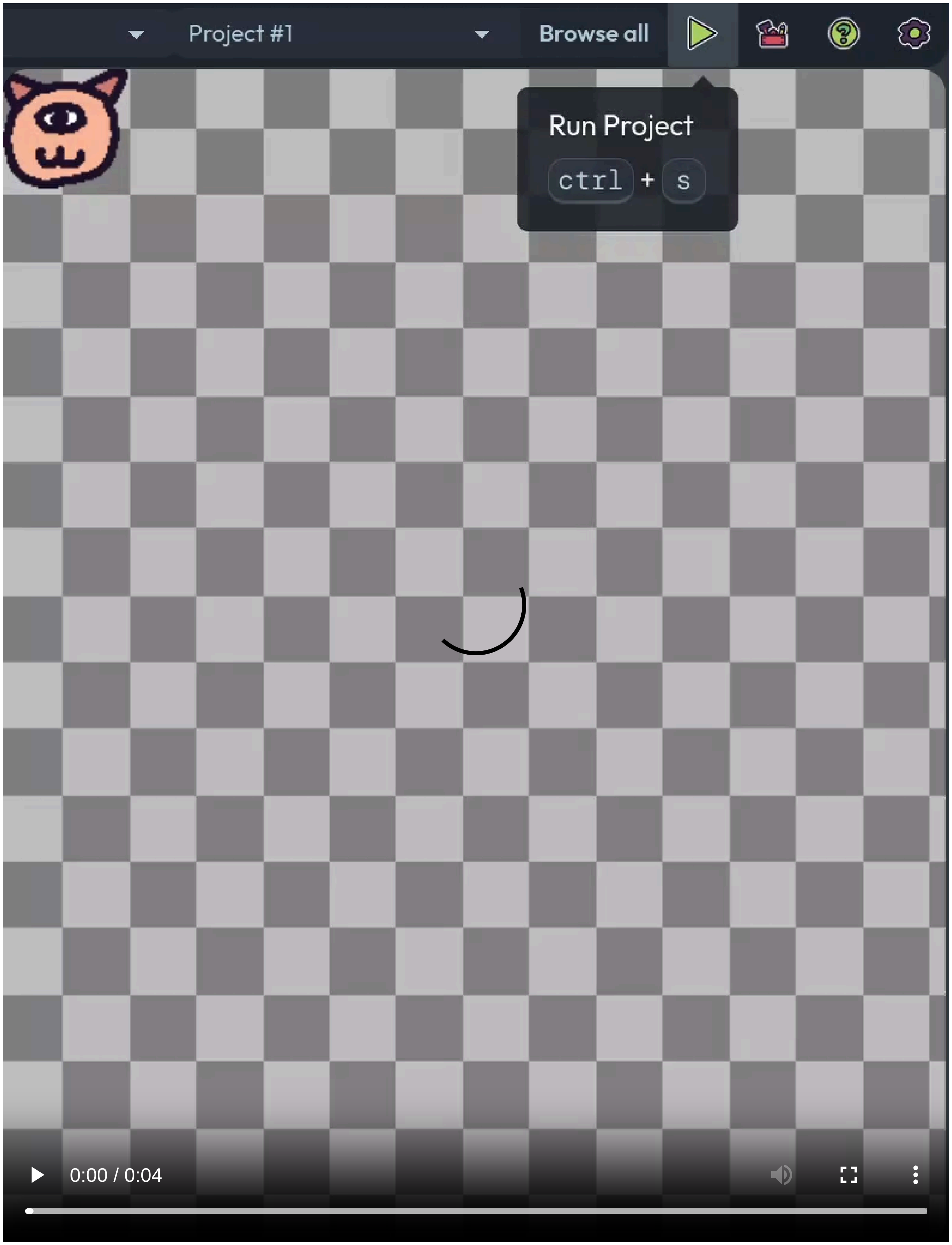
    - This list is defined by the square brackets (`[]`)
    - The list contains all of the **attributes** of our game objects
- We tell it that we want the game object to have a 2D image called a "**sprite**"
- We pass a string to `sprite` telling it which image we want to load
    - If you see in the bottom left of your screen, the only image we have is the "kat", so we'll use that one
    - You can configure others in a real game, but the "kat" will get the job done

Adding Gravity

Next, let's add some gravity. This is a platformer, so the player needs to be able to move and jump around. Remember how the computer doesn't do anything unless we tell it to? Let's tell it to add some gravity:

```
let kat = add([
    sprite("kat")
])

setGravity(1000)
```

Don't worry too much about the `1000` number, it's just a number I chose that felt "realistic" enough. Let's run our game and see what happened:



Nothing happens! Why???

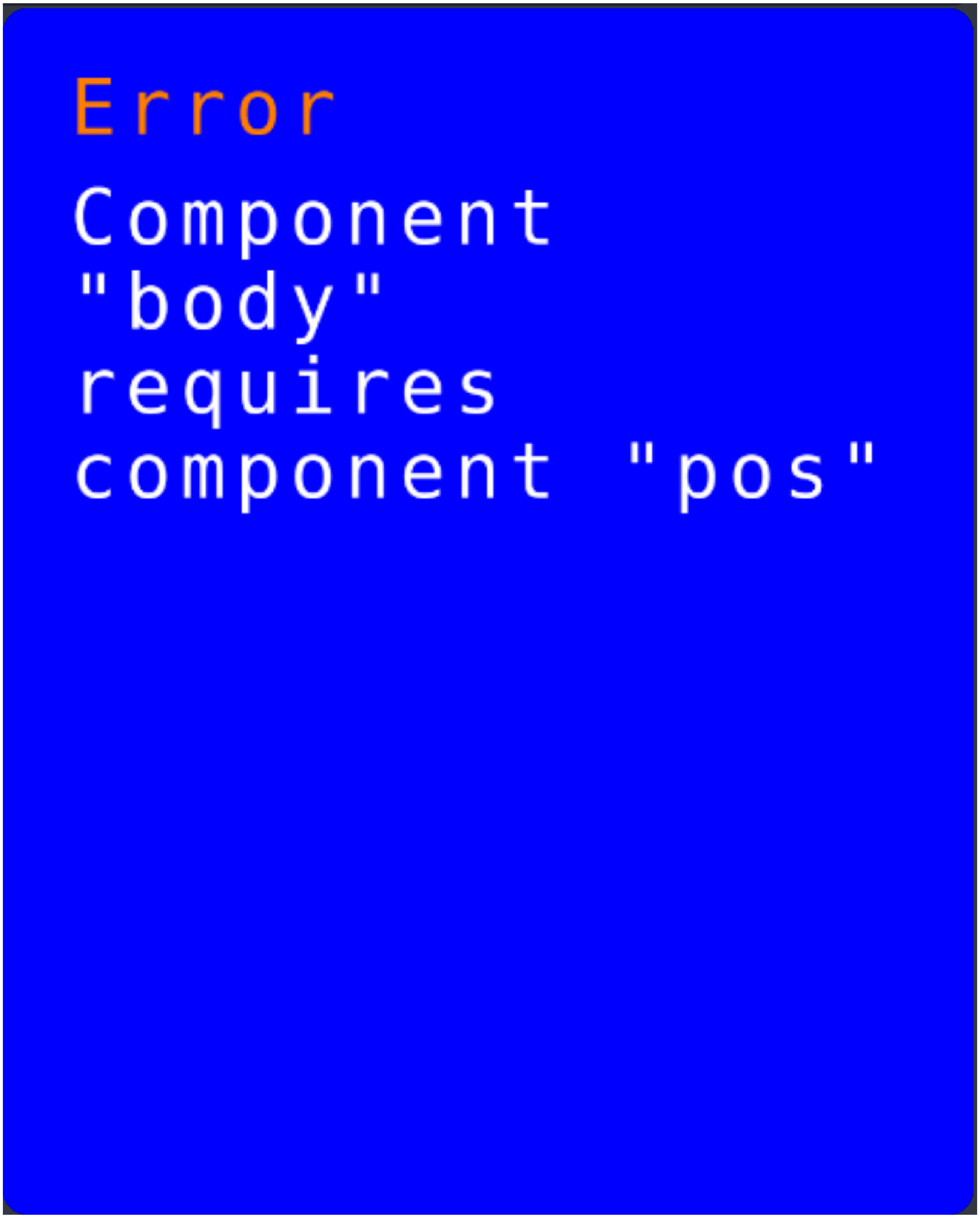
Remember the full game: when your character came into contact with the upside down "kat", some text appeared in the game. Gravity doesn't apply to that rectangle and text. By default, KAPLAY doesn't apply gravity to objects, so we need to *tell* it to do that. That's where the `body` function comes in:

```
let kat = add([
    sprite("kat"),
    body()
])

setGravity(1000)
```

A couple things to note:

1. After `sprite("kat")` we add a new "property" to the list called `body()`
2. `body()` is a function that tells KAPLAY that this object must have gravity applied to it
3. We add a comma between each "list item" in the list

Now, let's see what happens:



```
Error
Component
"body"
requires
component "pos"
```

Uh oh, it's not working. Remember the "close the door" example? Remember when you didn't tell me to remove the door stopper and the door wouldn't close? This is one of those moments.

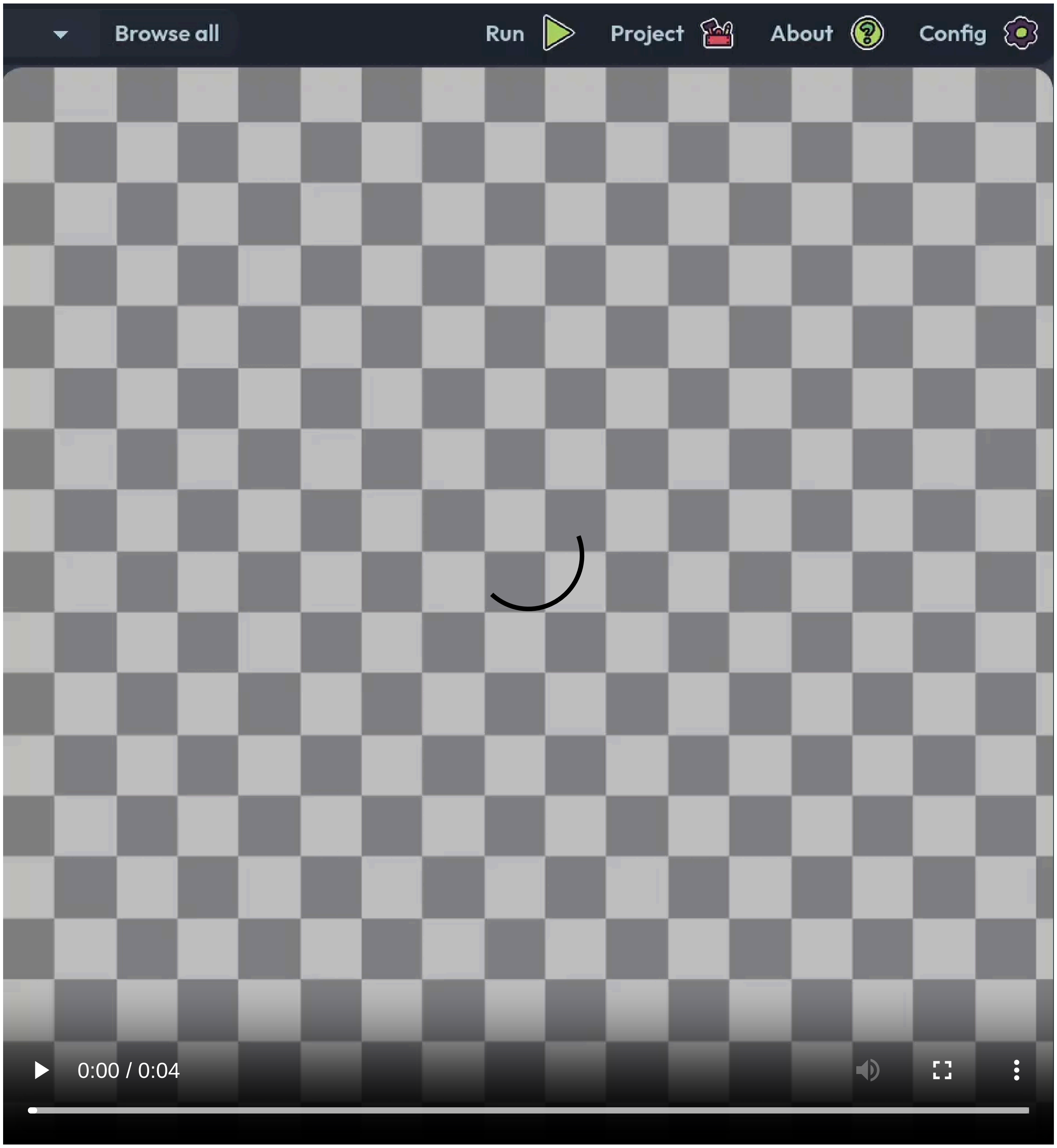
Just like in the real world, we get feedback on how to fix it. To enable physics, we need to tell the computer where to put the player on the screen. In other words, it's *position*:

```
let kat = add([
    sprite("kat"),
    body(),
    pos(center())
])

setGravity(1000)
```

Let's break this down:

1. I added another list item (which means another comma)
2. `pos`, short for position, tells it where to put the object on the screen and lets us move it around
3. `center()` is a function in KAPLAY that returns the position of the center of the screen

Let's see what happens now:



Perfect! Gravity gets applied to the player and they start to fall. We're making progress, but we don't want the player to fall forever. Since this is a platformer, let's add a platform for the player to stand on:

```
let kat = add([
    sprite("kat"),
    body(),
    pos(center())
])
```

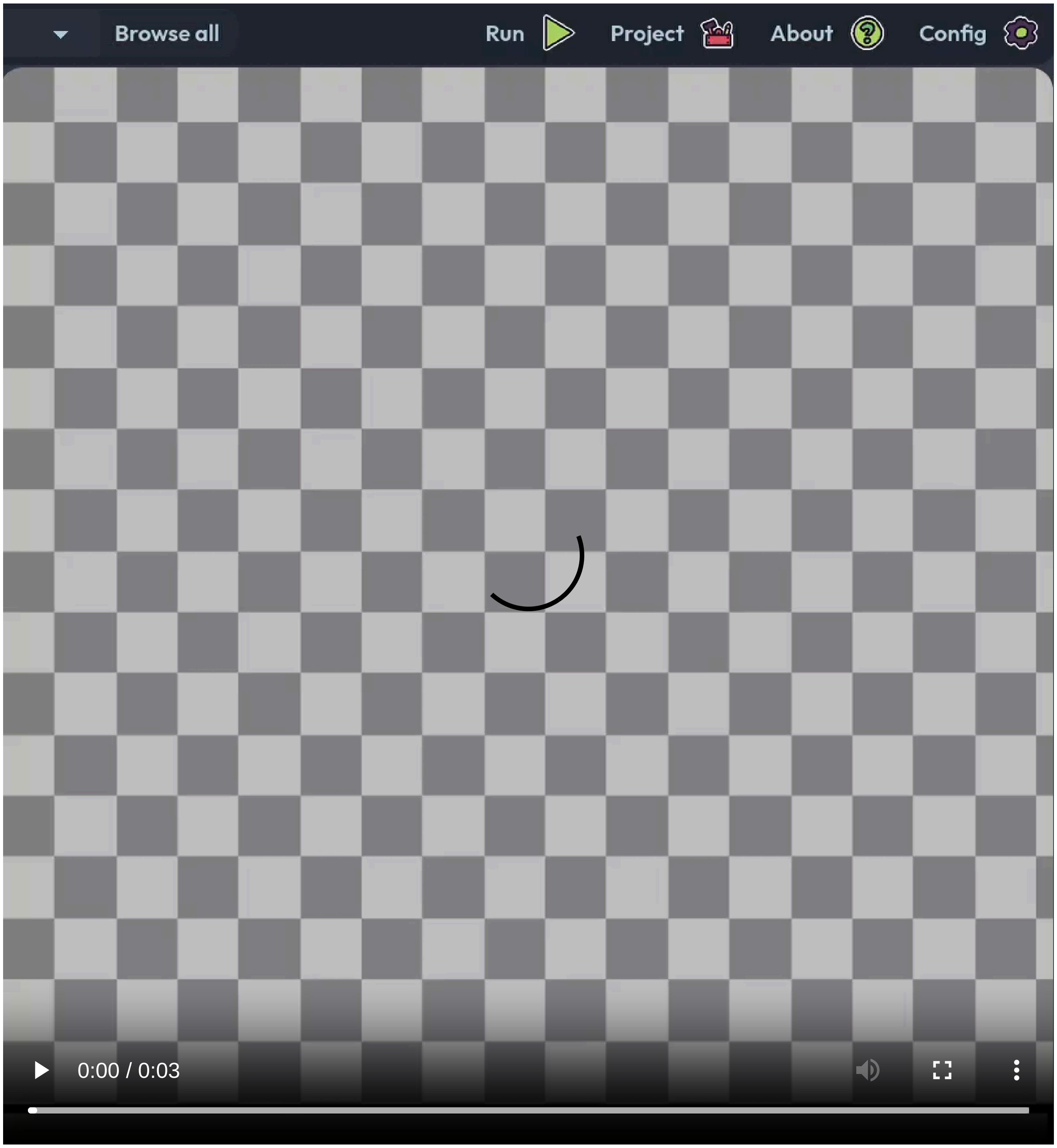
```
setGravity(1000)

// new code below

let platform = add([
    rect(width(), 15),
    pos(0, height() - 20),
    outline(5),
])
```

Let's break this down:

- The platform is another "game object"
- Instead of using a sprite, we tell KAPLAY to create a basic rectangle shape
    - `width()` is a function that gives the width of the game screen
    - `height()` does the same thing
    - For simplicity, the platform should take up the full width of the screen and be 15 pixels tall
    - We want it to appear at the bottom of the screen, so we use the coordinates `0` and `height() - 15`

> We'll talk more about these numbers next week. For now, just know that these numbers put the platform on the screen where we want it to go.

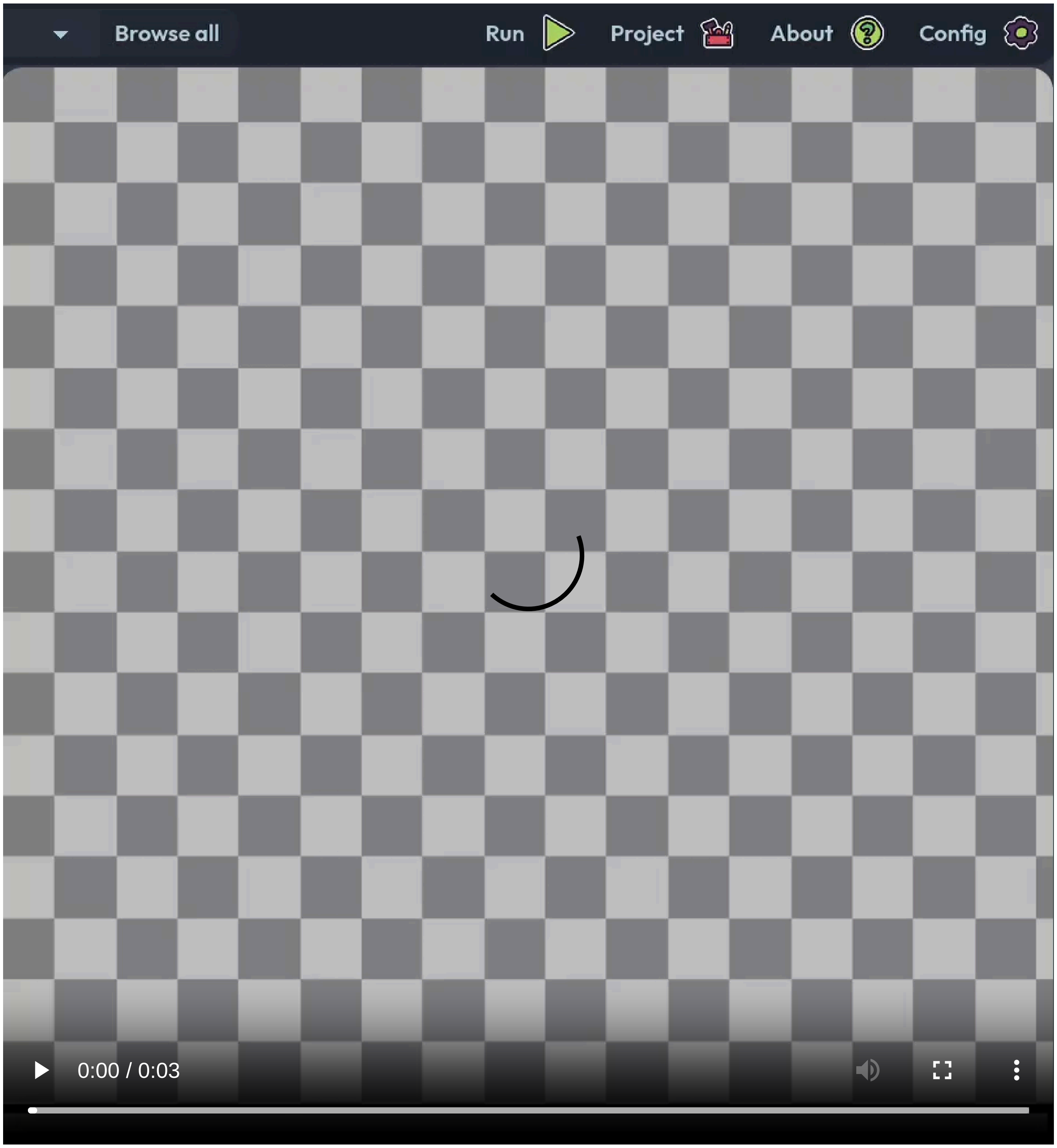
This creates a white platform with a black border at the bottom of the screen. The rectangle is the same width as the screen. We see it on the screen, but the player just falls through it:



Remember how we had to add the `body()` function to the player's game object? Maybe we need to do that here so it takes part in physics? Let's try that:

```
let kat = add([
    sprite("kat"),
    body(),
    pos(center())
])

setGravity(1000)
```

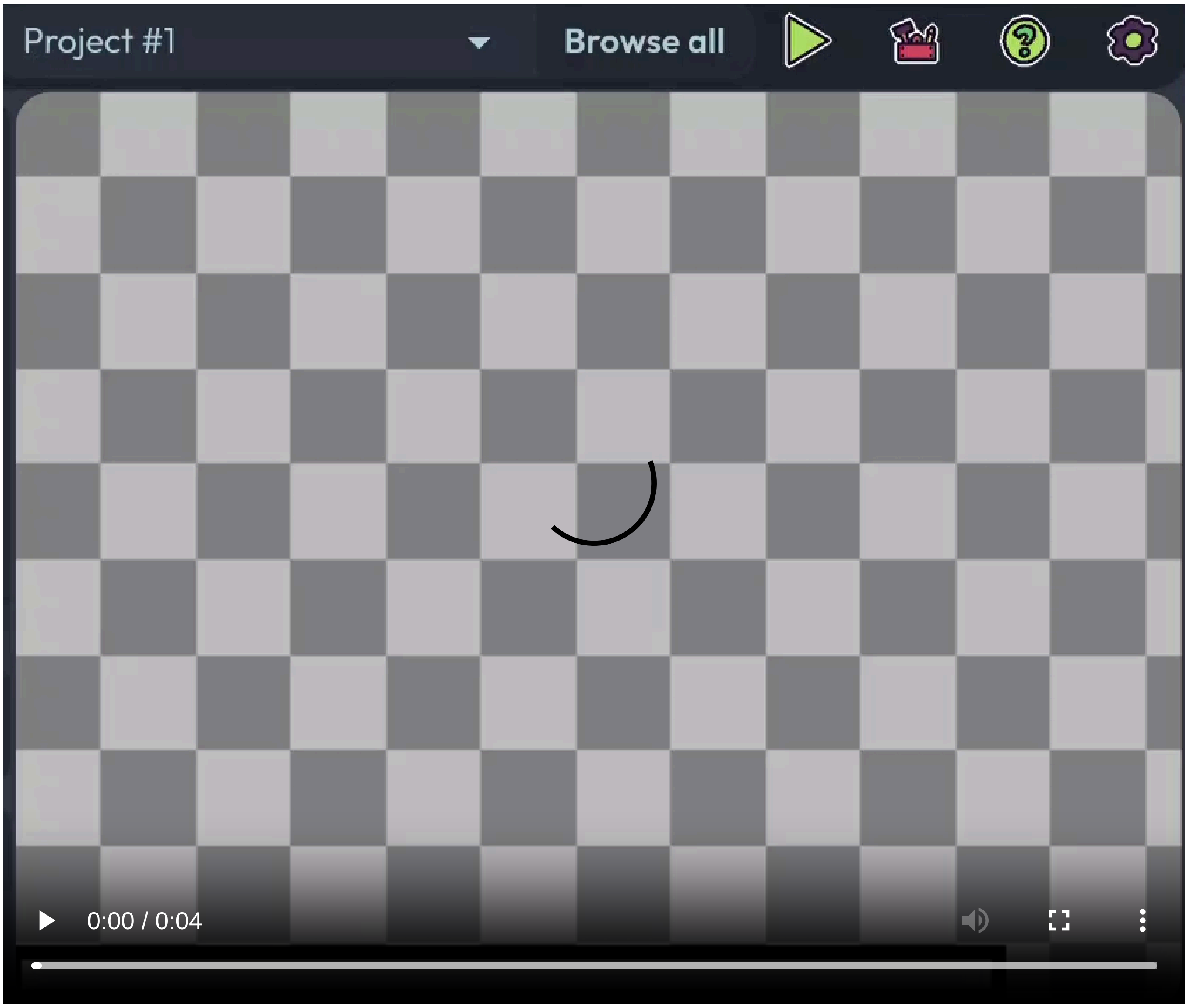
```
let platform = add([
    rect(width(), 15),
    pos(0, height() - 20),
    outline(5),
    // new code below
    body()
])
```



Hmm... the platform started to fall too... We don't want that! We want the platform to take part in physics, but we don't want to apply gravity to it. How can we do that? There's a way we can configure `body` to turn off gravity:

```
let kat = add([
    sprite("kat"),
    body(),
    pos(center())
])

setGravity(1000)

let platform = add([
    rect(width(), 15),
    pos(0, height() - 20),
    outline(5),
    // new code below
    body({ isStatic: true })
])
```

What happens now?



Well, the platform stays in place, but our character is still falling through it! Why? We told the game that they should have physics, doesn't that mean our character should stop when it touches the platform?

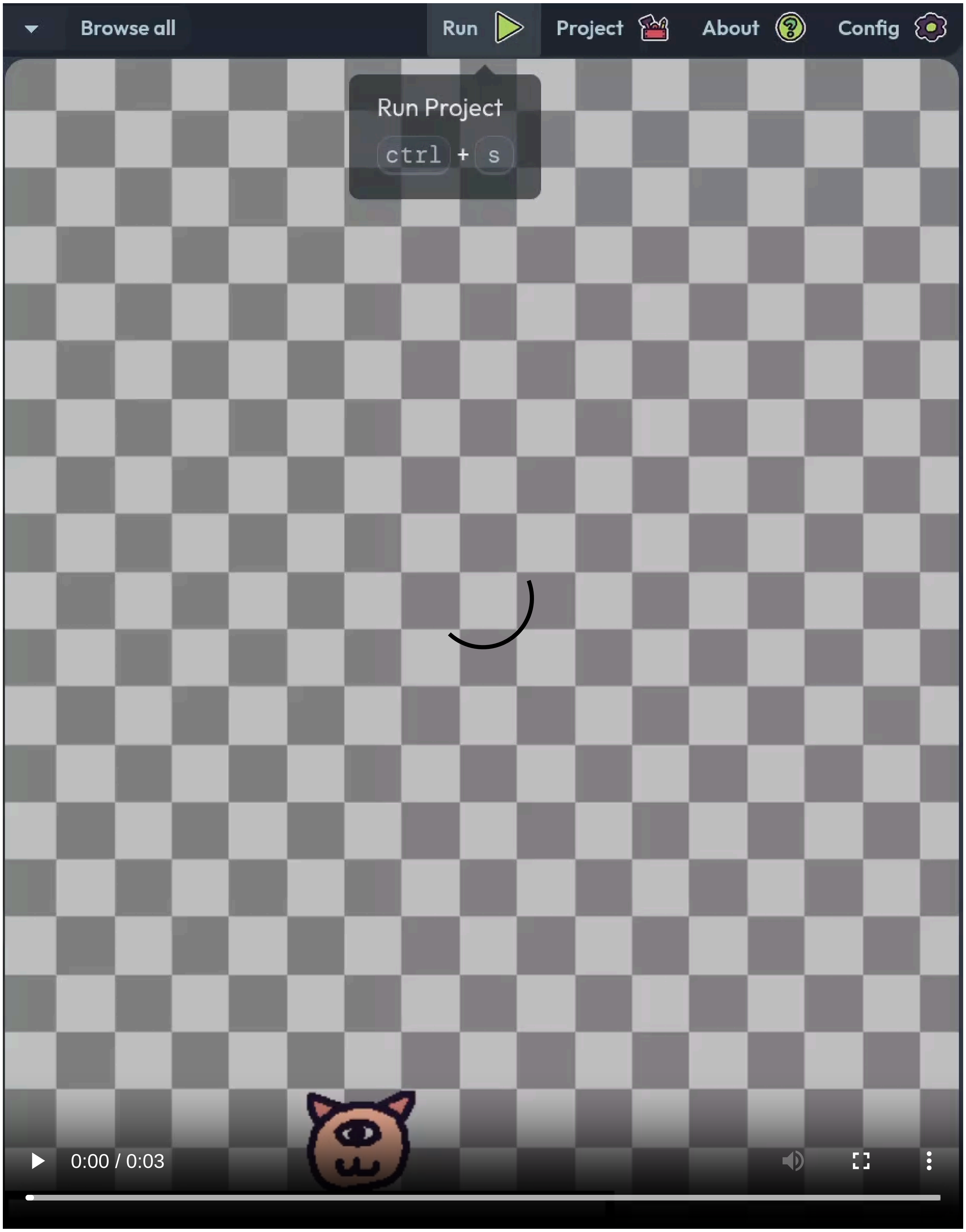
This is another important concept in games: **hitboxes**. Think of it like a box that goes around our player and our platform.

> Draw this on the whiteboard.

The game needs to know what the **hitboxes** are before it will register that two things have come into contact, or "collided" with each other. The **hitbox** is a "property" of our game objects, so let's add it:

```
let kat = add([
    sprite("kat"),
    body(),
    pos(center()),
    // new code here
    area()
])

setGravity(1000)

let platform = add([
    rect(width(), 15),
    pos(0, height() - 20),
    outline(5),
    body({ isStatic: true }),
    // new code below
    area()
])
```

That `area` function tells the game that the `kat` and `platform` have a **hitbox**, and it will automatically determine what it should look like. With that, the player no longer falls through the platform:



We're making progress!

Controlling the Character

We have gravity working and the player won't fall through the platform. Now, we need to add movement. Let's start with jumping.

Jumping

What we want to happen is: when you hit the spacebar, the player jumps. Our `kat` variable has a function for this: `onKeyPress`. When that function is called with the key that we provide, it will run another function for us. Let's see that in action:

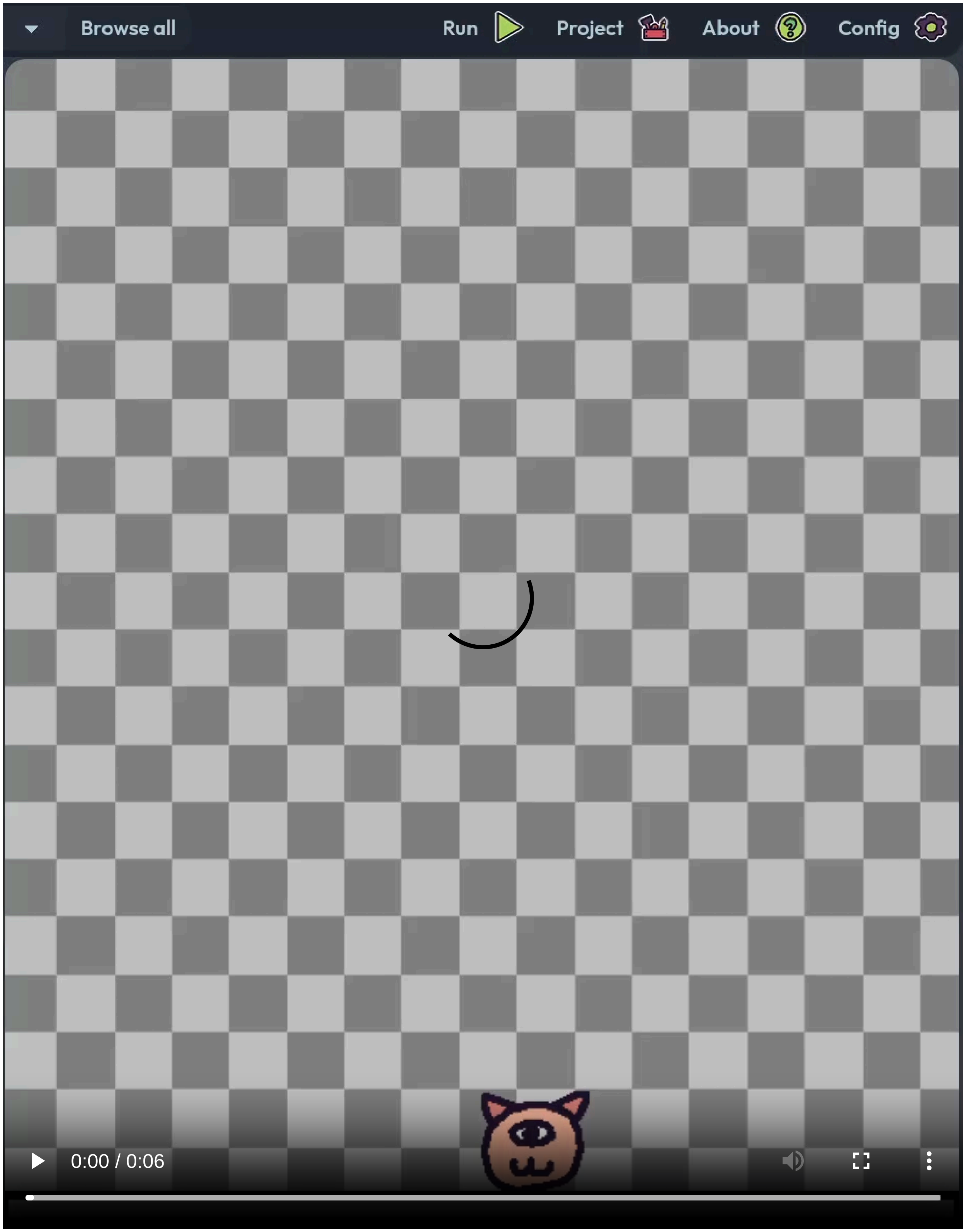
```
// add this below all the existing code

kat.onKeyPress("space", function () {
    kat.jump()
})
```

We've been *using* functions before, but now we are *defining* our own. Let's break this down:

1. We tell the computer to do something when a key is pressed (`kat.onKeyPress`)
2. We tell it which key on the keyboard to use (`"space"`)
3. We tell it to run some code when the space key is pressed `function`
4. We tell it to have the `kat` jump (`kat.jump()`)

When we've been calling `console.log` before, we only pass in one "value" or **argument** to the function. For `kat.onKeyPress`, we pass in two: the first is the string, the other is the function. They are separated by a comma, just like our list items.

If we run this code, then we'll see that our player can jump:



Preventing Double Jump

If you hit the space bar multiple times, you'll see that the player can jump as many times as they like before hitting the ground. It doesn't work like that in real life! We need to prevent them from doing a double jump.

> What have we learned so far that will run some code when something is true, but not run it when false?

That's right, an `if` statement! Let's add an `if` statement to check whether the player is on the ground or not:

```
// add this below all the existing code

kat.onKeyPress("space", function () {
    if (kat.isGrounded()) {
        kat.jump()
    }
})
```

`kat.isGrounded()` returns true if the player is on the ground, and false otherwise. If you run the game again, you can't double jump.

Moving Left

This works very similarly to jumping. Let's try changing our jumping code to move left:

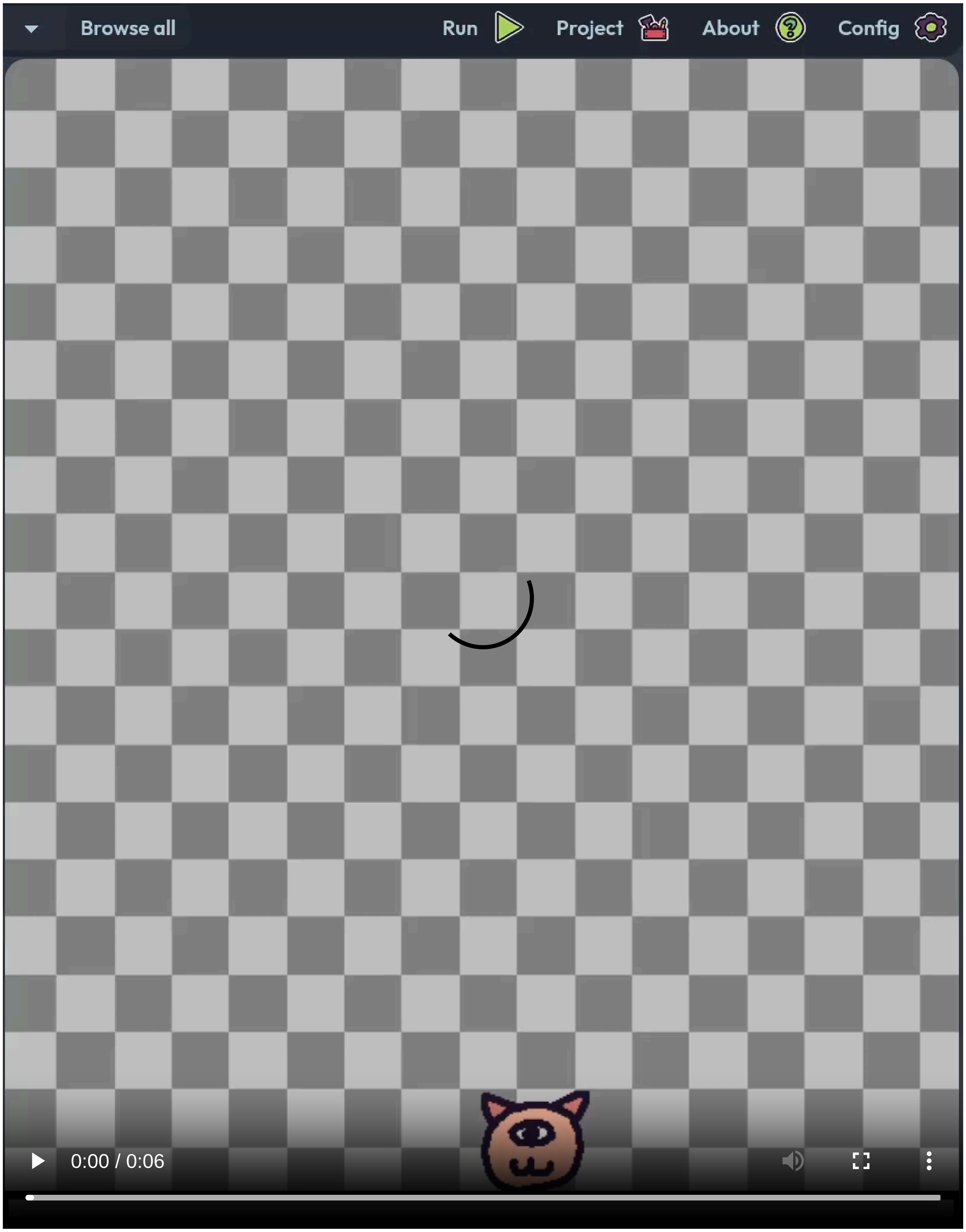
```
// add this below all the existing code

kat.onKeyPress("left", function () {
    kat.move(-250, 0)
})
```

Let's break this down

- `kat.move` takes two arguments: how fast to move *horizontally* and how fast to move *vertically*
    - We only want to move horizontally, so the second parameter is 0
    - A negative number means we move left, a positive number means we move right

> We will talk more about *why* positive and negative numbers do that in the game, but for now, that's all you need to know.

However, when we play the game, the player only inches left a little bit:

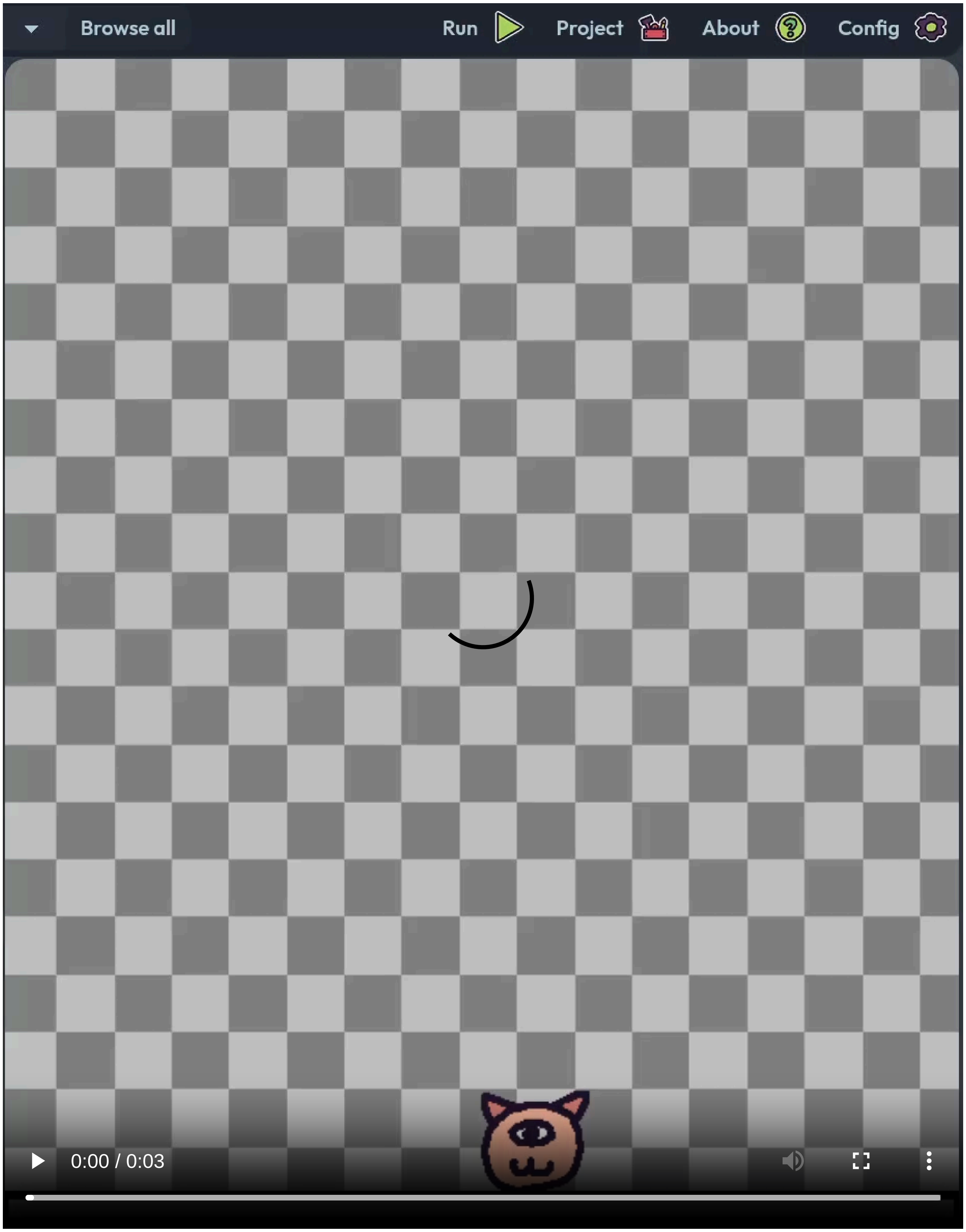


Each time it moves, I pressed the left arrow key on my keyboard. When I hold it down, nothing happens. For jumping, we only want to jump once when that key is pressed. If you hold down space bar, you only jump once. We don't want that for moving left and right, though.

There's a different function we can call for this instead of `kat.onKeyPress`: `kat.onKeyDown`. Let's change our code to that and see how it works:

```
kat.onKeyDown("left", function () {
    kat.move(-250, 0)
})
```

Now, it should move smoothly:



Exercise: Moving Right

> Ask them to try this on their own before showing the code.

```
// add this below all the existing code

kat.onKeyDown("right", function () {
    kat.move(250, 0)
})
```

Objective: Get to the Goal

Add the Goal Object

```
// add this below all existing code
let goal = add([
    sprite("kat", { flipY: true }),
    area(),
    body(),
    pos(width() - 75, height() - 100)
])
```

Add Collision Detection

```
// add this below all existing code
goal.onCollide("player", function () {
    console.log("You did it!")
})
```

Show the "You Win" Text

```
// update the existing onCollide event handler
goal.onCollide("player", function () {
    let bubble = add([
        anchor("center"),
        pos(center()),
        rect(400, 100, { radius: 8 }),
        outline(5),
    ])
    bubble.add([
        anchor("center"),
        text("You did it!"),
        color(BLACK),
    ])
})
```

Next Steps

> Ask the class what they want to see added to this game.

Some ideas I have are:

- Add coins/scoring (like some kind of collectables)
  - Add a high score list?
- Add more complex platforms (like moving platforms)
- Using scenes to implement multiple levels