13 - Final Project

Building our final project. An interactive game!

Let's Build a Game!

In this chapter, we will be building a game that makes use of everything that we have seen so far. We will also learn a couple of more tricks along the way as well. The fact that we can build a simple game using the p5.js library is pretty impressive and very illustrative of the capabilities of this library.

Our game is going to be simple. It is a typing speed game where we will be rapidly displaying numbers to the player and expect the player to enter the current number on the screen using their keyboard. If they enter the correct number in the given amount of time, they score. We will keep track of the score to be able to display it at the end of the game. It would be great if the game presents a strong visual experience but the primary focus is going to be around getting the game logic right.

Let's create a breakdown of things that we need to create:

- We need to display a number on the screen every N frames.
- We don't want the number to remain static on the screen. It should be animated to make it easier or harder to read with time.

- That number needs to remain on the screen until the next number is displayed or until the player presses a key in an attempt to match the number.
- If the player entry matches to the number on the screen, we will display a success message. If not, the failure will be indicated.
- We need to keep the amount of success and failure, and after X many frames / or attempts display the results to the user.
- We need to find a way to restart the game after it is over.

Getting Started

The first item on our list is to be able to display a unique number on screen at regular intervals. Remember that we used the **remainder** operator to achieve this feat before. Here, we will be displaying a number in between 0 and 9 on the screen every 100 frames.

Output
JavaScript
HTML

```
                              6
```
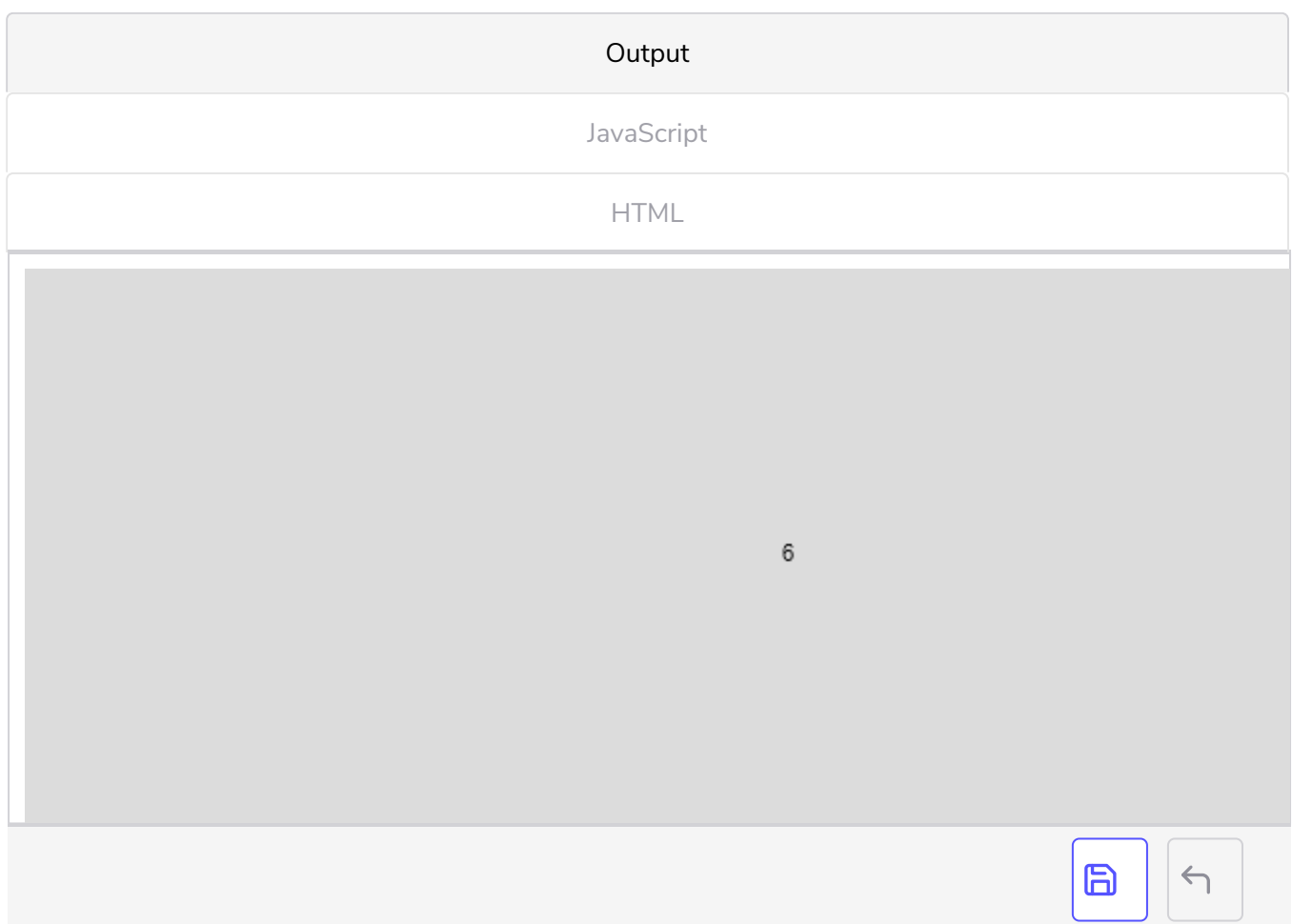
In this example, we are first initializing a variable called **content** in the global scope. Then in the **draw** function, we are using the **random** function to generate a random number on the first frame or every 100 frames and then save that value inside the content variable. Though the problem with the

random function is, it returns a floating point number. We would like to have whole numbers, integers, for the purpose of this game. So we are using the

parseInt function to convert the floating point number to an integer number. Remember that the **parseInt** function requires you to pass the second argument to set the base for the operation which is almost always going to be the number 10.

We are storing the generated number inside a variable called **content** and then pass that variable into a **text ** function that displays it in the middle of the screen.

We will need lots of custom behavior from the number that we will display on the screen; so we will create a JavaScript object to represent it. This way, the functions that we will be creating to manipulate the number such as transformation operations, color configurations, etc... can remain grouped under the object which helps with the organization of the program. We will call this new object **GuessItem**. I am well aware that's a terrible name but as they say, there are two hard things in computer science: *cache invalidation, naming things, and off-by-one errors*.

If we are to look at our code after this attempt at creating a JavaScript object that wraps the p5.js **text** function, it might look like we are adding additional complexity for no reason as our code grew almost twice in size. But containing the text drawing functionality under an object would help with organizing our code a lot down the road.

Output
JavaScript
HTML

0

Let's focus on the **GuessItem** object first. **GuessItem** is an object creating Constructor Function that requires three arguments: the x and y position and the scale of the shape that it draws to the screen. It also has two **methods** on itself. One of them is **getContent**, which generates a random number in between 0 and 10 and stores it inside a property called **content**. Another method it contains is **render** which displays the **content** property of a GuessItem object instance on the screen.

Every operation inside the **render** method lives under the **push** and **pop** function calls which allow us to contain the setting and transformation related state changes that happen inside this method contained in this object. Here, we are using the **translate** and **scale** transform functions to change the position and size of the text object. We didn't see the **scale** function before, but it's a transformation function that is very similar to **translate** and **rotate** functions. Just as the name implies, it controls the scale of the drawing area, and it has similar working principles to other transformation functions, so it is best to contain it in between **push** and **pop** functions.

We could have used a **textSize** function call for the size, but I usually find working with transform functions to be a bit more intuitive.

We will now be using this **GuessItem** constructor function to create an object that draws to the screen. We are instantiating a **GuessItem** object with several parameters **on line 10** and save it inside a variable called **guessItem**.

```
guessItem = new GuessItem(width/2, height/2, 1);
```

The number that the **GuessItem** is going to display is determined at the instantiation as well. Drawing this object to the screen happens **on line 13** using the **render** method it has.
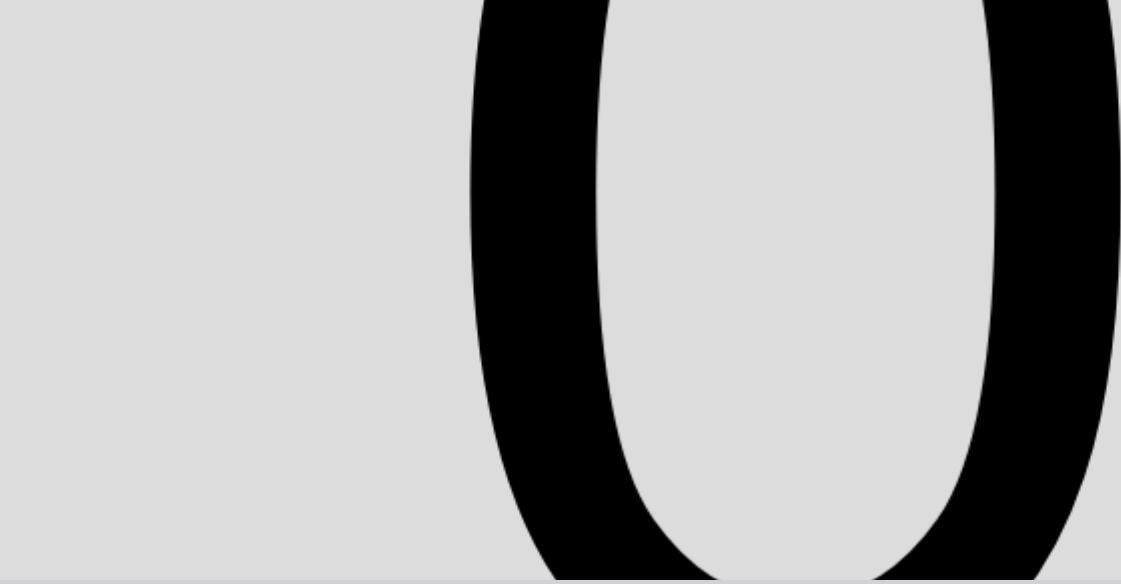
```
guessItem.render();
```

Let's make it so that the text grows in size during its lifetime to add some dynamism to the game.

Output
JavaScript
HTML

We added a way to increment the scale function with each call to the render function.

```
this.scale = this.scale + this.scaleIncrement;
```

We also added a new variable inside the **GuessItem** constructor function called **scaleIncrement ** that controls the speed of scaling. Play with this value to be able to change the pace of animation. We could, for example, increase this value to make the game harder.

We will add a bit more parameterization to our script to be able to control the way and the frequency the numbers are displayed.

| Output |
| --- |
| JavaScript |
| HTML |
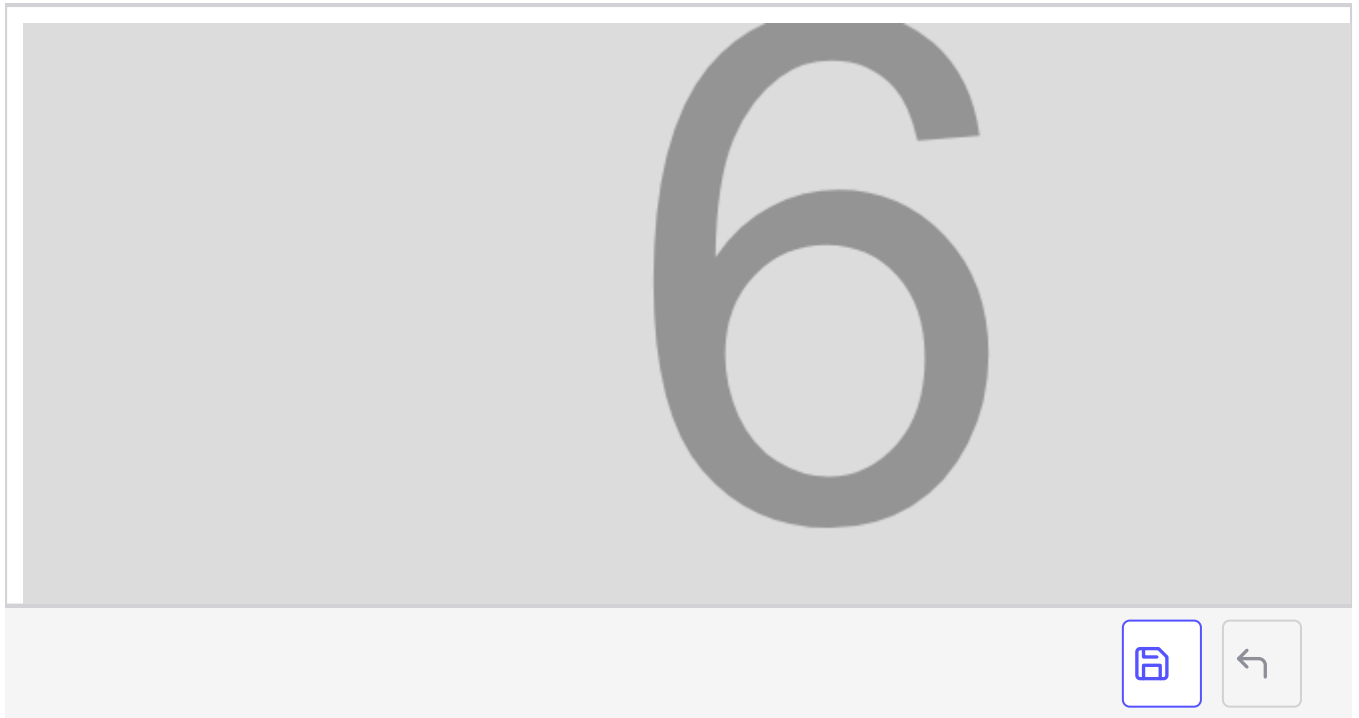
Here, we have a couple of more small tweaks. We added a **fill** function to the render method, and we are now dynamically setting alpha for the displayed number to get more transparent with each frame. I think that adds to the dynamism of the game. Set that number to something small to see things get stressful. We also parameterized the frequency of the creation of **GuessItem** using a global variable called **interval ** so that we can play with the value of that variable to make the game easier or harder.

By the way, can you guess why we named the number generating function **getContent**? That's because after we are done with this game, it should be a fairly trivial thing to update the game to display words instead of numbers on the screen. Keeping our function names generic helps a little bit with the future expansion work that we might want to do for this game.

So far, we only completed two items from our to-do list which are displaying a number on the screen by using a given interval and having that number animated on the screen to add dynamism to our game. In the next chapter, we will handle the player interaction.

User Interaction

We still have the outstanding task of fetching the user input and comparing it to the number on the screen. Let's implement that.

5
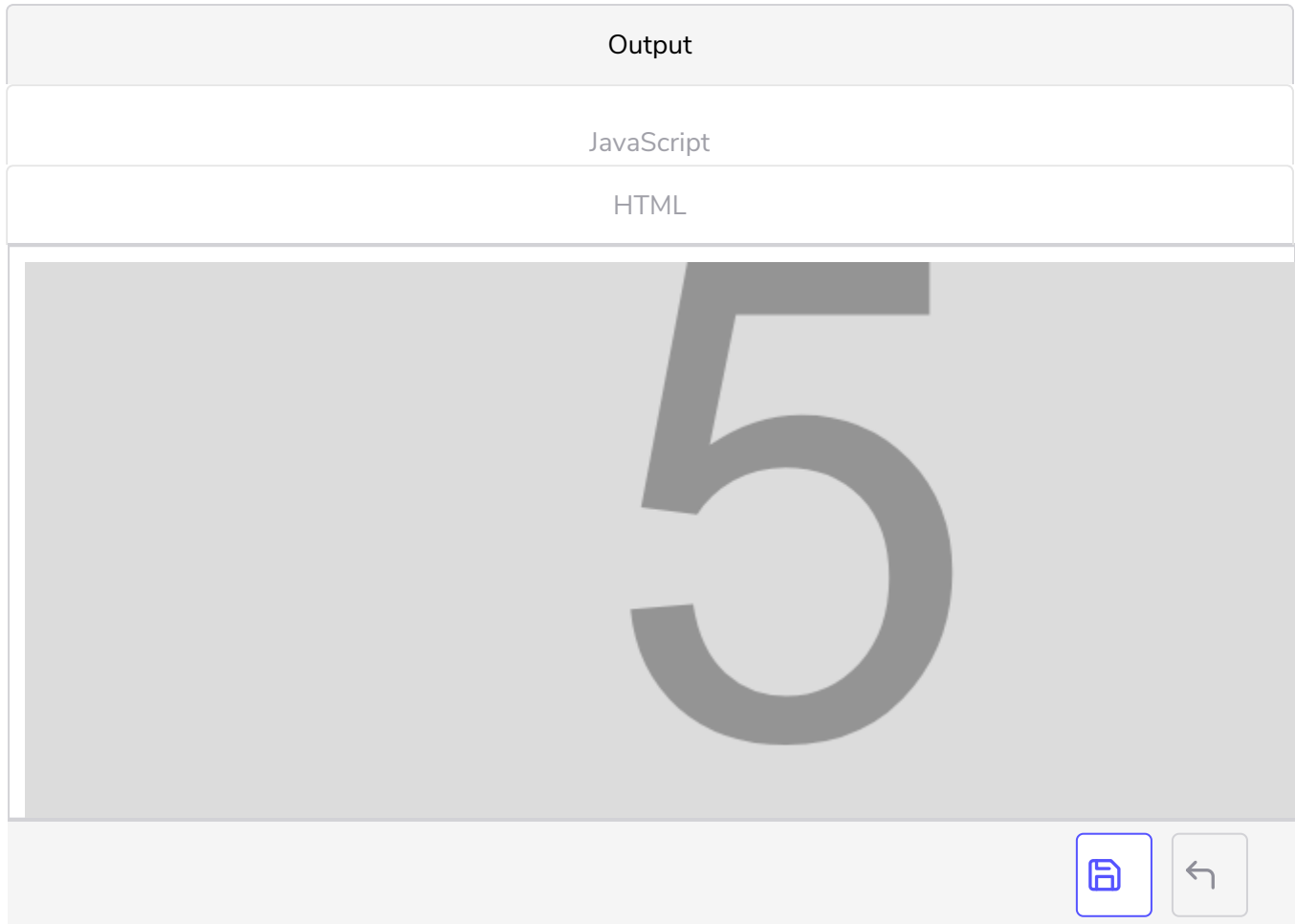
We updated the code in a bunch of places. To be able to achieve our task, we implemented a new method on the **GuessItem** object called **solve**. The **solve** method gets a user input and returns either **true** or **false** depending if the given user input matches to the **GuessItem content** variable. We end up saving the result inside a **solution** global variable.

```javascript
this.solve = function(input) {
    var solved;
    if (input === this.content) {
        solved = true;
    } else {
        solved = false;
    }
    this.solved = solved;
    return solved;
}
```

To be able to get user input, we have created a p5.js event function, **keyPressed**, which would be called every time the user presses a key. Inside this **keyPressed** function we are calling the **solve** method of a **guessItem** object to see if the pressed key matches the content of the **guessItem**. If so, the solution variable will be **true** and if not it would be **false**.

```
function keyPressed() {
    if (guessItem !== null) {
        console.log('you pressed: ', key);
        solution = guessItem.solve(key);
        console.log(solution)
        guessItem = null;
    } else {
        console.log('nothing to be solved');
    }
}
```

We are only reading the key presses from the player if there is a **GuessItem** that exists. That's because we are now assigning a **null ** to the **guessItem ** variable once the player makes a guess. Doing so effectively gets rid of the current **guessItem** object. That allows us to prevent the player from making multiple guesses for a number. Since the **guessItem** variable can now have a **null** variable, meaning there might not be a guess item present in the game because the user tried to guess its value, our call to the **render ** method might fail. To prevent that from happening, we are putting that **render ** call inside a conditional. Additionally, we have a couple of **console.log** functions inside the **keyPressed** function to have a sense of what's going on by looking at the console messages.

As a testing measure, we have added a conditional that changes the background color to black if the player guess is wrong and to white, if it is correct using the **solution ** variable.

Having said all that, this code doesn't work right now. Even our correct guesses are turning the screen to black. Can you guess why?

It turns out the reason is that the **keyPressed** function captures the pressed keys as *strings* whereas the generated content inside the **GuessItem** object is a *number*. Using triple equal signs, ===, we are looking to see if there is strict equality in between these two values and there is none - a number is never equal to a string - so our function returns **false**. To be able to fix this issue we are going to convert the number generated into a string by using the JavaScript function **String.**

```
    function getContent() {
        return String(parseInt(random(10), 10));

    }
```

Keeping the User Score

To be able to give feedback to the user as to how they are doing in the game, we will start storing their scores. We will make use of this stored data to make the game stop after a set amount of guesses or losses.

Output
JavaScript
HTML

3
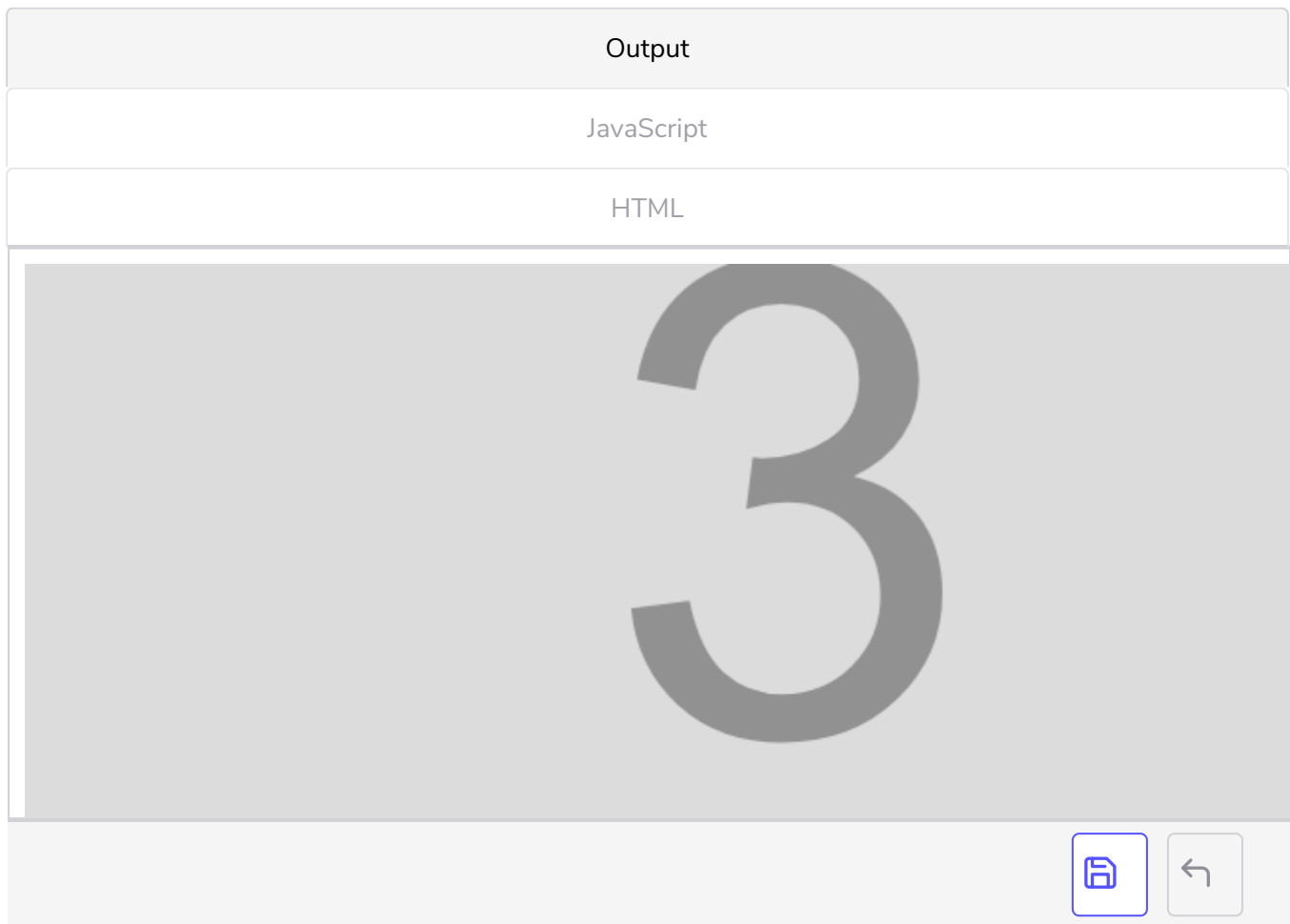
We created a **results** array to be able to store the player score. Every time the player makes a correct guess, we push a **true** value in there; and every time the player makes a wrong guess we push a **false.**

```
    if (solution) {
        results.push(true);
    } else {
        results.push(false);
    }
```

We should also build some functionality to get the value of the **results** array and evaluate it. For that purpose, we will be building a function called

getGameScore. It will get the **results** array and evaluate it to see what the current user score is.

Output
JavaScript
HTML

0

Our script is growing in size and complexity! Here is the most recent function that we have added **getGameScore**. It takes the score variable and loops through it to aggregate the number of wins and losses, as well as the total amount of guesses.

```javascript
function getGameScore(score) {
    var wins = 0;
    var losses = 0;
    var total = score.length;

    for (var i = 0; i < total; i++) {
        var item = score[i];
        if (item === true) {
            wins = wins+1;
        } else {
            losses = losses+1;
        }
    }

    return {win: wins, loss: losses, total: total};
}
```

We added a conditional at the beginning of the **draw** function to check the results of the **getGameScore ** function. If there are three losses or a total of 10 guesses the conditional executes what basically has a **return** statement in it.

```
    var gameScore = getGameScore(results);
    if (gameScore.loss === 3 || gameScore.total === 10) {
        return;
    }
```

Any line that comes after the **return** statement won't get executed since the current loop will terminate and a new one will begin - which will also terminate as long as the player's score remains the same.

```
    if (gameScore.loss === 3 || gameScore.total === 10) {
        return;
    }
```

We need a mechanism to restart the game at this point. First, we will build a screen that would get displayed when the game is over to display the player's score and to prompt the player to press a key, ENTER, to restart the game. Secondly, we will make it so that if the player is to presses the ENTER key after the game is over, it will get restarted.

Output
JavaScript
HTML

3

Let's see what we did with the **displayGameOver** function first. There are couple things happening here that we didn't learn about before.

```
function displayGameOver(score) {
    push();
    background(255);
    textSize(24);
    textAlign(CENTER, CENTER);
    translate(width/2, height/2);
    fill(237, 34, 93);
    text('GAME OVER!', 0, 0);
    translate(0, 36);
    fill(0);
    text('You have ' + score.win + ' correct guesses', 0, 0);
    translate(0, 100);
    textSize(16);
    var alternatingValue = map(sin(frameCount/10), -1, 1, 0, 255);
    fill(237, 34, 93, alternatingValue);
    text('PRESS ENTER', 0, 0);
    pop();
}
```

The first thing you should notice is that the **translate** function call results accumulate. If we perform a **translate** of `(0, 100)` after `width/ 2, height/2`, the resulting **translate** would be `width/2, height/2 + 100`.

Another thing that is new in this code is the p5.js **sin ** and **map** functions that we are using to create a blinking text. A **sin** function calculates the sine of an angle. Given sequential values the resulting **sine ** value would alternate in between -1 and 1. -1 and 1 are hardly useful to us as numeric values in our use case though. A value that alternates in between 0 and 255 would be vastly more useful if we are to use this value to set **alpha** of a **fill** function. This is where the **map ** function comes into play. The **map** function maps the given value that is within the given range (second and third arguments) to the new given range (fourth and fifth arguments).

```
var alternatingValue = map(sin(frameCount/10), -1, 1, 0, 255);
```

We are mapping the result of the **sin** function that is in between -1 and 1 to 0 and 255.

Instead of simply executing a **return** statement we can instead call this new function to display a message to the player. Next thing we implemented is a way to restart the game once it is over. For this, we require two things. First we need a way to respond to the **ENTER** key. And then we need to re-initialize the relevant game variables to create the impression of a new game starting.

This is the part of the **keyPressed** function that responds to the **ENTER** key.

```
if (gameOver === true) {
    if (keyCode === ENTER) {
        console.log('restart the game');
        restartTheGame();
        return;
    }
}
```

We are using the **keyCode** variable alongside with the **ENTER** variable to respond to the ENTER key press.

The contents of the **restartTheGame** function are simple. It just reinitializes a couple of variables that are in global scope such as the user score to make it start working again.

```
function restartTheGame() {
    results = [];
    solution = null;
    gameOver = false;
}
```

And this is it! We could keep working on it to make the game experience much better by tweaking the mechanics and enhancing the visuals of the game. But we have laid down the foundation that makes up the skeleton of our game which can now be developed further according to your specific needs.

Final Code

This is the final code. I decided to do a couple of updates for the version I was

working on. Instead of displaying numbers, I decided to display the words for

the numbers. I find that to be more visually pleasing and also more challenging from a gameplay point of view. I also added a new method into the **GuessItem** called **drawEllipse** that draws ellipses on the screen alongside with the words for a more visually engaging game. Finally, I tweaked the game parameters a bit to make the timing right and added messages to be displayed whenever the player enters a right or wrong number.

Output

JavaScript

HTML
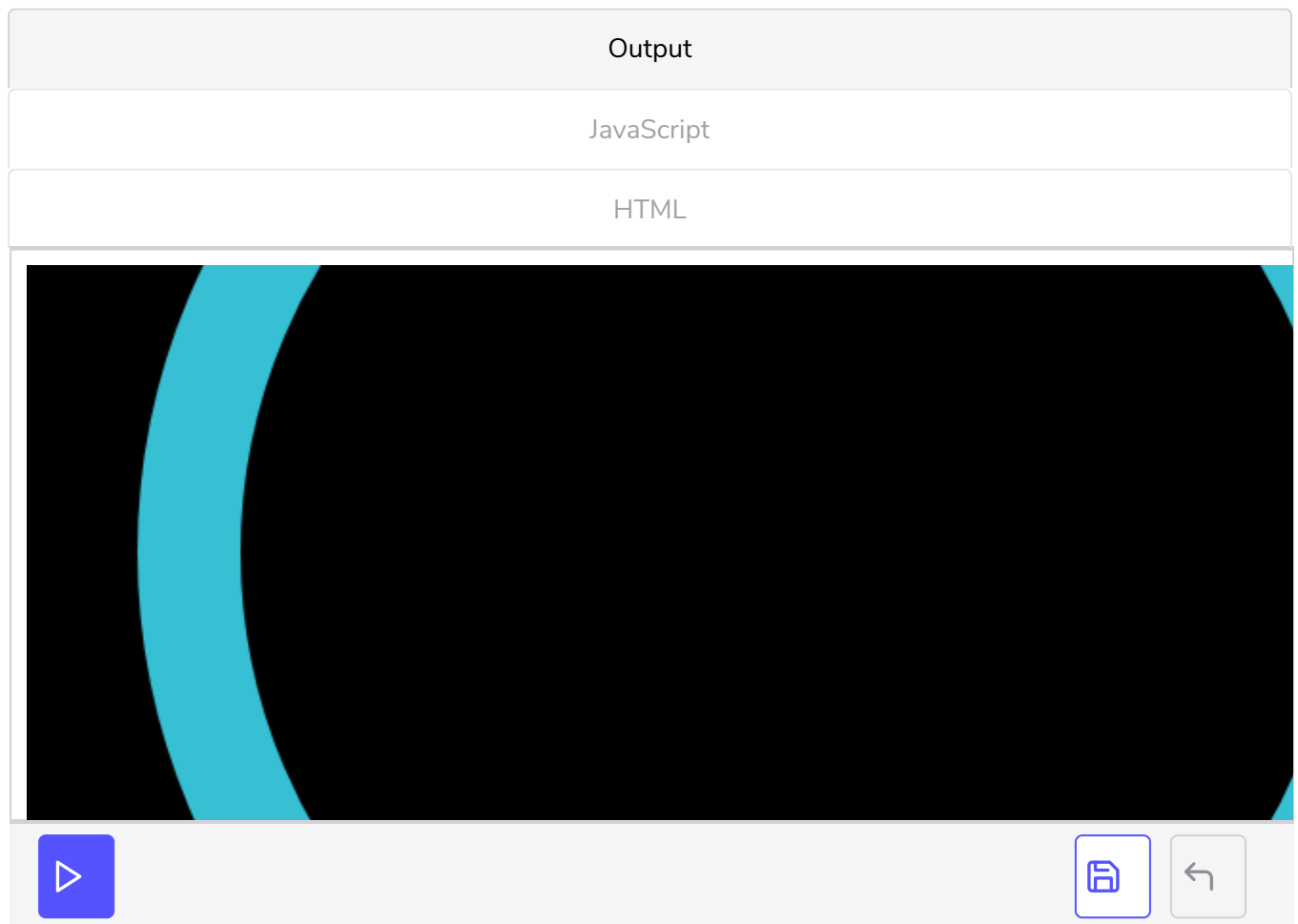
The biggest change to the code is the **solutionMessage** function so let's take a look at that in a bit more detail. Previously we were just using an if-else statement based on the value of the **solution** variable to decide what to display on screen. If the solution was **true**, we were displaying a white background, if the solution was **false** we were displaying a black background.

Now if the solution is either of these values (**true** or **false**) we are passing it to a function called **solutionMessage** which chooses a random message to display using **gameScore.total** as a seed for the **random** function.

```
if (solution == true || solution === false) {
    solutionMessage(gameScore.total, solution);
```

```
    }
```

Inside the **solutionMessage** function, there are two arrays with of bunch of message values that are to be displayed based on the value of the **solution**.

```
    if (solution === true) {
        background(255);
        messages = trueMessages;
    } else if (solution === false) {
        background(0);
        messages = falseMessages;
    }
```

We pick a random value from these arrays by converting the return value of the **random** function to an integer.

```
    text(messages[parseInt(random(messages.length), 10)], width / 2, heigh
t / 2);
```

Summary

This was definitely a challenging example that puts everything we have learned so far into a test.

It is very impressive that we can build a game just by using p5.js that can run on the web and can be played by millions of people. And it wasn't all that difficult as well; the entire program is just around 200 lines. There is certainly room for improvement where we can make the game difficulty dynamic based on the player performance, add more visual flair and add a dynamic scoring system where we can assign different points to correct guesses based on the amount of time it took to guess a number. The game can be converted to display words instead of numbers. It can show images that you need to type the name for or calculations that you need to answer. The possibilities are numerous!

Having said that, p5.js might not be the best platform to create games with if we wanted to build more advanced projects. A proper game library would come with features such as an asset loading system, sprite support, collision detection, physics engine, particle systems… which is more often than not required when building advanced games. This is not to say you can't use p5.js

to build a game, though; we just proved that it is entirely possible. It is just that there are other libraries out there that are more specialized around that solution whereas p5.js is more tailored towards creating interactive, animated experiences on the web.