Introducción

Desde el último Lab de videojuego hemos aprendido CSS muy superficialmente, pero en profundidad hemos visto cómo modificar las propiedades CSS de los elementos y esto nos va a permitir dar a nuestro videojuego un salto cualitativo muy grande, ya que gracias a poder modificar los atributos visuales de los elementos vamos a poder dejar el juego prácticamente finalizado.

En este Lab vamos a dotar a nuestro videojuego de interactividad, es decir, vamos a convertirlo en un juego.

Igual que hemos hecho hasta ahora, haced una copia de seguridad del proyecto en el estado actual que tengáis para que si en el desarrollo de este lab vuestro proyecto se estropea, siempre podáis volver atrás con garantías y volver a empezar.

Paso 1: Colocando los elementos

Lo primero que debemos hacer es colocar todos los elementos en pantalla de forma que sean todos visibles, para ello simplemente asignamos los atributos de ancho y alto de nuestras letras a un valor específico, por ejemplo: 200px.

El objetivo es poder visualizar todas las letras en pantalla de una sola vez, sin necesidad de hacer 'scroll'.

Paso 2: Arrastrando un elemento

En este punto tenemos el código necesario para detectar cuál es el elemento sobre el que se ha hecho 'click' y lo marcábamos con un marco rojo. Vemos que la introducción de manejadores como métodos de una clase suele suponer la necesidad de utilizar closures y el problema de que luego no se pueden des-registrar al no existir una referencia a la closure. Si quisiéramos arreglar esta situación entonces añadimos mucho código innecesario a nuestras clases que se convertirán en más difíciles de leer y mantener, por tanto, lo que se suele hacer en estos casos en crear funciones de utilidad que utilicen closures internamente para manejar ciertas interacciones.

Este va a ser el caso del manejo del arrastre de cada objeto. Lo primero que haremos será crear un archivo utils/drag.js que contendrá el código de nuestra función que se encargará de manejar el efecto de arrastre de las letras.

```
function drag(element, evt) {
    var scrollX = window.pageXOffset;
    var scrollY = window.pageYOffset;

    var startX = evt.clientX + scrollX;
    var startY = evt.clientY + scrollY;

    var origX = element.offsetLeft;
    var origY = element.offsetTop;

    var deltaX = startX - origX;
    var deltaY = startY - origY;

    document.addEventListener("mousemove", moveHandler, true);
    document.addEventListener("mouseup", upHandler, true);

    evt.preventDefault();

    function moveHandler(e) {
        var scrollX = window.pageXOffset;
        var scrollY = window.pageYOffset;

        element.style.left = (e.clientX + scrollX - deltaX) + "px";
        element.style.top = (e.clientY + scrollY - deltaY) + "px";

        e.stopPropagation();
    }

    function upHandler(e) {
        document.removeEventListener("mousemove", moveHandler, true);
        document.removeEventListener("mouseup", upHandler, true);
        e.stopPropagation();
    }
}
```

Este es el código del archivo drag.js. Simplemente define una función drag que recibe un elemento que será el que va a ser arrastrado y un evento.

Lo que hacemos es calcular, usando las propiedades geométricas de los elementos HTML la distancia del puntero del ratón a la esquina superior izquierda de la imagen donde se ha pulsado, y guardar esa distancia como delta. A continuación se registran manejadores capturadores para los eventos mousemove y mouseup, de esta manera, recibiremos cualquiera de estos eventos antes que cualquier manejador normal. Esto lo necesitamos para que, mientras el objeto está siendo arrastrado, sea el único que reciba estos eventos del ratón.
El manejador de movimiento simplemente hace que el elemento se mueva con el ratón, y el otro des-registra los capturadores para que, una vez se suelte el objeto, el programa vuelva al estado inicial.

A continuación debemos utilizar esta función y para asignar a cada clase la responsabilidad que le corresponde, desde mi punto de vista, la responsabilidad de detectar el toque de ratón en cada imagen pertenece a la propia imagen, además, así nos ahorramos la lógica de comprobar si el elemento clicado era o no una letra. Por tanto borramos toda la lógica asociada a la detección del click en el tablero y la pasamos a la letra de la siguiente manera:

```
LETTERS.Letter = function(source, index) {
  this.el = document.createElement('img');
  this.el.style.width = "200px";
  this.el.style.height = "200px";
  this.el.style.position = "absolute";
  this.el.style.top = "0px";
  this.el.style.left = parseInt(this.el.style.width) * index + "px";

  this.el.src = source;

  this.el.addEventListener('mousedown', clickClosure, false);

  var self = this;
  function clickClosure(evt) {
      self.clicked(evt);
  }
};

LETTERS.Letter.prototype = {
  el: null,   /** The HTML Element */

  clicked: function(evt) {
    drag(this.el, evt);
  }
};
```

Hemos hecho dos cambios importantes, el primero es colocar los elementos usando posición CSS absoluta, para permitir así que la función pueda moverlos, para esto necesitamos que el constructor sepa en qué posición se va a colocar su elemento. Deberéis, por tanto, modificar el bucle que crea estos elementos en el tablero para que le indique su posición.

Además vemos que hemos trasladado la detección del click del ratón a esta clase, ya que, como hemos dicho antes, debe ser la encargada del manejo de este evento. En el manejo de la bajada del ratón, lo único que hacemos es utilizar nuestra función de utilidad y ella se encargará del resto.

Por si hay alguna duda, el código del tablero quedaría así:

```
LETTERS.Board = function(htmlId) {
        var div = document.getElementById('#' + htmlId);
        if (!div) {
            div = document.createElement('div');
            div.id = htmlId;
            document.body.appendChild(div);
        }
        this.el = div;
        this.letters = [];
    };

    LETTERS.Board.prototype = {
        el: null,    /* HTML Element */
        letters: null,

        loadLetters: function(images) {
            for (var i = 0 ; i < images.length ; ++i) {
                var img = images[i];
                var letter = new LETTERS.Letter(img, i);
```

```
                this.letters.push(letter);
                this.el.appendChild(letter.el);
            }
        }
    };
```

Como vemos las modificaciones efectuadas son el cambio del bucle para indicarle al constructor Letter la posición en que debe colocar la letra, y hemos borrado todo el código de manejo del evento click que teníamos antes.

Si ejecutamos, podremos arrastrar por la pantalla las letras sin restricción alguna.

Paso 2: Añadir semitransparencia y eliminar efecto de "pasar por debajo" de la ficha en movimiento.

En este momento, si probamos a mover la letra 'A' sobre el resto de las letras, podremos comprobar que en realidad pasa por debajo y este efecto no es el que queremos dar a nuestro usuario. Debemos, por tanto asegurarnos de que la ficha que está en movimiento siempre está más arriba que las demás, como si la hubiéramos "levantado".

Esto lo conseguiremos con la propiedad zIndex, que controla el orden de apilamiento de las fichas.

Lo primero que vamos a hacer es empezar por definir dos constantes, que serán los valores de zIndex que las fichas van a tener cuando estén siendo arrastradas y cuando estén quietas. Modificamos el core.js para añadir lo siguiente:

```
var LETTERS = LETTERS|| {};

(function(){
  Object.defineProperty(LETTERS, "IDLE_Z_INDEX", { value: 0 });
  Object.defineProperty(LETTERS, "DRAGGING_Z_INDEX", { value: 1 });
})();
```

Si recordamos los apartados en los que hablábamos de los atributos de las propiedades de un objeto recordaremos que en JS se podían definir constantes reales utilizando los atributos y haciendo que la propiedad fuera no modificable y no enumerable. Definiendo las propiedades con Object.defineProperty, todo atributo no especificado es, por defecto false, así que, conseguimos nuestro objetivo.

Ahora queda utilizar estos valores. Aplicaremos IDLE_Z_INDEX a todas las fichas por defecto y DRAGGING_Z_INDEX a la ficha cuando sea seleccionada para moverse, asignándole de nuevo IDLE_Z_INDEX cuando es "soltada". Para esto, modificamos la clase letter.js con lo siguiente:

```
LETTERS.Letter = function(source, index) {
        ...
        this.el.style.zIndex = LETTERS.IDLE_Z_INDEX;
 };

 LETTERS.Letter.prototype = {
```

```
        el: null,   /** The HTML Element */

        clicked: function(evt) {
            this.el.style.zIndex = LETTERS.DRAGGING_Z_INDEX;
            var self = this;
            drag(this.el, evt, function() {
                self.released(evt);
                });
        },

        released: function(evt) {
            this.el.style.zIndex = LETTERS.IDLE_Z_INDEX;
        }
    };
```

Hemos añadido una sentencia en el constructor para que ponga IDLE_Z_INDEX a todas las fichas por defecto. En el manejador del click será donde pongamos DRAGGING_Z_INDEX, y fijaros que añadimos un nuevo atributo a nuestra clase drag, que será una función que queremos que ejecute cuando el objeto sea "soltado". En esta función volveremos a poner IDLE_Z_INDEX a la ficha.

Como hemos añadido un nuevo argumento a la función drag, debemos modificarla en consecuencia de esta manera:

```
function drag(element, evt, releasedHandler) {
  function upHandler(e) {
    ...
    releasedHandler(e);
  }
}
```

Añadimos el nuevo argumento a la función, y como última sentencia del manejador upHandler, invocamos la función pasándole el argumento.

Ejecutando ahora nuestra aplicación, veremos que las fichas que tenemos "cogidas" siempre pasan por encima del resto.

¿Cómo haríais ahora para que la ficha que estamos arrastrando sea semi-transparente? Dado que tanto el zIndex como la opacidad son modificadas siempre a la vez, como fruto de que estemos arrastrando o soltemos alguna ficha, se deberían definir clases de estilos y aplicarlos.

Creamos un nuevo archivo css/styles.css y lo enlazamos en la cabecera HTML.

```
.letter {
    width: 200px;
    height: 200px;
    position: absolute;
    top: 0;
    left: 0;
    z-index: 0;
}

.dragging {
    z-index: 1 !important;
```

```
    opacity: .75;
}
```

Estos son nuestros estilos definidos. Definimos dos clases, una para todas las letras y otra que se activará en una letra solo cuando esté siendo arrastrada.

Tendremos, por tanto, que modificar el código de la clase letter.js para hacer uso de estos estilos, y además, simplificaremos el código del constructor.

```
LETTERS.Letter = function(source, index) {
  this.el = document.createElement('img');
  this.el.src = source;

  this.el.classList.add('letter');
  var css =  window.getComputedStyle(this.el, null);
  this.el.style.left = parseInt(css.width) * index + "px";

  this.el.addEventListener('mousedown', clickClosure, false);

  var self = this;
  function clickClosure(evt) {
    self.clicked(evt);
  }
};

LETTERS.Letter.prototype = {
  el: null,   /** The HTML Element */

  clicked: function(evt) {
    this.el.classList.add('dragging');
    var self = this;
    drag(this.el, evt, function() {
      self.released(evt);
    });
  },

  released: function(evt) {
    this.el.classList.remove('dragging');
  }
};
```

Con esto vemos que el código de la clase letter.js queda mucho más corto. Si probamos la aplicación, efectivamente vemos que cuando estamos arrastrando una ficha, esta toma cierta transparencia.

Aquí finalizamos este Lab. Como siempre, no os quedéis con ninguna duda. Preguntadlas todas.