Introduction

This guide will detail a process for developing an HTML5 web application with the following requirements:

Develop a spelling game for children, on the iPad. The game logic should work as follows:

- In some fixed location on the screen, a word should appear.
- The letters of the word should be moving around the screen.
- The user will find the first letter, and drag it to the location of the word. The user will then repeat this for the remaining letters until the word is correctly spelled.
- If at any point, an incorrect letter is chosen, the user should be notified, and allowed to continue trying.
- After a word has been spelled, a new word should appear, and the process repeats.

I will outline the process I used to develop this project from start to finish, and explain the motivations and thoughts behind the many decisions I made during this process. I hope that this example project will be of assistance to students who are new to web development.

Also, please note that I do not claim that the method I used to develop this application is absolutely correct. It is simply meant to serve as an example. Feel free to make your own changes and additions as you follow this guide.

Finally, this guide assumes basic familiarity of the following technologies and frameworks: HTML5, CSS, JavaScript, and jQuery. If you do run into something unfamiliar, I would advise the use of online resources to gain a better understanding.

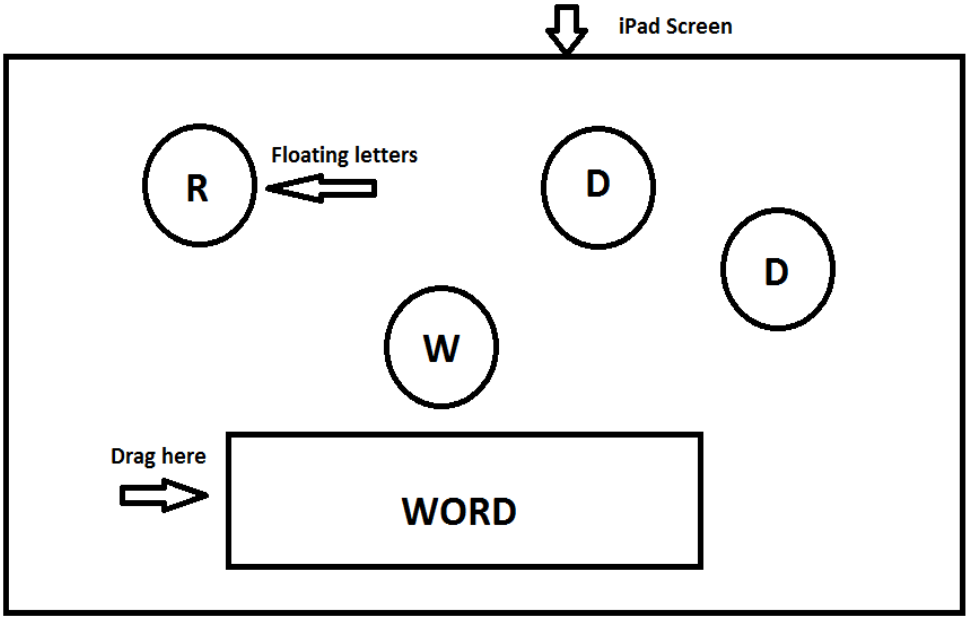
Getting Started

Before immediately diving into the coding process, I've always found it useful to first get an image of how the application will look like, and how the user will be able to use it. This will help guide me during the coding process.

Given the requirements, I can imagine this iPad Spelling Game work in the following manner:

- Have the floating letters look like "bubbles", that move randomly from one location to another.
- The user will drag the letters, in order, to a box, that has the word written inside
- If the word is correct, a new word will come up.

From that, we can go one step further, and draw some simple sketches of how the application will appear. I used to following sketch for my application. Note that the application runs in a landscape orientation.



Now that we have an idea of how our application will behave, the next step is to gain a basic understand our coding requirements.

The first thing I usually consider at this stage is the platform I am developing for. As mentioned before, we are developing a *mobile web application* on the iPad. **Since it is a web application, this immediately hints that we will use HTML5**. Moreover, since it is a mobile web application, we may also want to consider mobile application frameworks such as:

- jQuery Mobile
- Sencha Touch
- Titanium
- jQTouch
- etc.

Obviously, the next step is to research the above frameworks. From my research, I have learned that mobile web application frameworks provide rich support for mobile user interface development. They also provide support for touch events.

From the sketches, however, we can see that our application has a very simple user interface--there is only a single page. It seems that given the requirements of our game, the use of a mobile application framework is unnecessary. **Hence, for this project, we will opt out of using a mobile web application framework.**

Next, from the requirements, it should be clear that our application will also consist of the following:

- Animation
- Touch interactions

Given the above, we will likely need to use several frameworks to provide support for our requirements. However, I will research and choose the appropriate framework as each requirement becomes relevant in the development process. For now, I will focus solely on the initial requirement: getting the application set up for our platform, the iPad.

Platform Set Up

We will now set up the application for the iPad. The initial step is to first create an HTML page called **index.html**, which will contain our application. We do this as follows:

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8" />
	</head>
	<body>
		<div id = "container">

		</div>
	</body>
</html>
```

Above, we have simply created an HTML5 file, which contains a `div` with the `id` "container", that will hold our application.

To set up our application for the iPad, we must first set the appropriate dimensions for our container. From a quick search online, I found that for the iPad, the usable area in Safari (as it is a mobile web application), is 1024x690 pixels. Now, we must set this as the dimensions of our container. We accomplish this using CSS.

Create a CSS file, called **style.css**, as shown below, and add a link to it in the `head` of your index.html file: `<link rel="stylesheet" href = "style.css">`

```
#container{
	width: 1024px;
	height: 690px;
	border: 1px solid #BDD8DA;
	background-color: #BDD8DA;
}
```

In this file, we have first set the height and width of the container div (`#container`) to the appropriate dimensions. Next, in order to make the div distinguishable, I added a solid border and a background color, both set to the color `#BDD8DA`.

To view the application at this current stage, click here

As you can see, we have set the dimensions of our container div appropriately, and displayed the div using a border, and background. However, there appears to be an extra margin surrounding our div; this is easily apparent when viewed using the iPad. To get our application to properly fit on the iPad screen, we need to remove these extra margins. A quick online search reveals that these extra margins are simply the default for the `body` tag in html. To remove them, we simply add the following to our **style.css** file:

Now, the application will look like this.

As you can see, the default margins have been removed, and our application is now "set up" for the iPad. All further elements can be placed within the container div.

Initial Development

From here, we are ready to being coding the actual components of the spelling game. As a reminder, we need to accomplish the following things:

1. Add a location where the letters will be dragged to (droppable location)
2. Add a word in the droppable location, and its letters onto the screen.
3. Animate the letters
4. Add the touch interactions
5. Create the game logic

In the initial development phase, I have chosen to first finish bullets 1 and 2, as they are prerequisites for the rest of the application.

STEP 1:

First, I will create a **droppable location**. To do this, I simply create a div with id “droppable”, and place it within the container div. This is shown here:

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8" />
		<link rel="stylesheet" href = "style.css">
	</head>
	<body>
		<div id = "container">
			<div id= "droppable"></div>
		</div>
	</body>
</html>
```

Next, we must make this div visible, and place it in the correct location. We will accomplish this using CSS:

```
#droppable {
	position: absolute;
	left: 260px;
	top: 450px;
	height: 195px;
	width:480px;
	background-color: #FFFFFF;
	-moz-border-radius: 45px;
	border-radius: 45px;
}
```

As you can see above, we first set the position of the droppable div, using the position, left, and top properties. Note that because position is set to absolute, the left and top properties are set with respect to the droppable div's containing element: the container div.

I have also set the height and width of the div, and the background color to white (`#FFFFFF`), to make it visible.

Lastly, I added rounded corners to the droppable div, for aesthetic purposes. The `border-radius` specifies the radius of the corners, and the `-moz-border-radius` specifies the same for the Firefox browser.

To view the application at the current stage, click here.

STEP 2:

Now, we are ready to proceed to step 2: adding the word to the droppable div, and its letters onto the screen. However, before we begin, we must first realize what this step entails. As this is a spelling game, we will presumably have multiple words that we need to display, in succession. Hence, the elements of the container div will constantly change throughout the game; for example, after the user successfully spells a word, we must move onto to the next word, and add its letters to the container. Hence, we will require **DOM manipulation** in our application. To accomplish this, we can use the scripting language **JavaScript**. In addition, I will also use the popular JavaScript library **jQuery**, as it provides excellent support for DOM manipulation.

To be able to use jQuery, we will add the following `script` tag to our index.html file, inside the `head` tag:

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js">
</script>
```

Now our application will be able to use jQuery.

To add the word to the droppable div, we will create a script file: **anim.js** (shown below), and add a reference to it to it our **index.html** file using: `<script src="anim.js"></script>`

```
//execute this function when the DOM is fully loaded
$("document").ready(function(){
	//an array of the words we will use in our game
	var wordArray = ["cat", "dog", "tiger"];
	//an index for the current word
	var wordIndex = 0;
	//the current word
	var currentWord = wordArray[wordIndex];
	//add the current word to the droppable div
	$("#droppable").text(currentWord);
});
```

Within this file, we will define the array of words that our game will contain, and then add the first word to droppable div. All of this is placed inside a function. The `$("document").ready()` statement ensures that the function will execute only after the DOM is fully loaded.

After adding the word to the div, we will update our **style.css**, as shown here:

```
#droppable {
	position: absolute;
	left: 260px;
	top: 450px;
	height: 195px;
	width:480px;
	background-color: #FFFFFF;
	-moz-border-radius: 45px;
	border-radius: 45px;
	font-size: 500%;
	color:#7D9C9F;
	text-align: center;
	line-height: 195px;
}
```

Above, we first increased the font size of the new word. Then, we changed its color to match the background of the container. Finally, we horizontally and vertically centered the word within the div. The `text-align` property will center it horizontally. To center it vertically, we use the `line-height` property, and set to the same height as the droppable div. We are able to use the `line-height` property to vertically center since there is only a single line inside the div.

To view the project at the current stage, click here.

The next part of step 2 is to add the letters of the word onto the screen. As shown before, the first word in our array is “cat,” so we will add its letters to the container. We achieve this by creating a function called `renderLetters()`, which takes a variable `word` as a parameter.

```
//this method places the letters on the screen randomly. it then returns the
letters in an array
function renderLetters(word){
	//create an array to hold the letters of the word
	var letterArray = new Array();
	for(i=0; i < word.length; i++){
		//get the letter
		var letter = word.charAt(i);

		//create an id for the letter and add it to the letter array
		var id = "id"+i;
		letterArray.push(id);

		//create letter div
		var letterDiv = "<div id="+id+" class = 'letter'>"+letter+"</div>"
		//add it to the container
		$("#container").append(letterDiv);

		//place the letter in a random location using jquery's css method
		 var leftLocation = (Math.floor((Math.random()*800)+100));
		var topLocation = (Math.floor((Math.random()*350)));
		$("#"+id).css({"left":leftLocation+"px"});
		$("#"+id).css({"top":topLocation+"px"});
	}
	//return the array
	return letterArray;
}
```

We place the `renderLetters()` function in the **anim.js** file, and call it at the end of the $(“document”).ready() function that we defined earlier.

In the function, we first create an array to hold all of the letters of the word. Then, we iterate through the letter array, and add them to the container div. Note that each letter has a unique id, and that all letters are part of the `letter` class. After adding them to the, we select a random `left` and `top` position for each letter, and set the `left` and `top` CSS properties accordingly using jQuery’s css() method. In order to use the `left` and `top` properties, we must also set the `position` property for each letter to either `absolute` or `relative`. We do this in the **style.css** file, where we set the position of the `letter` class to `absolute`.

We have also defined the height and width of each letter div, and added curved borders around the div. Moreover, we have specified a background color, font color, and font size. Finally, we centered the letter within the div. All of these settings follow the same logic as the droppable div mentioned before.

The relevant code is shown below:

```
.letter{
	position: absolute;
	height: 100px;
	width: 100px;
	-moz-border-radius: 50px;
	border-radius: 50px;
	border: 1px solid #BDD8DA;
	background-color: #FFFFFF;
	text-align: center;
	font-size: 400%;
	color:#7D9C9F;
	line-height: 100px;
}
```

At this stage, the application looks like this.

Note how the letters are given a new, random location each time the page is refreshed.

At this stage, we have essentially completed our "initial development" phase. We have set up the platform, and implemented some very basic, yet essential elements of our application. As a reminder, here is our task list:

1. ~~Add a location where the letters will be dragged to (droppable location)~~
2. ~~Add a word in the droppable location, and its letters onto the screen.~~
3. Animate the letters
4. Add the touch interactions
5. Implement the game logic

As the list indicates, we will next focus on a critical portion of our application: animation.

Adding Animation

As mentioned in the project description, we would like the letters to have a “floating” animation effect. Hence we need to determine how our animation will appear. As this is meant to be an example project, I have chosen a very simple animation scheme:

> Each letter will move from one random location to another random location in the container div. This animation will run continuously. If the movement is at the correct speed, it will hopefully provide an adequate “floating” effect for the letters.

The next step is to determine how to implement our animation scheme. Ideally, I would like to use a JavaScript framework that has strong support for animation, and provides clean, smooth animation. Based on personal experience, I know that a common library for creating simple animation in web applications is jQuery, using its **animate()** function. This function allows us to animate many CSS properties, including position properties such as `left` and `top`, which we are most concerned with for our application.

However, we must first ensure that using jQuery for animation will be satisfactory on the iPad, as that is our intended platform. To do this, I will write simple test program. As a side note, I always encourage the use of test programs when judging the usefulness of a particular framework or library. This allows us to quickly judge whether or not the framework will be useful for our needs.

I will write a small test program that demonstrates:

- Animation of position
- The ability to stop animation (as this is required when the user starts to drag the letter)

The test program will simply consist of a box that completes a single animation loop around the screen. Also, if the user ever clicks or taps the box, the animation will stop.

Our HTML file, **testIndex.html**, will look as follows:

```
<!DOCTYPE html>
<html>
	<head>
		<meta charset="UTF-8" />
		<link rel="stylesheet" href = "testStyle.css">
		<script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.7.2/jquery.min.js"></script>
		<script src="testAnim.js"></script>
	</head>
	<body>
		<div id= "testDiv"></div>
	</body>
</html>
```

We have simply defined a div called "testDiv," which will serve as our box. Along with that, we have also added jQuery in the `head` tag, along with style.css, and anim.js (both of which will be defined next).

In our CSS file, **testStyle.css** file, we simply set the height, width, background color, and border for the testDiv. We also set the `position` property to absolute, so that we can manipulate the `left` and `top` properties using jQuery.

```
#testDiv{
	position: absolute;
	height: 200px;
	width: 200px;
	border: 1px solid;
	background-color: #BDD8DA;
}
```

Finally, we define our JavaScript file, **testAnim.js**. In this file, we bind use the jQuery animate() function to create the animation. The animate() function takes three parameters:

1. The property being animated, and its new value
2. The duration for this animation
3. An optional "onComplete" function that will execute after the animation.

Given these, we can see that we are creating an animation that will move the testDiv div around the screen.

Finally, we also bind “touchstart” and “click” events to the testDiv div, which if fired (caused either by the user touching, or clicking on the div), will cause the animation to stop via jQuery’s stop() method.

```
//execute when the DOM is ready
$("document").ready(function(){
	//if the user touches the testDiv, stop the animation
	//this is for the iPad
	$("#testDiv").bind("touchstart", function(){
		$("#testDiv").stop();
	});
	//if the user clicks the testDiv, stop the animation
	$("#testDiv").bind("click", function(){
		$("#testDiv").stop();
	});

	//Each animate function is called after the previous completes
	$("#testDiv").animate({"left": "700px"}, 5000, function(){
		$("#testDiv").animate({"top": "400px"}, 5000, function(){
			$("#testDiv").animate({"left": "0px"}, 5000, function(){
				$("#testDiv").animate({"top": "0px"},5000);
			});
		});
	});
});
```

This concludes our simple test program of jQuery’s animation. You can view the test clicking here.

As you can see, with this test we have shown that jQuery will allow us to both animate the position property, and easily stop the animation.

However, if we view the test using the iPad (the intended platform for this project), a clear issue arises: the animation is not smooth. Instead, it is quite blurry and choppy. The iPad, lacking the processing power, cannot properly handle something as heavy as jQuery’s animation. Although this will not hinder the functionality of our spelling game, it would be ideal if we had smoother animation.

Hence, I researched other animation techniques, which are more lightweight, and will allow us to have better results on the iPad. Of the ones I saw, I was most interested by the library **jQuery Transit**. Unlike jQuery’s animation() function, which uses JavaScript to create its animation, jQuery Transit uses *CSS3 Transitions*.

A bit of background information on CSS3 transitions:

CSS3 transitions allow for effects on CSS properties without the use of JavaScript, or Flash. Hence, it is quite lightweight. With CSS3 transitions, you can:

- Specify a property to “translate”
- Specify the duration of the transition
- Add advanced timing control, such as easing
- Set delays for transitions

CSS3 transitions are also supported by most major browsers (including Safari).

Read more here: http://www.css3.info/preview/css3-transitions/

From an initial glance, this method appears to be a promising replacement for our jQuery animation. Although, a potential difficulty is that CSS3 transitions are generally predefined in a CSS file, and can be messy to handle using JavaScript, which is essential for our application.

However, this is where the jQuery Transit library comes in, which appears to have simplified the process of using JavaScript to control CSS3 transitions through its **transition()** function. But we cannot be sure until we test it using a test-program (as we did for jQuery animation).

To test it, the first step is to add jQuery Transit to our **testIndex.html** file. That is, I will add:

```
<script src="jquery.transit.min.js"></script>
```

Our **testStyle.css** file remains unchanged.

Now, we will focus on implementing the animation, in our **testAnim.js** file. As we mentioned before, the jQuery Transit library provides a transition() function to support the CSS3 transitions. The parameters for the transition function are:

1. The property to translate
2. The duration (optional)
3. An “onComplete” function, which is called when the transition finishes.

This will take care of creating the animation:

```
//Each animate function is called after the previous completes
	$("#testDiv").transition({"x": "700px"}, 5000, function(){
		$("#testDiv").transition({"y": "400px"}, 5000, function(){
			$("#testDiv").transition({"x": "0px"}, 5000, function(){
				$("#testDiv").transition({"y": "0px"},5000);
			});
		});
	});
```

Note, an important distinction of CSS3 transitions is that instead of using `left` and `top` for position, `x` and `y` are preferred.

Next, we will focus on stopping the transition. This, however, appears tricky. The jQuery Transit library provides no support for transition pausing or stopping. Hence, we must implement it ourselves.

After some research, I learned that CSS3 transitions can be stopped on an element by setting the property `'WebkitTransition'` to `null`. Hence, we can now implement a function to stop the CSS3 transition:

```
$.fn.stopTransition = function() {
	this[0].style['WebkitTransition'] = null;
}
```

There a few things to note here:

- **`.fn`** is shorthand in jQuery for the prototype property in JavaScript.
- We use `this[0]` to access the testDiv div.

So, now our updated testAnim.js file looks like this:

```
$.fn.stopTransition = function() {
	this[0].style['WebkitTransition'] = null;
}

//execute when the DOM is ready
$("document").ready(function(){
	//if the user touches the testDiv, stop the animation
	//this is for the iPad
	$("#testDiv").bind("touchstart", function(){
		$("#testDiv").stopTransition();
	});
	//if the user clicks the testDiv, stop the animation
	$("#testDiv").bind("click", function(){
		$("#testDiv").stopTransition();
	});

	//Each animate function is called after the previous completes
	$("#testDiv").transition({"x": "700px"}, 5000, function(){
		$("#testDiv").transition({"y": "400px"}, 5000, function(){
			$("#testDiv").transition({"x": "0px"}, 5000, function(){
				$("#testDiv").transition({"y": "0px"},5000);
			});
		});
	});
});
```

To view the test program at this current stage, click here.

From the above test program, we can make several important observations:

- The animation, especially on the iPad, using CSS3 Transitions is much smoother than using jQuery animation.
- Currently, the transitions do not stop in place. Instead, they stop and move the element to its end position. For our application, we would like the element to stop at the position that it was either clicked or tapped on at. Hence, there is still some tweaking required for this method.
- Even if we stop the transition, the remaining "onComplete" functions are still called. However, this will not be a major issue for us, as we will likely not use the onComplete functions in our spelling game. Hence, we can most likely ignore this.

To rectify our stopTransition() function, we can try to use the following workaround:

1. First, retrieve the x and y position of the transition when it was clicked/tapped on.
2. Then, set the "`WebkitTransition`" to `null`, like before, to stop transition.
3. Immediately after, set the element's position to the x and y values retrieved in step 1.

This will hopefully set the elements position to the clicked/tapped location, instead of the end location. The updated stopTransition() function is show here:

```
$.fn.stopTransition = function() {
	//retrieve the x and y position values
	var id = $(this).attr("id");
	var matrix = $("#"+id).css("-webkit-
transform");
	var values =
matrix.split('(')[1].split(')')[0].split(',');

	//stop the transition
	this[0].style['WebkitTransition'] = null;

	//set the position of of the element to
the clicked/tapped location
	$("#"+id).css({"x":values[4]+"px"});
	$("#"+id).css({"y":values[5]+"px"});
}
```

To view the updated version of the example, click here.

As you can see, the element now stops in-place, instead of finishing the transition. Hence, we have satisfied both requirements (positional animation, and ability to stop animations) using the jQuery Transit framework. Furthermore, due to its superior animation in comparison to jQuery animation, especially on the iPad, we will use it for our spelling game project.

Adding Animation to Our Project

Now that we have decided to use jQuery Transit in order to create our animation, the next step is to implement it in our spelling game. Recall our simple animation scheme:

> Each letter will move from one random location to another random location in the container div. This animation will run continuously. If the movement is at the correct speed, it will hopefully provide an adequate "floating" effect for the letters.

In order to do this, we need to repeatedly animate each letter from its current location to another random location. This is done as follows:

First, we need to download jQuery Transit, and add it to our **index.html**, as shown:

```
<script src="jquery.transit.min.js"></script>
```

Next, in our **anim.js** file, we will define a function, called **animateLetters()**, which will animate each letter to a random location:

```
//this method animates each letter
function animateLetters(letterArray){
	//for each letter, pick a random location, and time, and then animate to
that location
	for(i = 0; i < letterArray.length; i++){
		//get current letter
		var letter = letterArray[i];
		//choose random x location
		 var xLocation = (Math.floor((Math.random()*800)+100));
		 //choose random y location
		var yLocation = (Math.floor((Math.random()*350)));
		//choose random time
		var time =   Math.floor((Math.random()*5000)+7000);
		//transition the currrent letter. Note that we use x and y for
position, instead of left and top
	$("#"+letter).transition({x: xLocation+"px",y: yLocation+"px"}, time);
	}
}
```

In the above method, we simply iterate through the array of letters using a for-loop, and then animate each letter to a random position. Note that to specify position, we use *x* and *y*, instead of *left* and *right*, as required by jQuery Transit. This is inconsistent with our initial position setting in the renderLetters() function, where we have the following statements:

```
$("#"+id).css({"left":leftLocation+"px"});
$("#"+id).css({"top":topLocation+"px"});
```

We must change this to:

```
$("#"+id).css({"x":leftLocation+"px"});
$("#"+id).css({"y":topLocation+"px"});
```

Note that since jQuery Transit defines its own **css()** method, which overwrites jQuery's css() method. I have also renamed the variables leftLocation, and topLocation to **xLocation**, and **yLocation**, respectively. Finally, we call the animateLetters() function from our $("document").ready() function.

At this current location, the application appears like this.

So, we have animated the letters once. Next, we need to create continuous animation by repeatedly calling the animateLetters() function. Since JavaScript is single threaded, we cannot use a blocking loop, such as a for-loop, to achieve our continuous animation. Instead, we will use another technique: the **setInterval()** function.

The setInterval() function will repeated execute a block of code, after delay for a specified time. During the delay time, other code can be executed, making this a non-blocking solution. Hence, we can call our animation function like this:

```
setInterval(function(){animateLetters(letterArray);},12000);
```

The above function will wait 12 seconds between repeated calls to the **animateLetters()** function. To view the project at the current stage, click here:

At this stage, our animation is complete. Notice how the letters are animated continuously within the bounds of the container div. Our next step is to **add touch interactions**, and build the associated game logic.

Adding Touch Interactions

Next, we will allow users to interact with our application via touching (the type of interaction used in the iPad). As mentioned in the original project description, users should be able to drag and drop the letters onto the droppable area. Accordingly, we will first attempt to implement **drag and drop** for the letters.

We begin with dragging. When a user touches a letter, and moves his finger, we want the letter to follow his finger. We will accomplish this using an event based solution; that is, we look for the following touch events:

- **touchstart**: an event fired when the user starts touching.
- **touchmove**: an event fired when the user moves while touching

When these events are fired, we have predefined handler functions which will move the letter along with the user's finger.

We start with the handler for touchstart:

```
        //a variable to hold x and y values of touched location
      var offset = null;

    //function to handle a touch start
      var start = function(e) {
       //prevent default handling
        e.preventDefault();
        var id = $(this).attr("id");

        //stop the jquery transit transition and set the position
             var matrix = $("#"+id).css("-webkit-transform");
             var values = matrix.split('(')[1].split(')')[0].split(',');
             this.style['WebkitTransition'] = null;
             $("#"+id).css({"x":values[4]+"px"});
              $("#"+id).css({"y":values[5]+"px"});

           //find the x and y location at which you clicked on
        var orig = e.originalEvent;
        var pos = $(this).position();
        //set the offset variable to the respective locations
        offset = {
          x: orig.changedTouches[0].pageX - pos.left,
          y: orig.changedTouches[0].pageY - pos.top
        };
      };
```

In the above code, we first declare a variable offset, which will hold the x and y locations of the letter when it was first touched. Next, we define a function **start()**, which will handle a touchstart event. In this function, we first stop the CSS Transition (as shown before in the Adding Animation section of this guide). Then, we set the value of offset to an array containing the x and y values of the touched location. This will be useful information for when the user drags his finger.

When dragging, the touchmove even will fire. We handle it with:

```
var move = function(e) {
     //
  e.preventDefault();
  var orig = e.originalEvent;
  $(this).css({
    y: orig.changedTouches[0].pageY - offset.y,
    x: orig.changedTouches[0].pageX - offset.x
  });
};
```

The above function, **move()**, will be called continuously as the user drags his finger. Essentially, we want the function to compute the location of the users finger, and then set the letter to that position. As you can see above, this is exactly what the function does.

Lastly, we need to actually bind the events to the functions that we have defined above. The process for doing this is quite simple:

```
$(this).on("touchstart", start);
$(this).on("touchmove", move);
```

We simply use jQuery's **on()** method, which takes parameters for an event, and a handler. We have set the appropriate handler for each event, as you can see.

Finally, in order to make our design neater, we will wrap the above code in a prototype attribute, called **draggable**, and place it in a file called **dragDrop.js** (do not forget to add a reference to this in your index.html file). The code for draggable is shown below:

```
$.fn.draggable = function() {
          //a variable to
        var offset = null;

             //function to handle a touch start
        var start = function(e) {
         //prevent default handling
          e.preventDefault();
          var id = $(this).attr("id");

          //stop the jquery transit transition and set the position
               var matrix = $("#"+id).css("-webkit-transform");
               var values = matrix.split('(')[1].split(')')[0].split(',');
               this.style['WebkitTransition'] = null;
               $("#"+id).css({"x":values[4]+"px"});
            $("#"+id).css({"y":values[5]+"px"});

            //find the x and y location at which you clicked on
          var orig = e.originalEvent;
          var pos = $(this).position();
          //set the offset variable to the respective locations
          offset = {
            x: orig.changedTouches[0].pageX - pos.left,
            y: orig.changedTouches[0].pageY - pos.top
          };

        };

        var move = function(e) {
          e.preventDefault();
          var orig = e.originalEvent;
          $(this).css({
            y: orig.changedTouches[0].pageY - offset.y,
            x: orig.changedTouches[0].pageX - offset.x
          });
        };

        $(this).on("touchstart", start);
        $(this).on("touchmove", move);
};
```

Now in our **anim.js**, we can simply define a function called **addInteractions()**:

```
//this function will make each letter draggable
function addInteractions(letterArray){
       for(var i = 0; i < letterArray.length; i++){
             //get current letter
             var letter = letterArray[i];
             //make it draggable
             $("#"+letter).draggable();
       }
}
```

Here, we are simply iterating through the letterArray, and giving each letter the draggable functionality we defined in **dragDrop.js**.

To view the application at the current stage, click here (view only on iPad, as we have implanted touch interactions). As you can see, the user is now able to drag the letters around using his finger.

However, if viewed carefully, it is clear that there are a few minor issues with our current implementation:

- Sometimes, a drag is mistaken for a scroll, and causes the container to move down.
- A multi-touch gesture can be mistaken for a "pinch," which is used for zooming.

Ideally, we would like to disable both the scrolling, and pinching gestures for our application. This will allow the user to have smoother interactions with our application. After researching online, I have found the following possible solutions:

To disable zooming, we simply add the following tag in the head of our **index.html** file:

```
<meta name="viewport" content="width=device-width, maximum-scale=1.0, user-
scalable=no" />
```

This essentially states that the viewport, or screen, is not scalable. This will prevent zooming.

To disable scrolling, we first define the following javascript function, and place it in the head of **index.html**:

```
<script type = "text/javascript">
	function blockMove() {
		event.preventDefault() ;
	}
</script>
```

Then, we call this function when the body is moved. To do this, simply add the blockMove() function as a handler to the ontouchmove event in our html file, as shown:

```
<body ontouchmove="blockMove()">
	<div id = "container">
		<div id= "droppable"></div>
	</div>
</body>
```

To view the project at this stage, click here. As you can see, the application no longer scrolls or zooms. At this point, we have completed a basic implementation of touch interactions in our application.

Next, we will focus on implementing the game logic of our application.

Implementing Game Logic

Up until now, we have implemented most of the basic functionality of our application. However, for our application to actually behave as a "game," we need to set up the game logic.

The most basic aspect of our game is when the user drags a letter to the droppable div. We need to be notified of this event, so that we can handle it properly. Hence, we need to define an event that will be fired when a letter is dropped onto the droppable div.

We implement this here, in our **dragDrop.js** file's **droppable** function:

```
// create an event to signal a drop over the droppable
            var dropEvt = document.createEvent('Event');
            // define that the event name is `build`
            dropEvt.initEvent("dropped", true, true);

        //check to see if its over the droppable container
        //if so, fire an event
        var stop = function(e){
            var orig = e.originalEvent;
            var dragLocation =
            {
                  x: orig.changedTouches[0].pageX,
            y: orig.changedTouches[0].pageY
            }
                  var dropLocation = $("#droppable").position();
                  var width = $("#droppable").width();
            var height = $("#droppable").height();
            //is the letter over the droppable
                  if(
                        (dragLocation.x > dropLocation.left)&&
                        (dragLocation.x < (dropLocation.left + width))&&
                        (dragLocation.y > dropLocation.top)&&
                        (dragLocation.y < (dropLocation.top + height)))
                        {
                              //if so, fire the dropped event
                              this.dispatchEvent(dropEvt);
                        }
        };
```

Above, we first create a custom event, called "dropped." We will fire this when a letter is dropped onto the droppable div. Then, we define a function called **stop()**. This function checks to see whether or not the any part of the letter's position is on the droppable div. If it is, we fire the "dropped" event that we had created.

We want our **stop()** function to be called whenever the user stops dragging. Hence, we will bind it to the **touchend** event, as shown below:

```
$(this).on("touchend", stop);
```

At this point, our application will fire a “dropped” event whenever a letter is dropped onto the droppable div. The next logical step is to handle this “dropped” event. This is where we define most of our game logic.

I have done this in the following way.

```
	var letterIndex = 0;
	var expectedLetter = currentWord[letterIndex];
	document.addEventListener('dropped', function(event) {
		// e.target matches the elem from above
		var el = event.srcElement;
		//if the letter matches the next expected letter,
		//remove it from the screen, and change expectedLetter to
		//the next letter in the word.
		if ($(el).text() == expectedLetter) {
			$(el).remove();
			letterIndex++;
			expectedLetter = currentWord[letterIndex];
		}
		//if all of the letters have been dropped,
		//and the word has been spelled, go to the next word.
		if (letterIndex == currentWord.length) {
			wordIndex++;
			//if all the words have been spelled, start again
			if (wordIndex == wordArray.length) {
				wordIndex = 0;
			}
			//get the next word in the array
			currentWord = wordArray[wordIndex];
			$("#droppable").text(currentWord);

			//start from the first letter of the word.
			letterIndex = 0;
			//set expected letter
			expectedLetter = currentWord[letterIndex];

			//put all of the letters in the word on the screen again.
			letterArray = renderLetters(currentWord);
			//add touch interactions
			addInteractions(letterArray);
			//start the animation again
			animateLetters(letterArray);
			//animate repeatedly
			setInterval(function() {
				animateLetters(letterArray);
			}, 12000);
		}
	}, false);
```

Note, the above block of code is placed in the **anim.js** file, after the calls to the animation function.

In the code above, we first create a variable **letterIndex**, which serves as an index for the expected letter of our current word. We also create a variable **expectedLetter**, which holds the next expected letter. Then, we add an event listener to the document node for our "dropped" event (which we fire from dragDrop.js). In the handler, we first get the source element for the event (in other words, the letter that was dropped). If the letter matches the expected letter, then we remove it from the container, and update our indexes point to the next expected letter of the word. If the user spells the entire word(occurs when length of letterIndex equals the word length), we need to obtain the next word from the word array, render its letters to the container, animate its letters, and add touch interactions to it letters. We also need to update our droppable div to display the current word. If at any point the user has correctly spelled all of the words in the game, then we simply restart the game (by setting wordIndex to 0).

This takes care of nearly all of our game logic. However, for the sake of maintaining good coding style, we will note that we have some used some repeated code above. Particularly, the sequence:

```
	var letterArray = renderLetters(currentWord);

	//add touch interactions
	addInteractions(letterArray);

	//animate the letters
	animateLetters(letterArray);
	//animate repeatedly
	setInterval(function() {
		animateLetters(letterArray);
	}, 12000);
```

To minimize repeated code, we will simply create a new function, **setupWord()**, that will make the above calls. The function is shown below:

```
function setupWord(currentWord){
	//add letters to the screen, and return an array of the
letter ids
	var letterArray = renderLetters(currentWord);

	//add touch interactions
	addInteractions(letterArray);

	//animate the letters
	animateLetters(letterArray);
	//animate repeatedly
	setInterval(function() {
		animateLetters(letterArray);
```

By replacing the repeated code with the above function, we will make our code much cleaner, and easier to understand.

At this point, we have completed implementing our game logic, and along with that, our application. Just to ensure this, let us revisit our application requirements:

1. ~~Add a location where the letters will be dragged to (droppable location)~~
2. ~~Add a word in the droppable location, and its letters onto the screen.~~
3. ~~Animate the letters~~
4. ~~Add the touch interactions~~
5. ~~Implement the game logic~~

As you can see, we have finished implemented all of our requirements. The final version of the application can be seen here (iPad only).