1. **Introducción.**

Por fin hemos llegado a la práctica final. Podríamos decir que ahora toca algo más suave, pero mentiríamos porque vamos a ver un coñazo de la hostia. En este caso vamos a ver cómo se construye un modelo de movimiento de cámara. También nos explayaremos en el *HUD* (*Head-Up Display*) y en el uso de nuevas fuentes.

Veremos, en resumen:  
 - cómo utilizar fuentes especificas externas,

* Usar *tweens* para generar animaciones nuevas.
* Separar la parte del diseño con la referida a la programación.
* Usar la cámara para poder reflejar zonas nuevas del *background*.
* Mejorar nuestro conocimiento acerca de las cajas de colisión.

Para ello, como de costumbre, trabajaremos en base a un juego propuesto por el profesorado. El juego es un *scroll lateral* que se acabará en el momento en el que se pasen todos los niveles. Primero veamos el menú, que contiene el acceso a tres pantallas (representadas, a su vez, por tres estado, claro).



*ABOUT →* descripción.

*CONFIG →* elección de dificultad.

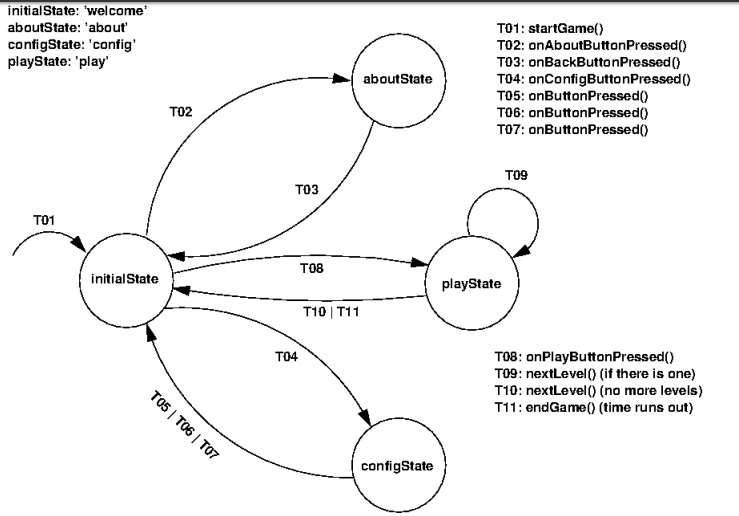
*PLAY →* inicio del juego.

El juego dispone de una barra de vida, de tal forma que cada ataque restará a la misma, mientras que la pantalla se irá moviendo en función del movimiento del jugador.

**2. Estructura del juego.**

La estructura del juego es similar al usado en la anterior ocasión (la del *shoot’em up*).

Disponemos de una serie de estados que se relacionan entre ellos mediante la ejecución de unas funciones, tal y como se muestra en la siguiente imagen:



En el **site root** del proyecto tenemos las carpetas *assets*, *css* y *js*, además del *index.html* (claro). La carpeta de *assetes* contiene todos los *sprites*, sonidos, etc. Por su parte, la de *CSS* lo único que hace es guardar el archivo *.css* del proyecto. Finalmente, la *js* guarda los archivos *javascript* del juego, que son: *welcome.js* (para *initialState*), *about.js* (para *aboutState*), *configure.js* (para *configState*) y *play.js* (para *playState*). También esta el archivo principal: *main.js*, que carga algunas fuentes externas.

Atentos a la sepración en *files* distintos el código del juego. Es bastante útil para no tener un puto quilombo de proyecto imposible de leer.

**3. Fuentes externas.**

A continuación vamos a ver cómo le introducimos fuentes externas (es decir, que no están en los paquetes por defecto del ordenador) a nuestro proyecto. Las que tenemos por defecto son *serif, sans-serif, monospace* y *cursive*. Todas las fuentes pertenecientes a estas familias podrán ser empleadas sin problemas. Por lo contrario, tocará hacer lo que describimos a continuación:

De primeras tocará descargarse una librería *js* la cual podemos encontrar en [**este enlace**](https://developers.google.com/fonts/docs/webfont_loader).

Su implementación es tal cual ocurría con *Phaser*, de tal forma que para emplear la librería tocará gastar las del pdf pasado (en el que se explicaba cómo introducir *phaser* a tu proyecto).

***Webfontloader*** básicamente nos descargará la fuente que queramos mediante el uso de *WebFontConfig*, objeto encargado de realizar este tipo de acciones. Para llamar al *loader* de esta librería será necesario utilizar el código siguiente (en este caso) en el *main.js*:

**var game;**

**var wfConfig = {**

**active: function () { startGame(); },**

**google: { families: ['Rammetto One', 'Sniglet'] },**

**custom: { families: ['FerrumExtracondensed'],**

**urls: ["https://fontlibrary.org/face/ferrum"]**

**}**

**};**

**WebFont.load(wfConfig);**

La variable *game*, por supuesrto, la veremos más tarde. Ahora centrémonos en *wfConfig,* que contiene por una parte una función a la que se asocia la tipografía, un parámetro *google* (que indica el navegador desde el que se extraerá la fuente), otro llamado *custom* (encargado de añadirle nuevas características a la información del tipo de escritura) y finalmente un parámetro *urls* que es desde el cual se extrae toda nuestros datos.

El cierre de este código se efectúa mediante el uso de la función *load* del objeto *WebFont*, que es el que ejecuta toda la información contenida en *wfConfig*. A continuación definimos los estados mediante la función de añadir estado que *game* nos proporciona en *Phaser*:

**function startGame() {**

**game = new Phaser.Game(800, 600, Phaser.CANVAS, 'platformGameStage');**

**// Welcome Screen**

**game.state.add('welcome', initialState);**

**// About Screen**

**game.state.add('about', aboutState);**

**// Config Screen**

**game.state.add('config', configState);**

**// Play Screen**

**game.state.add('play', playState);**

**game.state.start('welcome');**

**}**

En cuanto esto esté renderizado, las fuentes requeridas ya estarán preparadas para ser empleadas a nivel visual. Ahora, por supuesto, quedaría indicar de forma específica qué se gastará en cada momento. Para eso se seguirá una estructura como la que vemos a continuación (esto se sitúa en el *welcome.js*, eh):

**function displayScreen() {**

**[...]**

**var textTitle = 'Lab 6: A Simple Phaser Platform Game';**

**var styleTitle = {**

**font: 'Rammetto One',**

**fontSize: '22pt',**

**fontWeight: 'bold',**

**fill: '#b60404'**

**};**

**game.add.text(50, game.world.height / 6, textTitle, styleTitle);**

**[...]**

**}**

Pero chacho, vayamos al padre de todas las llamadas a fuentes externas, como quien dice… En *about.js* hay un momento en el que se llaman a fleje y no está mal mirarlo un poco más que sea:

**function showInstructions() {**

**[...]**

**var textTitle = 'A Simple Phaser Platform Game';**

**var styleTitle = {**

**font: 'Rammetto One', fontSize: '20pt',**

**fontWeight: 'bold',**

**fill: '#b60404'**

**};**

**game.add.text(75, 25, textTitle, styleTitle);**

**var credits = 'Brought to you by...\n';**

**game.add.text(125, game.world.height / 6, credits, {font: 'bold 26pt FerrumExtracondensed', fill: '#b60404'});**

**var authors = 'THE LECTURERS!!! (in alphabetical order):';**

**game.add.text(125, game.world.height / 6 + 75, authors, {font: 'bold 20pt Sniglet', fill: '#b60404'});**

**[...]**

**var instrucText = game.add.text(0, 0, instructions, {font: '14pt Sniglet', fill: '#b60404'}); [...]**

**}**

**3.1. Propiedades de texto y eventos.**

Ahora veamos algunas cuestiones interesantes, en este caso, del archivo *about.js*. Quizás el más interesante es que se pueden emplear manejadores en *Phaser* con elementos de texto. Basta con habilitar el *handler* mediante su propiedad *inputEnabled*. Miren por ejemplo el siguiente código, que sirve para manejar la aparición de texto (el evento se activa al poner el ratolí encima de la caja de texto):

**function showInstructions() {**

**[...]**

**author1 = game.add.text(175, game.world.height / 6 + 130, 'Juan Carlos Amengual Argudo', { font: 'bold 18pt Sniglet', fill: '#b60404'} );**

**author1.inputEnabled = true;**

**author1.events.onInputOver.add(overText, this); author1.events.onInputOut.add(outText, this);**

**[...]**

**}**

**[...]**

**function overText(text, pointer) { text.fill = '#0e0eb3';**

**}**

**function outText(text, pointer) {**

**text.fill = '#b60404';**

**}**

Otra cosa a tener en cuenta es que las instrucciones relativas al texto utilizan las propiedades *wordWrap* y *wordWrapWidth* para ajustar el texto dentro de los márgenes, y *boundsAlignH* junto a *boundsAlignV* para alinearlo:

**function showInstructions() {**

**[...]**

**var instructions = 'You will have to collect all of the stars before the timer runs out to exit each level. ';**

**[...]**

**var instrucText = game.add.text(0, 0, instructions, { font: '14pt Sniglet', fill: '#b60404' });**

**instrucText.setTextBounds(30, game.world.height - 170, game.world.width - 60); instrucText.boundsAlignH = 'center';**

**instrucText.boundsAlignV = 'middle';**

**instrucText.wordWrap = true;**

**instrucText.wordWrapWidth = game.world.width - 60;**

**[...]**

**}**

**4. *Tweens*.**

La propiedades animadas de los objetos (vamos, los *tweens*) son utilizadas a menudo para situaciones que requieran pequeños cambios puntuales en algún *sprite*. En este caso, tocará hacer uso de los métodos *Tween.to* y *Tween.from*. Generalmente, los parámetros que se gastarán serán las posición (x e y), la escala y un valor *alfa* que corresponderá al nivel de transparencia del objeto (para los *fade in* tocará ir de 0 a 1 y para los *fade out* será de 1 a 0). El *tween* funciona tal cual como las animaciones de *CSS*: se realiza una interpolación entre dos imágenes por medio de una serie de parámetros indicados.

Como hemos podido observar, en el ejemplo que gastamos se utilizan los cosos esos en el menú, de tal forma que tenemos que ir al archivo llamado *welcome.js* para encontrar un buen ejemplo del uso de los *tween*s.

**var mainTween, downTween1, downTween2;**

**[...]**

**function displayScreen() {**

**[...]**

**mainTween = game.add.tween(hero3) .to({y: 32}, 2000, Phaser.Easing.Linear.None) .to({angle: -90}, 500, Phaser.Easing.Linear.None) .to({x: game.world.width - (32 \* 2)}, 4000, Phaser.Easing.Linear.None);**

**mainTween.delay(3000); mainTween.loop(true);**

**mainTween.start();**

**downTween1 = game.add.tween(hero1.scale) .to({x: 5, y: 5}, 1500, Phaser.Easing.Cubic.Out) .to({x: 1, y: 1}, 1500, Phaser.Easing.Cubic.Out); downTween1.onComplete.add( onDownTweenCompleted, this);**

**downTween2 = game.add.tween(hero2) .to({alpha: 0.05}, 2500, Phaser.Easing.Linear.None) .to({alpha: 1.0}, 2500, Phaser.Easing.Linear.None); downTween2.onComplete.add( onDownTweenCompleted, this) downTween1.start();**

**[...]**

**}**

**function onDownTweenCompleted(object,tween)**

**{**

**if (tween === downTween1)**

**downTween2.start();**

**else**

**downTween1.start();**

**}**

Atentos a una cosa muy importante: estamos encadenando los *tweens*. Es decir, luego de acabar la acción del primero éste pasa a ejecutar las características del segundo, etc.

Esa concatenación es lo que hace interesante el uso de *tweens*. Imagínense la de movidas fleje cremas que se pueden hacer.

Realmente todos los *sprites* se están sacando de una sola imagen, que actúa a modo de *spritesheet*. Ahora bien… ¿Cómo sabe qué cacho coger? Pues mediante índices para cada *frame*, claro.

**function displayScreen() {**

**[...]**

**hero1 = game.add.sprite(game.world.width / 4, game.world.height - 200, 'hero', 4); hero1.anchor.setTo(0.5, 0.5);**

**[...]**

**}**

Es muy importante darse cuenta de otro detalle: solamente hemos llamado al primer *tween*.Es decir,se está efectuando como si de un recursivo se tratara, de tal forma que cada uno llama a la siguiente acción al final de su método.

**5. El juego.**

El código está almacenado en el archivo *play.js*, que corresponde al estado *playState* de *Phaser*. Se trata del único estado de tres fases del juego de ejemplo: *preload*, *create* y *update*. Cuando *phaser* corre la fase de *preload*, entonces la función *loadPlayAssets* será llamada. Ésta se compone de la siguiente manera:

**function loadPlayAssets() {**

**loadSprites();**

**loadImages();**

**loadSounds();**

**loadLevel(levelToPlay);**

**}**

Como se puede observar, solamente estamos cargando imágenes, sonidos, etc. por medio de métodos que sirvan a modo de intermediario. Ya ahondaremos en este tema.

En la fase *create* tocará llamar a la función *createLevel*. Acá hay mil cosas que comentar, pero *por el momento* nos quedaremos con estas pocas:

1. La existencia de la variable *exitingLevel*, declarada en *play.js* e inicializada en *false*. Esta variable se convertirá en *true* cuando el jugador haya coleccionado todas las estrellas (de cara a pasar a la siguiente fase del juego).
2. La habilitación de los *colliders* ahí tó guapos para poder usar la salida de cara a pasar al siguiente nivel. La acción se lleva a cabo gracias a la función *updateLevel* y *endLevel*:  
     
   **function updateLevel() {   
    [...]   
   if (!exitingLevel)   
   game.physics.arcade.overlap(player, exit, endLevel, null, this);   
   }   
    [...]   
   function endLevel() {   
   if (totalNumOfStars === 0) {   
   exitingLevel = true;   
   resetInput();   
   soundLevelPassed.play();   
   stopPlayer();   
   game.time.events.remove(timerClock);   
   game.time.events.add(4000, nextLevel, this);   
   }**Con *exitLevel* nos estamos ahorrando en salud de cara a un aspecto fundamental: impedir la llamada constante a la función *endLevel*, que se verá constantemente referenciada en caso de que no se le pare de alguna forma. Eso nos llevaría a ejecutar el audio asociado al final de cada nivel una y otra vez, superponiéndose.
3. La variable *totalNumOfStars* declarada en *play.js* está inicializada a 0. Este valor se le otorga antes de empezar cada nivel y se coloca en la fase de *create*, por supuestísimo.
4. Todas las plataformas de un nivel, incluyendo el suelo, están colocadas dentro de un mismo grupo (llamado *platforms*). Las físicas se activarán para todas las plataformas, claro:  
    **platforms = game.add.group();   
    platforms.enableBody = true;**
5. La variable *remainingTime* declarada en *play.js* está inicializada con el valor de *secondsToGo*. Eso lo veremos más tarde. Básicamente indica el tiempo que ha pasado desde que has empezado a jugar en ese nivel. Es obvio que toca ponerlo en el *create*, pues eso permitirá resetear el contador en caso de que, por ejemplo, se pase a otra fase.
6. El *sprite* del jugador es añadido al juego en coordinación con el archivo *JSON* para poder recortar la movida en plan bien, claro. También añadimos todos los valores relativos a la física del personaje y manejamos sus animaciones:  
   **// Player physics properties. Give the   
   // little guy a slight bounce.   
   player.body.bounce.y = 0.2;   
   player.body.gravity.y = BODY\_GRAVITY;  
   player.body.collideWorldBounds = true;   
    [...]   
   // Our two animations, walking left and   
   // right.   
   player.animations.add('left', [0, 1, 2, 3], 10, true); player.animations.add('right', [5, 6, 7, 8], 10, true);**
7. Finalmente habilitaremos el teclado y dispondremos de un temporizador para actualizar el tiempo pasado desde que se empezó el nivel.  
   **cursors = game.input.keyboard. createCursorKeys();   
   timerClock = game.time.events.loop( Phaser.Timer.SECOND, updateTime, this);**Las variables *damage*, *healthAid*, *secondsToGo*, *jumpsToKill* y *playerDeathTimePenatly* se declaran e inicializan en *welcome.js*.   
   Respectivamente, sus funciones son:  
   - El valor a sustraer de la barra de salud del personaje.  
   - El valor a añadir en la barra de salud cuando coges una caja de primeros auxilios.  
   - El número de segundos pasados al completar el nivel.  
   - El número de golpes a darle para eliminar un rival.  
   - El número de segundos que serán sustraídos cuando la barra de salud quede a 0.

Como era de esperar, todos estos parámetros son los realmente determinantes a la hora de colocar la dificultad del título. De hecho, gracias a que son valores con posibilidad de ser redefinidos podemos hacer las elecciones relativas al menú de elección de dificultad. Acá muestro el código de esa parte (situada en *configure.js*):

**function onButtonPressed(button) {**

**if (button === btnEasy) {**

**damage = DEFAULT\_DAMAGE;**

**healthAid = DEFAULT\_HEALTH;**

**secondsToGo = DEFAULT\_TIME;**

**jumpsToKill = DEFAULT\_JUMPS\_TO\_KILL;**

**playerDeathTimePenalty = DEFAULT\_PLAYER\_DEATH\_TIME\_PENALTY;**

**}**

**else if (button === btnAvg) {**

**damage = DEFAULT\_DAMAGE \* 1.5;**

**healthAid = DEFAULT\_HEALTH - 2;**

**secondsToGo = DEFAULT\_TIME - 90;**

**jumpsToKill = DEFAULT\_JUMPS\_TO\_KILL + 1;**

**playerDeathTimePenalty = DEFAULT\_PLAYER\_DEATH\_TIME\_PENALTY + 10;**

**}**

**else {**

**damage = DEFAULT\_DAMAGE \* 2;**

**healthAid = DEFAULT\_HEALTH - 5;**

**secondsToGo = DEFAULT\_TIME - 150;**

**jumpsToKill = DEFAULT\_JUMPS\_TO\_KILL + 3;**

**playerDeathTimePenalty = DEFAULT\_PLAYER\_DEATH\_TIME\_PENALTY + 25;**

**}**

**// Come back to welcome**

**game.state.start('welcome');**

**}**

Y ahora vamos a hablar del punto importante de los 3 estados: el *update* del estado *playState*, situado a su vez en la función *updateLevel* de *play.js*. En este punto se hacen bastantes cosas. Lo primero y más importante para el funcionamiento básico del juego es el detectado de colisiones (para las plataformas y el jugador, claro). Para ello tenemos la variable *hitPlatforms* (que detecta la situación de una plataforma con respecto al *sprite* del usuario) y una función *playerInPlatform* que permite saber cuál es la parte que está siendo tocada:

**var hitPlatform = game.physics.arcade.collide( player, platforms, playerInPlatform, null, this);**

**[...]**

**function playerInPlatform(player, platform) {**

**if (player.body.touching.down) playerPlatform = platform;**

**}**

Luego se comrpueba el *collider* de las estrellas y los botiquines:

**game.physics.arcade.collide(stars, platforms);**

**game.physics.arcade.collide(firstAids, platforms);**

Ahora vamos a comprobar si el jugador puede recoger una estrella o un botiquín. Observen que nosotros hemos suado *arcade.overlap* en vez de *arcade.collide*. Por otro lado, *collectStar* y *getFirstAid* emiten un sonido, eliminan un *ítem* y actualizan el número de estrellas o la barra de salud (respectivamente):

**game.physics.arcade.overlap(player, stars, collectStar, null, this); game.physics.arcade.overlap(player, firstAids, getFirstAid, null, this);**

**[...]**

**function collectStar(player, star) {**

**soundCollectStar.play();**

**star.kill();**

**totalNumOfStars -= 1;**

**}**

**function getFirstAid(player, aid) {**

**soundGetAid.play();**

**aid.kill();**

**healthValue = Math.min(MAX\_HEALTH, healthValue + healthAid);**

**updateHealthBar();**

**}**

El resto de funciones dedicadas a, por ejemplo, indicar si podemos ir a un nuevo nivel, son prácticamente iguales (siguen el mismo esquema).

Ahora pasemos a comentar tres aspectos a puntualizar:

* La variable *toRight* nos indica si el jugador se está moviendo a la izquierda o a la derecha.
* La función *stopPlayer* lo único que hace es evitar que se siga reproduciendo la animación de movimiento del bicho.
* *playerPlatform* pasa a *undefined* en caso de que el jugador salte.

**5.1. El mundo y la cámara.**

Entramos en el apartado más curioso de esta práctica. Como ya sabemos, la sección del nivel mostrado en pantalla mide 800x600. Ahora bien, el total realmente es mayor (de 1100 x 825). ¿Cómo hacemos pues para hacer que nos podamos desplazar? De primeras, lo normal en estos casos es cargar la imagen como si fuera un *sprite*:

**function createLevel() {**

**[...]**

**game.world.setBounds(0, 0, 1100, 825);**

**var bg = game.add.tileSprite(0, 0, game.world.width, game.world.height, 'bgGame');**

**bg.scrollFactorX = 0.7;**

**bg.scrollFactorY = 0.7;**

**[...]**

**}**

Atentos a que, como *scrollFactorX* y *scrollFactorY* son menos de 1 entonces la transición se produce de forma más suave. Por su parte, *setBounds* cambia el tamaño del mundo.

El mundo se mueve en función del jugador gracias a la función *follow* del objeto *camera* (definido en el *create* de *play.js*):

**game.camera.follow(player);**

Con respecto al temporizador, el descuento de tiempo se efectúa gracias a *updateTime*:

**function updateTime() {**

**remainingTime = Math.max(0,remainingTime-1);**

**hudTime.setText(setRemainingTime( remainingTime));**

**if (remainingTime === 0) {**

**resetInput();**

**soundOutOfTime.play();**

**stopPlayer();**

**game.time.events.remove(timerClock);**

**game.time.events.add(2500, endGame, this);**

**}**

**}**

**[...]**

**function endGame() {**

**clearLevel();**

**goToWelcome();**

**}**

Si se acaba el juego entonces tocará ir de nuevo a la pantalla de inicio. Eso se logra con la función *goToWelcome*:

**function goToWelcome() {**

**game.world.setBounds(0, 0, game.width, game.height);**

**game.state.start('welcome');**

**}**

**5.2. Estrellas y cajas de primeros auxilios.**

Estos elementos son creados mediante las funciones *createStars* y *createAids*. Las dos funciones son llamadas dentro de *createLevel*.

**function createAids() {**

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

**firstAids.enableBody = true;**

**firstAids.createMultiple(MAX\_AIDS, 'aid');**

**firstAids.forEach(setupItem, this);**

**}**

**function createStars() {**

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

**stars.enableBody = true;**

**stars.createMultiple(MAX\_STARS, 'star');**

**stars.forEach(setupItem, this);**

**}**

**function setupItem(item) {**

**item.anchor.setTo(0.5, 0.5);**

**item.body.gravity.y = BODY\_GRAVITY;**

**}**

Colocarle gravedad al objeto es necesario para poder gastar luego las funciones *setup*:

**function setupAid(aid, floorY) {**

**var item = firstAids.getFirstExists(false);**

**if (item)**

**item.reset(aid.x, floorY - AID\_STAR\_Y\_OFFSET);**

**}**

**function setupStar(star, floorY) {**

**var item = stars.getFirstExists(false);**

**if (item) {**

**item.reset(star.x, floorY - AID\_STAR\_Y\_OFFSET);**

**totalNumOfStars += 1;**

**}**

**}**

**5.3 El ataque de los enemigos.**

Los enemigos están presentes en los dos niveles que componen el juego con tal de dificultar la obtención de las estrellas. Su constitución se realiza mediante la siguiente función:

**function Enemy(spritesheet, tween, plat, right, limit)**

**{**

**this.sprite = spritesheet;**

**this.flash = tween;**

**this.platform = plat;**

**this.faceright = right;**

**this.stepLimit = limit;**

**this.origX = spritesheet.x;**

**this.hitsToBeKilled = jumpsToKill;**

**this.isPatrolling = true;**

**}**

Por cada ratolí nosotros almacenaremos su *sprite*, el *tween* y la plataforma donde anda situado. Además de un booleano que indica si está mirando hacia la izquierda o hacia la derecha. Cada personaje de tipo *enemigo* se almacena un array que será el que cuente el total que aparecen por pantalla. Además, dependiendo de hacia donde mire el bicho, tendremos un tope hacia la derecha o hacia la izquierda, tal y como se muestra en el siguiente código:

**var theEnemy = game.add.sprite(enemy.x, plat.y - ENEMY\_Y\_OFFSET, 'enemy');**

**theEnemy.anchor.setTo(0.5, 0.5);**

**if (enemy.right === 0) {**

**theEnemy.scale.x = -1;**

**isRight = false;**

**limit = Math.max(Math.max(0, plat.x) + ENEMY\_X\_OFFSET, enemy.x-ENEMY\_STEP\_LIMIT);**

**}**

**else {**

**isRight = true;**

**limit = Math.min(Math.min(plat.x + plat.width, game.world.width) - ENEMY\_X\_OFFSET, enemy.x + ENEMY\_STEP\_LIMIT);**

**}**

También se crea un *tween* con dos animaciones (para el ataque y para cuando corre):

**var flash = game.add.tween(theEnemy).to( {alpha: 0.0}, 50, Phaser.Easing.Bounce.Out) .to({alpha: 0.8}, 50, Phaser.Easing.Bounce.Out) .to({alpha: 1.0}, 50, Phaser.Easing.Circular.Out);**

Cuando un ratolí está caminando la función *patrol* debe ser llamada. Ésta recibe una instancia de *Enemy*. Su código se basa es una velocidad que se pasa como parámteo, así como un indicativo de que el enemigo está en el límite de su movimiento hacia un lado (lo que conlleva un cambio de lado):

**if (enemyObject.sprite.body.x < thePlayer.body.x) {**

**enemyObject.sprite.scale.x = 1;**

**playerAtRight = true;**

**}**

**else {**

**enemyObject.sprite.scale.x = -1;**

**playerAtRight = false;**

**}**

A continuación incrementamos la velocidad del ratón:

**enemyObject.sprite.body.velocity.x = Math.trunc( ENEMY\_VELOCITY \* 1.4) \* enemyObject.sprite.scale.x;**

**enemyObject.sprite.animations.play('swing');**

Luego tocará cambiar los valores de los límites de X, puesto que cuando el enemigo se encuentra cerca del personaje principal se deberá acercar a atacar, independientemente de su posición. Eso quiero decir que habrá que recalcular los parámetros:

**if (enemies[i].platform === playerPlatform) {**

**dist = Phaser.Math.distance(enemies[i].sprite. body.x, enemies[i].sprite.body.y, player.body.x, player.body.y);**

**if (Math.round(dist) <= ENEMY\_DISTANCE\_ATTACK)**

**enemies[i].isPatrolling = false;**

**else**

**enemies[i].isPatrolling = true;**

**}**

**else**

**enemies[i].isPatrolling = true;**

**if (enemies[i].isPatrolling)**

**patrol(enemies[i]);**

**else**

**attack(enemies[i], player);**

**if (game.physics.arcade.collide(player, enemies[i].sprite))**

**playerVsEnemy(player, enemies[i].sprite, enemies[i], i);**

Por otra parte, también tenemos *resetPlayer*, que básicamente nos devuelve a los parámetros originales en caso de que la barra de salud pasa a 0:

**function resetPlayer() {**

**stopPlayer();**

**player.x = levelConfig.collectorStart.x;**

**player.y = game.world.height - levelConfig.collectorStart.y;**

**remainingTime = Math.max(0, remainingTime - playerDeathTimePenalty);**

**healthValue = MAX\_HEALTH;**

**updateHealthBar();**

**}**

**6. Diseño de niveles (*JSON*)**

Se aconseja separar el diseño y la preparación del juego de la programación y su desarrollo. Así, se suele emplear algún formato de datos independiente del idioma de programación que se enmplea, como puede ser, por ejemplo, *JSON*. Para cada nivel, hemos hecho un *JSON* tó guapo. Para el primer nivel, por ejemplo, tenemos:

**{**

**"collectorStart": {"x": 32, "y": 90},**

**"ground": {"enemies": [ {"x": 240, "right": 0}, {"x": 880, "right": 0} ],**

**"aids": [ {"x": 470}, {"x": 955} ],**

**"stars": [ {"x": 100}, {"x": 190}, {"x": 840} ] },**

**"platformData": [ {"x": 0, "y": 350, "enemies": [ {"x": 200, "right": 1} ],**

**"aids": [ {"x": 20} ],**

**"stars": [ {"x": 100} ] },**

**[...]**

**{"x": 800,**

**"y": 650,**

**"enemies": [ {**

**"x": 900,**

**"right": 0} ],**

**"aids": [ ],**

**"stars": [ {"x": 1050} ]**

**}**

**]**

**}**

El archivo contiene un objeto (el nivel) que tiene una serie de nombres/valores. Lo nombres están en formato cadena y sirven como clave para acceder al contenido que queremos. De hecho, todo está justificado de tal manera que podamos evaluar toda la información de forma estandarizada.

¿Cómo leemos el archivo *JSON*? Primero debemos crear una nueva variable que forme toooda la info:

**var levelsData = ['assets/levels/level01.json', 'assets/levels/level02.json'];**

Como se puede observar, la separación entre el contenido del juego y la programación nos permite añadir nuevo contenido al juego sin necesidad de estar liándola de mala manera. Eso es muy importante, pues permite tirar de reutilizar código. La carga del nivel (así como de la información numérica) se realiza en la funciones pertinentes:

**function loadPlayAssets() {**

**[...]**

**loadLevel(levelToPlay);**

**}**

**[...]**

**function loadLevel(level) {**

**game.load.text('level', levelsData[level-1], true);**

**}**

**var levelToPlay;**

**[...]**

**function displayScreen() {**

**levelToPlay = 1;**

**[...]**

**}**

Importante: el método *game.load.text* carga el archivo *JSON* y tiene como tercer parámetro un *true* que actúa como indicativo para permitir sobreescribir la información.

Ahora, en la fase *create*, dentro de la función *createLevel*, se llama a *JSON.parse*, pasando los datos a la caché como parámetro:

**levelConfig = JSON.parse(game.cache. getText('level'));**

Las funciones relativas a la creación de *sprites* en el interfaz gráfico suelen tirar de bucles (con tal de poder administrar todos los enemigo y mierdas de esas):

**function createGround() {**

**ground = platforms.create(0, game.world.height - 64, 'ground');**

**ground.scale.setTo(2.75, 2);**

**ground.body.immovable = true;**

**for (var i = 0, max = levelConfig.ground. enemies.length; i < max; i++)**

**setupEnemy(levelConfig.ground.enemies[i], ground);**

**for (var i = 0, max = levelConfig.ground. aids.length; i < max; i++)**

**setupAid(levelConfig.ground.aids[i], ground.y);**

**for (var i = 0, max = levelConfig.ground. stars.length; i < max; i++)**

**setupStar(levelConfig.ground.stars[i], ground.y);**

**}**

**function createPlatforms()**

**{**

**levelConfig.platformData.forEach( createPlatform, this);**

**}**

Para pasar al siguiente nivel, solamente tendremos que hacer una llamada al *updateLevel* situado en *play.js*.

**function updateLevel() {**

**[...]**

**if (!exitingLevel)**

**game.physics.arcade.overlap(player, exit, endLevel, null, this);**

**}**

**[...]**

**function endLevel() {**

**if (totalNumOfStars === 0) {**

**exitingLevel = true;**

**resetInput();**

**soundLevelPassed.play();**

**stopPlayer();**

**game.time.events.remove(timerClock);**

**game.time.events.add(4000, nextLevel, this);**

**}**

**}**

La función *endLevel* será llamada únicamente en caso de que colisionemos con la salida.

La funcióna seguir se trata de *nextLevel*. Cuando se ejecuta, hay un *delay* de 4 segundos (en plan pa que suene la musiquita y tal). El reseteo del teclado se efectua mediante la función *resetInput*:

**function resetInput() {**

**game.input.enabled = false;**

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

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

**cursors.up.reset(true);**

**cursors.down.reset(true);**

**}**

**function nextLevel() {**

**clearLevel();**

**levelToPlay += 1;**

**if (levelToPlay > levelsData.length)**

**goToWelcome();**

**else {**

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

**game.state.start('play');**

**}**

**}**

**7. HUD**

El *HUD* de nuestro juego es realmente un grupo de *Phaser* que consiste en:

* La imagen de un corazón.
* La imagen de la barra de salud.
* La imagen de un rectángulo lleno representando la salud del jugador.
* Un texto mostrando el tiempo que falta pa completar el nivel.

La función de éste es:

**function createHUD() {**

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

**hudGroup.create(5, 5, 'heart');**

**hudGroup.create(50, 5, 'healthHolder');**

**healthBar = hudGroup.create(50,5,'healthBar'); hudTime=game.add.text(295,5,setRemainingTime( remainingTime), {**

**font: 'bold 14pt Sniglet', fill: '#b60404'**

**});**

**hudGroup.add(hudTime);**

**hudGroup.fixedToCamera = true;**

**healthValue = MAX\_HEALTH;**

**}**

Para actualizar la barra de salud, tocará tirar de una nueva función a la que llamaremos *updateHealthBar*, encargada de actualizar el estado de dicho objeto gráfico:

**function updateHealthBar() {**

**if (healthTween)**

**healthTween.stop();**

**healthTween = game.add.tween( healthBar.scale);**

**healthTween.to({x: healthValue/MAX\_HEALTH}, 300);**

**healthTween.start();**

**}**

**8. Ajustando la caja de colisiones del *sprite*.**

Las físicas *Arcade* se basan en el uso de las *bounding box*, que permite detectar colisiones. Estas cajas pueden ser grandes y a veces no nos sirven (por el tamaño del sprite a contener y mierdas de esas). Es por eso que tocará cambiarlas de alguna manera. Eso lo hacemos dentro de la función *createLevel*:

**function createLevel() {**

**[...]**

**exit.body.setSize(88, 58, 20, 33);**

**[...]**

**}**

**[...]**

**function setupEnemy(enemy, plat) {**

**[...]**

**theEnemy.body.setSize(41, 43, 3, 10);**

**[...]**

**}**

Se puede saber la zona donde se ha chocado un objeto (arriba, abajo, izquierda o derecha) gracias a los métodos propios de las *bounding box*: hablamos de ***right, left, down*** y ***up***.

**function playerInPlatform(player, platform) {**

**if (player.body.touching.down)**

**playerPlatform = platform;**

**}**

De hecho, esto de las métodos es útil para cargarse al enemigo. La estructura que sigue sería algo así como esto:

**function playerVsEnemy(player, enemySprite, enemyObject, position) {**

**if (enemySprite.body.touching.up) {**

**[...]**

**}**

**9. Básicas para la depiuración de juegos en *Phaser*.**

*Phaser* contiene una colección de métodos relativos a la depuración de los objetos del juego. Ellos están incluidos en la clase *Debug*. Podemos usar *Phaser* para ajustar las *bounding boxes* de algunos *sprites* y arreglar los *bugs* de las colisiones.

Si tu juego está corriendo en modo *Canvas* (como es nuestro caso), entonces debes llamaer a todos los métodos de depuración desde la función enlazada a la fase de *render* del estado del juego que quieres deurar. Esto se debe a que dibuja en el *canvas* directamente todos los elementos.

En *Phaser* hay un bucle interno que se encarga de todo esto en el *update*.