**Diseño y desarrollo de juegos de rollo *Shoot’em up* (se viene mojonazo).**

Para el que eso de los años 90 les suene a chino, un *Shoot’em up* es un tipo de videojuego caracterizado por el uso de animaciones 2D, enemigos predefinidos y mecánicas muy simples a la hora de jugar. Son los típicos juegos de puntuaciones y esas cosas. A continuación vamos a ver cómo se pueden hacer algunos títulos simples que sirvan de referencia en Phaser.

1. **Introducción.**

En la anterior práctica hemos visto algunos conceptos fundamentales de *Phaser*. A continuación veremos unos aspectos más cercanos a la creación de juegos en específico. Para ello, hay que tener claro el concepto de *estado*, así como el tipo de *sistema de físicas* a emplear.

**1.1. Estados de Phaser.**

El concepto de estado es prácticamente el núcleo de *Phaser*, pues es el que permite un mayor manejo de las fases conceptuales que forman un proyecto (títulos, créditos, etc).

Los estados permiten cortar el código en diferentes secciones para que puedan ser gestionados de mejor manera.

En la anterior práctica vimos los típicos (aquellos que ya vienen predefinidos en *Phaser*): *create()*, *preload()* y *update()*. A continuación veremos de forma más acentuada los usos de estos estados.

**1.2. Físicas de *Phaser*.**

Las físicas sirven para simular el sistema de físicas del mundo real (o al menos crear uno con sentido dentro del universo del juego): hablamos de la gravedad, de la aceleración, etc.

En *Phaser* hay tres tipos de físicas, las cuales tienen sus ventajas y desventajas frente al resto:

* *Física Arcade* o *Arcade Physics*: Es la más usada y en consecuencia las mejor documentada. Es la más veloz, pero la más simple. Eso provoca un serio problema con las colisiones, pues se manejan mediante cajas (rectángulos) en los objetos, de tal forma que eso nos la puede liar en caso de que hagamos algo que no guarda unas proporciones lógicas (en plan el puto Doraemon, que con el cabezón que tiene con respecto a su cuerpo pues dejaría un hueco en blanco quedaría pie a fallos visuales evidentes a la hora de chocarse con otros objetos).
* *Física Ninja o Ninja Physics*: Originalmente estaban destinadas a *Flash*, aunque se portearon a *Phaser*. Tienen más complejidad que las de Arcade y las colisiones las manejan con pendientes.
* *Físicas P2* o *P2 Physics*: El más complejo y realista, pues se pueden hacer formas muy extrañas como péndulos y otras movidas de croatas y serbios. A cambio es el que requiere mayor coste computacional.

En general vamos a gastar las físicas *Arcade* porque son las más simples y tal. Podemos observar algunos ejemplos guays para copiar en [**este link**](https://phaser.io/examples/v2/category/arcade-physics).

A continuación trabajaremos con un ejemplo clásico: el juego de los marcianitos, proporcionado por Amengual en la propia práctica.

**2. Los marcianitos.**

**2.1. Estructura *index.html* y *main.js*.**

El *file* de nombre *init.js* contiene la declaración del *initState*, que es la primera pantalla del juego /la del título, vamos). Básicamente contiene información acerca del juego, a modo de presentación (da pie al *playState*).

Ahora veamos algo muy interesante para cambiar de pantalla: los métodos *checkWorldBound*s y *onOutOfBounds*. Vamos a plantear el siguiente escenario:

Tenemos un evento llamado *btnStart* el cual hará que un monigote se mueva por pantalla.

¿Cómo hacemos para que suceda una determinada acción cuando se haya salido de la pantalla? Pues mediante los ya citados métodos **checkWorldBounds** y **onOutOfBounds**.

El primero de todos lo que hace es comprobar si el objeto está dentro de los límites que nosotros hemos establecido. Se define de la siguiente manera:

**btnStart.checkWorldBounds = true;**

Ese *true* lo que hace es iniciar el proceso de comprobación.

Luego está el segundo, que se define de la siguiente forma:

**btnStart.events.onOutOfBounds.add(***función***, this);**

Básicamente lo que hace esto tan bonito de acá es activar la función (el primer parámetro) en caso de que se salga por pantalla (eso ya lo comprueba él solito).

**3. El spacecraft: hacer que la cosa funcione.**

Básicamente acá haremos que funcione el jueguito de marras, activando todos los elementos correspondientes en el estado *playState*. Esta cosa está definida en el *file* *play.js*, a modo de objeto.

Primero empecemos por crear la nave. Para ello tocará crear, de forma global, dos variables:

**var craft;**

**const HUD\_HEIGHT = 50;**

Básicamente *HUD\_HEIGHT* es el tamaño del mundo en el que jugaremos, mientras que *craft* es la nave.

Luego, en la función **preload** cargamos la imagen asociada a *craft* de la siguiente manera:

**game.load.image('craft','img/craft.png');**

En la función **create** colocamos una llamada a un método que desarrollaremos más tarde:

**createCraft();**

Finalmente declaramos esa función:

**function createCraft() {**

**var x = game.world.centerX;**

**var y = game.world.height - HUD\_HEIGHT;**

**craft = game.add.sprite(x, y, 'craft');**

**craft.anchor.setTo(0.5, 0.5); };**

Básicamente lo que está haciendo la función es definir la posición y la muestra por pantalla del objeto.

**3.1. Eventos de teclado y ratón.**

Ahora tocará que el bicho se mueva. Para ello tendremos que definir los eventos relativos al teclado y al ratolí.

Primero tenemos que definir dos nuevas variables, que almacenarán, respectivamente, el cursor y la velocidad de desplazamiento:

**var cursors;**

**const CRAFT\_VELOCITY = 150;**

Luego, en el *create* colocamos una llamada a la siguiente función:

**createKeyControls();**

Esto se complementará, fuera del *create*, con la elaboración de dicha función,que será la que permita constituir los controles del objeto:

**function createKeyControls() {**

**cursors = game.input.keyboard.createCursorKeys();**

**game.physics.arcade.enable(craft);**

**};**

Atentos, que hemos colocado la activación de las físicas *arcade* para el objeto *craft* (nuestra nave).

En *update* colocamos la siguiente llamada:

**manageCraftMovements();**

Y por fuera la declaración de la misma, claro:

**function manageCraftMovements() {**

**craft.body.velocity.x = 0;**

**if (cursors.left.isDown || game.input.speed.x < 0)**

**craft.body.velocity.x = -CRAFT\_VELOCITY;**

**else if (cursors.right.isDown || game.input.speed.x > 0)**

**craft.body.velocity.x = CRAFT\_VELOCITY;**

**};**

Esta función básicamente es la encargada de aplicarle el movimiento (con su respectiva velocidad) a la nave.

**3.2. Cercando los límites del mundo.**

Porque claro, querremos limitar la zona de actuación del muñequín, dado que tal y como lo tenemos todo pues se sale de los márgenes.

Para ello habrá que colocar la siguiente línea dentro de la función *createCraft*:

**craft.body.collideWorldBounds = true;**

Previo a eso hay que habilitar la física de la misma manera que hemos visto en la función *createKeyControl()*, eh.

**3.3 Haciendo que el fondo se mueva.**

Como tener un cuadrado oscuro como fondo no lo llevamos bien, tocará colocar un fondo. Eso lo sabemos hacer (o al menos deberíamos), pues es tan simple como cargar una imagen con las dimensiones del recuadro del juego. Pero no nos motivemos. Primero creamos la variables *stars*, al inicio del todo:

**var stars;**

Luego en el *preload()* cargamos la imagen:

**game.load.image('stars','img/stars.png');**

Y en el *create()* la colocamos en su posición inicial:

**var w = game.world.width;**

**var h = game.world.height;**

**stars = game.add.tileSprite( 0, 0, w, h, 'stars');**

Como se puede observar, hemos escalado la imagen para que ocupe el cuadrado del fondo. Cabe destacar que esto lo hemos puesto ANTES de que inicialicemos la nave. Más que nada porque si no el fondo solaparía el otro *sprite* (ustedes no empiezan una tarta colocando la nata y luego la base, ¿no? Pues es lo mismo).

**4. Los láseres.**

Para trabajar con los láseres, primero tenemos que definir la variable correspondiente:

**var lasers;**

Luego, en el preload, cargamos la imagen correspondiente al disparo:

**game.load.image('laser','img/laser.png');**

Para luego hacer, en el create, una llamada a una nueva función:

**createLasers(40);**

Esa nueva función la definimos afuera de la siguiente manera:

**function createLasers(number) {**

**lasers = game.add.group();**

**lasers.enableBody = true;**

**lasers.createMultiple(number, 'laser');**

**lasers.callAll('events.onOutOfBounds.add', 'events.onOutOfBounds', resetMember);**

**lasers.callAll( 'anchor.setTo', 'anchor', 0.5, 1.0);**

**lasers.setAll('checkWorldBounds', true);**

**};**

Básicamente estamos escribiendo una función que será la encargada de crear la imagen y las propiedades del disparo de la nave.

Luego también declaramos la función *resetMember()*, que básicamente quitará de memoria un objeto determinado (ideal para los disparos, pues su utilidad de forma individual no pasa de una ocasión):

**function resetMember(item) {**

**item.kill();**

**};**

Esta función también la gastaremos más tarde para los OVNIs.

**4.1. El disparo de los lásers.**

Para ello primero debemos crear una variable donde almacenaremos todo lo referido a los disparos:

**var fireButton;**

¿Se acuerdan de *createKeyControls()*? Es una función que creamos para manejar todos los controles del juego. Hasta el momento teníamos solamente asociados aquellos botones referidos al movimiento. Ahorá le tocará a la barra espaciadora, que servirá como botón de disparo, metiéndole la siguiente línea al final de la función ya mencionada:

**fireButton = game.input.keyboard.addKey( Phaser.Keyboard.SPACEBAR);**

Y ahora volvamos a lo nuestro. En el *update()* hacemos una llamada a una nueva función:

**manageCraftShots();**

Y ahora, como era de esperar, la definimos:

**function manageCraftShots() {**

**if (fireButton.justDown || game.input.mousePointer. leftButton.justPressed(30))**

**fireLasers();**

**};**

Básicamente estamos indicando que, en caso de que se mantenga pulsado el botón de la barra espaciadora, tocará disparar (llamando a la función *fireLasers()*).

¿Cómo es *fireLasers()*? Pues así:

**function fireLasers() {**

**var lx = craft.x - 11;**

**var rx = craft.x + 12;**

**var y = craft.y - 10;**

**var vy = -500;**

**var laserLeft = shootLaser(lx, y, vy);**

**var laserRight = shootLaser(rx, y, vy);**

**};**

Esta función está calculando la posición del disparo en función del lugar donde está la nave.

A su vez está llamando a shootLaser, una función que (por fin) se encarga de efectuar el disparo:

**function shootLaser(x, y, vy) {**

**var shot = lasers.getFirstExists(false);**

**if (shot) {**

**shot.reset(x, y); shot.body.velocity.y = vy;**

**}**

**return shot; };**

**4.2. El sonido del disparo.**

Para efectuar el sonido del disparo, tocará primero hacer una variable:

**var soundLaser;**

En el *preload()* cargamos el audio en cuestión de la siguiente manera:

**game.load.audio('sndlaser', 'snd/laser.wav');**

En el *create()*, colocamos lo siguiente:

**createSounds();**

Esa llamada se complementa, por supuesto, con su respectiva función:

**function createSounds() {**

**soundLaser = game.add.audio('sndlaser');**

**}**

Finalmente, al final de la función *fireLasers()* colocamos la siguiente condición:

**if (laserLeft || laserRight)**

**soundLaser.play();**

**5. Poner en marcha los OVNIS (y eso).**

Ahora veremos como funcionan los *ovnis*.

Para ello, primero creamos una variable con la que podamos agrupar todo lo referido a los enemigos:

**var ovnis;**

Luego en el *preload()* cargamos la imagen que queremos que esté asociada al platillo volante:

**game.load.image('ovni', 'img/ovni.png');**

Y en el *create()* invocamos una función (como de costumbre):

**createOvnis(200);**

Esta función la definimos por separado del *create*:

**function createOvnis(number) {**

**ovnis = game.add.group();**

**ovnis.enableBody = true;**

**ovnis.createMultiple(number, 'ovni');**

**ovnis.callAll('events.onOutOfBounds.add', 'events.onOutOfBounds', resetMember);**

**ovnis.callAll( 'anchor.setTo', 'anchor', 0.5, 1.0);**

**ovnis.setAll('checkWorldBounds', true);**

**};**

Está definiendo las características básicas del ovni.

Ahora hagamos aparecer a los bicharracos (y que su aparición sea aleatoria, como Amengual manda).

De primeras definimos las siguientes variables (y una constante):

**const TIMER\_RHYTHM=0.1\*Phaser.Timer.SECOND;**

**var currentOvniProbability;**

**var currentOvniVelocity;**

A continuación expandimos la función *createOvnis()* para que contenga el evento que permitirá el movimiento del monigote:

**currentOvniProbability = 0.2;**

**currentOvniVelocity = 50;**

**game.time.events.loop( TIMER\_RHYTHM, activateOvni, this);**

Y ahora activamos el manejador:

**function activateOvni() {**

**if (Math.random() < currentOvniProbability) {**

**var ovni = ovnis.getFirstExists(false);**

**if (ovni) {**

**var gw = game.world.width;**

**var ow = ovni.body.width;**

**var w = gw - ow; var x = Math.floor(Math.random() \* w);**

**var z = ovni.body.width / 2 + x; ovni.reset(z, 0);**

**ovni.body.velocity.x = 0;**

**ovni.body.velocity.y = currentOvniVelocity;**

**}**

**}**

**}**

Atentos al random de esta función, que lo que está haciendo es colocar al objeto en el eje x de forma aleatoria.

**6. Colisiones.**

Ahora veamos cómo manejamos todo eso las colisiones entre objetos. Para ello tocará ponerle físicas tanto a los ovnis como a los disparos de la nave. Es por ello que en el *update()* colocamos lo siguiente:

**game.physics.arcade.overlap( lasers,ovnis,laserHitsOvni,null,this); game.physics.arcade.overlap( craft,ovnis,ovniHitsCraft,null,this);**

Básicamente está indicando que, en caso de choque, tocará realizar la acción indicada en la función que actúa como tercer parámetro.

Las dos funciones a las que se llaman son las siguientes:

**function laserHitsOvni(laser, ovni) {**

**ovni.kill();**

**laser.kill();**

**};**

**function ovniHitsCraft(craft, ovni) {**

**ovni.kill();**

**craft.kill();**

**};**

Se encargan de eliminar los objetos en pantalla.

**6.1. Animando una explosión.**

Porque claro, lo que tenemos ahora solamente quita de la pantalla al ovni y tal. ¿Cómo hacemos una explosión como Dios manda?

Inicialmente creamos la variable adecuada:

**var blasts;**

A continuación cargamos la imagen correspondiente a la explosión en el *preload()*:

**game.load.spritesheet( 'blast', 'img/blast.png', 128, 128);**

Luego en el *create* invocamos el siguiente método:

**createBlasts(30);**

Ahora pasemos a definir dos funciones claves para el funcionamiento de la explosión:

**function createBlasts(number) {**

**blasts = game.add.group();**

**blasts.createMultiple(number, 'blast');**

**blasts.forEach(setupBlast, this);**

**};**

**function setupBlast(blast) {**

**blast.anchor.x = 0.5;**

**blast.anchor.y = 0.5;**

**blast.animations.add('blast');**

**};**

Al final de la función *laserHitsOvni()* tocará colocar una nueva línea de código que actuarán de forma visible en caso de que haya una explosión:

**displayBlast(ovni);**

Finalmente, la función a la que referencian esa línea de código se coloca separado del resto:

**function displayBlast(ship) {**

**var blast = blasts.getFirstExists(false);**

**var x = ship.body.center.x;**

**var y = ship.body.center.y;**

**blast.reset(x, y);**

**blast.play('blast', 30, false, true);**

**};**

**6.2. El sonido de la explosión.**

De la misma manera que hemos hecho anteriormente con el sonido del disparo, el de la explosión se efectúa colocando, en un inicio, una nueva variable:

**var soundBlast;**

A continuación, en el *preload()*, cargamos el audio:

**game.load.audio('sndblast', 'snd/blast.wav');**

Después de eso, en la función *createSounds()* añadimos la siguiente línea de código:

**soundBlast = game.add.audio('sndblast');**

Y en *laserHitsOvni* hacemos que se ejecute el sonido, añadiendo la siguiente línea:

**soundBlast.play();**

**7. HUD.**

El *HUD* es, básicamente, todo lo que aparece arriba en plan puntuación, tiempo, nivel,etc. Vamos, un amasijo de información.

Para hacer un *HUD* básico, tenemos que hacer los siguientes pasos.

Como ya sabemos, cada elemento tendrá asociado una variable. Por eso declaramos las siguientes:

**var score;**

**var scoreText;**

**var level;**

**var levelText;**

**var lives;**

**var livesText;**

A continuación le damos valores numéricos a ciertas variables:

**score = 0;   
 level = 1;   
 lives = 3;**

En el *create()* escribimos una llamada a la siguiente función:

**createHUD();**

A su vez, esta función es:

**function createHUD() {**

**var scoreX = 5;**

**var levelX = game.world.width / 2;**

**var livesX = game.world.width - 5;**

**var allY = game.world.height - 25;**

**var styleHUD = {fontSize: '18px', fill: '#FFFFFF'};**

**scoreText = game.add.text( scoreX,allY,'Score: '+score,styleHUD);**

**levelText = game.add.text( levelX,allY,'Level: '+level,styleHUD); levelText.anchor.setTo(0.5, 0);**

**livesText = game.add.text( livesX,allY,'Lives: '+lives,styleHUD); livesText.anchor.setTo(1, 0);**

**};**

Tocará, además, añadir las siguientes líneas en la función *laserHitsOvni()*:

**score++;**

**scoreText.text = 'Score: '+score;**

También tocará bajar las vidas del jugador en caso de que se choque con el bicho. Añadiremos pues las siguientes líneas en la función *ovniHitsCraft()*:

**lives--;**

**livesText.text = 'Lives: '+lives;**

**7.1. Incrementar la dificultad del juego a través de los niveles.**

Para ello hay que variar ciertos valores dentro del juego.

De primeras debemos declarar una serie de constantes:

**const NUM\_LEVELS = 5;**

**const LEVEL\_OVNI\_PROBABILITY = [0.2, 0.4, 0.6, 0.8, 1.0];**

**const LEVEL\_OVNI\_VELOCITY = [50, 100, 150, 200, 250];**

**const HITS\_FOR\_LEVEL\_CHANGE = 50;**

Tocará cambiar los valores de probabilidad de aparición y de velocidad de los platillos volantes en la función *createOvnis()*:

**currentOvniProbability = LEVEL\_OVNI\_PROBABILITY[level-1];**

**currentOvniVelocity = LEVEL\_OVNI\_VELOCITY[level-1];**

Al final del código de *laserHitsOvni()* tocará anexar una nueva parte de código, referida al manejo de la actualización del nivel, probabilidad y velocidad del ovni:

**if (level < NUM\_LEVELS && score===level\*HITS\_FOR\_LEVEL\_CHANGE) {**

**level++;**

**levelText.text = 'Level: ' + level; currentOvniProbability = LEVEL\_OVNI\_PROBABILITY[level-1]; currentOvniVelocity = LEVEL\_OVNI\_VELOCITY[level-1]; }**

**7.2. Borrar todo cuando se acabe el juego y tal.**

Si el juego se finiquita, entonces tenemos que reiniciarlo, por así decirlo. Es por ello que habrá que realizar las siguientes acciones:

Primero, expandimos la función *ovniHitsCraft()* con las siguientes líneas:

**ovnis.forEach(clearStage, this);**

**lasers.forEach(clearStage, this);**

**game.input.enabled = false; currentOvniProbability = -1;**

**game.time.events.add( 2000, continueGame, this);**

Y luego la función para la limpiza de la pantalla, claro:

**function clearStage(item){**

**item.kill();**

**};**

Finalmente, tenemos una función para hacer que furule el juego para delante:

**function continueGame() {**

**game.input.enabled = true;**

**if (lives > 0) {**

**var x = game.world.centerX;**

**var y = game.world.height - HUD\_HEIGHT;**

**craft.reset(x, y);**

**cursors.left.reset();**

**cursors.right.reset();**

**currentOvniProbability = LEVEL\_OVNI\_PROBABILITY[level-1 ];**

**}**

**else**

**startHOF();**

**};**

**8. *Hall of Fame*.**

El almacenamiento de los datos en un juego puede ser de dos tipos: **localStorage** (el utilizado más conmúnmente y el que gastaremos) y **sessionStorage** (cuando los datos solamente persisten por sesión). La constitución del primer tipo se realiza a partir del siguiente código:  
 **localStorage.name = 'Shooter';**

**localStorage['name'] = 'Shooter';**

**localStorage.setItem('name', 'Shooter');**

**var myname = localStorage.name;**

**myname = localStorage['name'];**

**myname = localStorage.getItem('name');**

**localStorage.removeItem('name');**

Nuestro ejemplo contiene una clase *HallOfFame*, encargada de todo lo relacionado con el almacenamiento del ranking de mejores jugadores. A continuación diremos qué hace cada apartado del *file hof.js*.

**loadFromStorage() →** Carga la información ya almacenada.

**saveToStorage() →** Hace justamente lo contrario a *loadFromStorage()*.

**parse() y stringify() →** Convierten o no la información con la que trabajamos en cadenas.

**addNewScore() →** Permite añadir una nueva *score* sirviéndose de los otros métodos ya realizados.

**displayOnStage()** → Escribe la lista del *hall of fame* en la pantalla del juego de *Phaser*.

**8.3. *hof.js* y *hofState*.**

*hofState* es el objeto principal del archivo *hof.js*, el cual lidera las acciones relativas a la pantalla del ranking. Este objeto solamente utiliza las fases *preload()* y *create()*.

En *preload()* nos encargamos de crear el objeto de tipo *HallOfFame*, usando la variable global *shooterHOF*, siendo cargado en el registro de puntuaciones que tenemos. También insertamos en esta fase el nuevo *score*.

La función *create()*, por su parte, genera la pantalla del *Hall of Fame* procesada previamente en la fase *preload*. Vamos, que es la que se encarga del apartado gráfico.

También permite una llamada a la función *restartPlay()*, encargada de reiniciar la partida.