CONTENIDOS

CONTENIDOS 1

1. World 2

1.1 Configuración del World 2

1.2 Configuración de la cámara 2

2. Groups 3

3. Physics 5

3.1 Incorporar física al juego 5

3.2 Incorporar física a un Sprite 6

3.3 Colisiones 7

1. World

En este apartado veremos cómo realizar el seguimiento con la cámara a través del escenario, para obtener el desplazamiento del fondo.

Para ello vamos a utilizar la clase **world**, cuya documentación oficial se encuentra disponible en:

docs.phaser.io/Phaser.World.html

1.1 Configuración del World

Se considera el *world* al entorno donde se encuentran todos los actores del videojuego. Un videojuego posee un solo *world* y por defecto se crea del mismo tamaño que el escenario. Como veremos en breve, el *world* tiene relación directa con el manejo de la cámara.

Por defecto, al invocar a la clase principal del juego *Game* se estará creando un *world* del mismo tamaño:

```
var game = new Phaser.Game(800, 600, Phaser.AUTO, '', {
 preload: preload,
 create: create,
 update : update });
```

Para configurar el *world* con otro tamaño se debe utilizar el método ***setBounds(x, y, width, height)***.

El método ***setBounds*** de la clase ***world*** posee los siguientes argumentos:

- x: coordenada x de la esquina superior izquierda del world.
- y: coordenada y de la esquina superior izquierda del world.
- Width: nuevo ancho del world. No debe ser menor que Game.width.
- Height: nuevo alto del world. No debe ser menor que Game.height.

Aquí se configura el tamaño del *world* a 1200 por 600, es decir que se aumenta en 400 su tamaño en *x*:

```
game.world.setBounds(0, 0, 1200, 600);
```

1.2 Configuración de la cámara

Una vez configurado el *world* por medio de *setBounds*, se puede hacer un desplazamiento de la cámara usando el método ***camera*** de la clase ***Game***.

Si se busca obtener un desplazamiento horizontal de la cámara, se debe alterar su coordenada *x* del siguiente modo:

```
game.camera.x += velCamaraX;
```

Donde en la variable *velCamaraX* se tiene almacenado el incremento horizontal de la cámara.

Si se desea fijar un determinado objeto de la escena a la cámara, para que no se desplace con el resto de los objetos, debemos modificar la propiedad ***fixedToCamera*** del mismo. Si fuese un videojuego en 3D, su equivalente sería fijar el arma del personaje a la cámara mientras éste se desplaza.

```
Logo = game.add.sprite(0, 0, 'logo');

// Fija el sprite a la camara, es decir que el logo la acompañara
cuando la desplacemos con game.camera.x

Logo.fixedToCamera = true;
```

Con *fixedToCamera* fijamos el sptrite en la esquina superior izquierda del *world*. Si queremos que el actor se encuentre en una posición distinta, necesitamos usar método ***setTo(OffsetX, OffsetY)*** de la clase ***cameraOffset***:

```
// Luego de fijar el sprite a la camara, lo desplazamos respecto a
la esquina superior izquierda

Logo.cameraOffset.setTo(20, 10);
```

En el primer ejemplo de esta sección se realizó el desplazamiento horizontal de la cámara, alterando su coordenada *x* por medio de la variable *velCamaraX*. Haciendo el desplazamiento esa manera se tiene el inconveniente que si la velocidad de avance del personaje no se encuentra sincronizada con la que posee la cámara, el personaje quedará atrasado o adelantado con respecto al avance de la cámara. Si lo que buscamos es que la cámara siga al personaje sin tener variaciones, debemos usar el método ***follow***. En el siguiente fragmento de código estamos definiendo que la cámara siga al sprite:

```
Patricio = game.add.sprite(200, 200, 'atlas');

//la camara sigue al sprite

game.camera.follow(Patricio);
```

Con ello la cámara y el personaje estarán sincronizados.

En los ejemplos **00_Camera_con_fisica** , **00_Camera_sin_fisica** y **00_Camera_con_fisica_follow** se pueden observar en detalle las implementaciones del *world* y la cámara.

2. Groups

En unidades anteriores aprendimos a detectar colisiones entre objetos mediante el método *intersects()*. En el ejemplo visto el personaje podía colisionar con un item, pero en el caso de que hubiese muchísimos items deberíamos tener presente la eficiencia en el uso de recursos, ya que estaríamos realizando una llamada a *intersects* para cada uno de ellos.

Para atender estas cuestiones Phaser nos facilita la clase ***Group***, que representa un grupo de actores. Es decir, un grupo es una clase especial de actor que puede contener dentro de sí mismo a otros actores, incluso otros grupos.

Los objetos del juego pueden agregarse a un grupo de la misma forma que a una escena. A su vez, un grupo puede ser agregado a la escena como cualquier otro actor.

Para manejar grupos es necesario utilizar el método ***add*** la clase principal del juego *Game*. Por ejemplo, la siguiente sintaxis agrega el grupo *enemies*:

```
enemies = game.add.group();
```

Para crear una nueva instancia de Phaser.Sprite y agregarla al grupo se debe emplear el método ***create()*** :

create(x, y, key)

El método ***create*** de la clase ***group*** posee los siguientes argumentos:

- x: coordenada x del nuevo Sprite, en relación a Group.x point.
- y: coordenada y del nuevo Sprite, en relación a Group.y point.
- Key: el ID de la imagen asociada al sptrite.

```
game.load.image('invader',
'../../assets/games/invaders/invader.png');

enemies.create(Math.random() * 800, 120 + Math.random() * 200,
'invader');
```

También hubiese sido possible agregar un sprite exitente al grupo *enemies* recién creado:

```
enemies.add(player);
```

Si quisiéramos aplicarle un comportamiento o defirnirle un atributo a algún elemento del grupo podríamos podemos obtener un elemento particular mediante la función getAt, que recibe como parámetro un índice que indica la posición del elemento en el grupo. Por ejemplo:

```
enemies.getAt(i);
```

Donde *i* es el índice del emento.

Un ejemplo del uso de getAt se encuentra en el directorio **03_getAt**.

Si quisiéramos aplicar una propiedad o atributo a todos los elementos del grupo, podemos utilizar las funciones setAll y callAll.

setAll sirve para modificar un atributo y su sintaxis es la siguiente:

Grupo.setAll (**key**, **value**)

Donde:

- key: Es el atributo a modificar
- value: Es el valor que le vamos a asignar

De esta manera, si quisiéramos que todos los elementos del grupo tuvieran habilitados los eventos, haríamos:

```
enemies.setAll('inputEnabled', true);
```

De la misma manera, si quisiéramos definirle un método a todos los elementos del grupo, podríamos utilizar la función callAll, cuya sinyaxis es la siguiente:

Grupo.callAll (**method**, **context**, **parameter**)

Donde:

- method: Es el método a modificar

- context: Es el contexto del método a modificar

- parameter: Son los parámetros que va a recibir el método (separados por coma).

Por ejemplo, continuando con el ejemplo, si quisiéramos que todos nuestros objetos del grupo tuvieran habilitado el drag, deberíamos poner:

```
enemies.callAll('input.enableDrag', 'input',true);
```

Input.enableDrag es el método a modificar, *input* es la clase (o contexto) al cual vamos a acceder ya que *enableDrag* se encuentra en *input*. Finalmente *true* es el parámetro que vamos a modificar para habilitar el enableDrag.

En el ejemplo **02_set_all_call_all** está implementado el drag para todo el grupo mediante la utilización de las funciones *setAll* y *callAll*. En el ejemplo **02_group_atlas** se utiliza la función *setAll* y *callAll* para definirle una animación a todos los elementos del grupo.

3. Physics

Demás está explicar las ventajas de contar con un sistema de física en una herramienta para el desarrollo de nuestro juego. Phaser ofrece tres sistemas de física, la Arcade, la Ninja y la P2 (¡Con Box2d y Chipmunk en desarrollo!). La Arcade es una configuración simple, con colisiones únicamente de AABB, es decir que sólo evalúa solapamientos y no considera objetos no convexos, esta configuración es considerablemente barata y sirve para colisiones de alta velocidad. La configuración Ninja agrega cuerpos no convexos y rotaciones, obviamente es más costosa de procesar. Finalmente la P2 es un sistema completo que agrega uniones, resortes, polígonos y otras características, claramente es un sistema costoso, aunque aporta realismo a nuestro juego. Phaser no permite que un objeto tenga más de un sistema, sin embargo, permite tener diferentes sistemas actuando en una misma escena.

Para esta materia veremos algunos métodos de la física Arcade (que es la que está configurada en Phaser a menos que especifiquemos lo contrario). La clase de este sistema es Phaser.Physics.Arcade, y es una clase muy extensa por lo que sólo veremos los conceptos más importantes.

3.1 Incorporar física al juego

Claramente lo más interesante de incorporarle física a nuestro juego es la posibilidad de que los elementos reaccionen a la gravedad y se le pueda imprimir velocidad.

Para habilitar la física al juego debemos escribir la siguiente línea en el estado Create:

```
game.physics.startSystem(Phaser.Physics.ARCADE);
```

Con ello iniciamos el sistema de física Arcade. En realidad Phaser tiene este sistema habilitado por defecto, por lo que la línea podría omitirse, no así si quisiéramos utilizar otro sistema, de todos modos es una buena práctica habilitarlo por las dudas.

A continuación podremos fijar las variables de nuestro sistema, como la aceleración o la velocidad.

Vamos a imprimir, por ejemplo, una velocidad de 100 (pixeles por segundo) en el eje y.

```
game.physics.arcade.gravity.y = 100;
```

También podríamos haber definido una gravedad en el eje x. Notar que esta condición se aplica al mundo, es decir, al contexto Game, luego podremos definir particularmente otras características particulares a los elementos.

Por último tendremos que definir los objetos que va a ser afectados por la física.

```
game.physics.enable( elemento, Phaser.Physics.ARCADE);
```

Donde *elemento* puede ser un arreglo y contiene todos los actores que van a ser afectados por la gravedad. Pueden ser Sprites independientes o bien, grupos.

Un ejemplo de la aplicación de la gravedad se puede ver en el ejemplo **00_incorporar_fisica_game**

3.2 Incorporar física a un Sprite

Con la gravedad definida en el mundo, el sprite, respetará las condiciones que esta dicte, sin embargo, es posible definirle valores especiales o adicionales al mismo.

Definimos la física para el mundo como ya vimos:

```
game.physics.enable( elemento, Phaser.Physics.ARCADE);
game.physics.arcade.gravity.y = 100;
game.physics.enable( elemento, Phaser.Physics.ARCADE);
```

Ahora le daremos propiedades particulares al elemento, como por ejemplo gravedad adicional.

```
elemento.body.gravity.y = 200;
```

Con esto el *elemento* estará afectado por la gravedad del mundo y adicionalmente la gravedad propia. El objeto *body* es el que controla las respuestas generadas por el sistema físico. Del *body* podemos modificar manipular atributos como: la gravedad, la velocidad, la velocidad angular, la elasticidad, el rebote, en otros. Y manipular métodos que nos darán respuestas a las acciones sobre el mundo, como saber hay colisión contra otro objeto.

Para saber más sobre el objeto body, pueden consultar la documentación:

http://phaser.io/docs/Phaser.Physics.Arcade.Body.html

En el ejemplo **02_incorporar_fisica_sprite_bounce** se pueden ver dos objetos con gravedades diferentes, uno con la gravedad del mundo y otro con gravedad propia adicionada.

Aun así, a veces, es necesario tener objetos que puedan reaccionar a un objeto con física pero continuar estáticos. Para ello debemos darle al elemento 3 características, debemos decirle que esté fijo, que no respete la gravedad y que en todo momento su velocidad sea 0.

Para modificar esas características debemos escribir en el estado *Create*:

```
        //lo declaro inamovible
        elemento.body.immovable = true;

        //que no sea afectado por la gravedad del mundo
        elemento.body.allowGravity = false;
```

Demás está decir que previamente o posteriormente tenemos que habilitarle la física a este elemento, como lo hicimos anteriormente.

Ahora en el estado *Update* debemos decirle que la velocidad en cada instante cera cero.

```
elemento.body.velocity.setTo(0, 0);
```

Ahora el elemento estará fijo en la posición que le especifiquemos. Sin embargo aun no interactúa con los otros elementos, es decir, no colisiona con otros elementos.

En el ejemplo **03_incorporar_fisica_sprite_inamovible**

3.3 Colisiones

Ya sabemos cómo hacer que nuestros elementos respeten la física del mundo, ahora veremos cómo hacer para que interactúen entre ellos mediante colisiones.

Para definirle una colisión a un elemento debemos utilizar la función *collide* perteneciente al objeto *physics.arcade*. Esta función es general y sirve para comparar 2 elementos, los cuales pueden ser sprites contra sprites, sprites contra grupos, grupos contra grupos, sprite contra tilemap, o bien grupo contra tilemap. La sintaxis completa de esta función es la siguiente:

physics.arcade.collide (**object1**, **object2**, **collideCallback**, **processCallback**, **callbackContext**)

Donde:

- object1: El primer objeto para comparar la colisión, puede ser sprite, grupo o tilemap
- object2: El segundo objeto para comparar la colisión, puede ser sprite, grupo o tilemap
- collideCallback: La función que se ejecutará al realizarse la colisión. Opcional
- processCallback: La función que se ejecutará al instante previo de la colisión. Opcional
- callbackContext: El contexto en el cual se ejecutarán las funciones callback.

Ambas funciones callback reciben como parámetros los dos objetos. La función collideCallback se ejecuta cuando ocurre el solapamiento y la función processCallback se ejecuta en el instante previo a la colisión y se utiliza para obtener más datos acerca de la colisión, como la velocidad previa del objeto, etc.

Volviendo al último ejemplo, si quisiéramos definirle al objeto estático la interacción con los demás objetos, debemos definir en el estado Update la función collide, de la siguiente forma:

```
game.physics.arcade.collide(elemento, otro_objeto);
```

Donde *elemento* es nuestro actor estático y *otro_objeto* es algún otro actor, o bien un grupo o un Tilemap.

En el ejemplo **04_incorporar_fisica_sprite_inamovible_collision** puede verse un ejemplo de un objeto estático que reacciona a dos pelotas que pican en la pantalla.

Si queremos definir alguna función que se ejecute cuando colisionan los objetos debemos definir un callback, por ejemplo:

```
function update(){
  game.physics.arcade.collide(elemento,otro_objeto,chocaron,null,this);
     }

   function chocaron (ob1, ob2)
       …
       }
  }
```

En el ejemplo **05_colision_sprite_vs_sprite** se muestra la implementación de una función callback ante la colisión de dos sprites.

En el ejemplo **06_colision_sprite_vs_grupo** se muestra la implementación de la colisión entre un sprite y un grupo

Finalmente en el ejemplo **07_colision_grupo_vs_grupo** se muestra la implementación de una función callback ante la colisión de dos grupos.