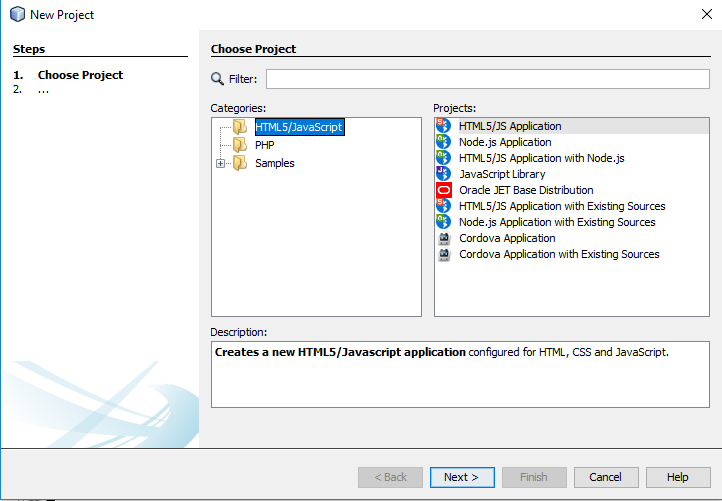
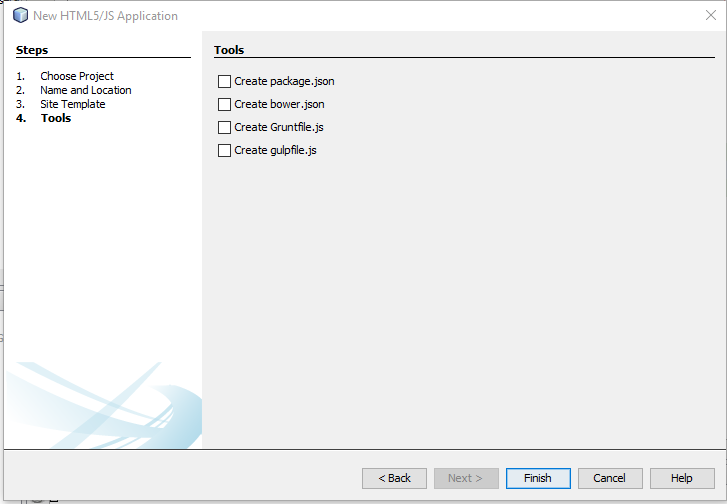
A continuación vamos a hacer una introducción a *Phaser*, que es un entorno de desarrollo de juegos web muy utilizado actualmente. En este documento nos centraremos sobre todo en definir tres conceptos: ***sprite sheet***, ***tweening*** y ***particles***.

Primero veamos cómo abrir el bueno de *Phaser*. Como tal, debe de abrirse por medio de *NetBeans*. Primero debemos descargarnos la versión de *Phaser* que nos hayan dado en el aula virtual (en mi año fue la versión **2.9.4**, pero muy probablemente haya cambiado) y luego, en *NetBeans*, seguir los siguientes pasos:

De primeras le damos a *file → new project*. Ahí le damos a *Next* a la primera ventana de la misma manera que siempre.

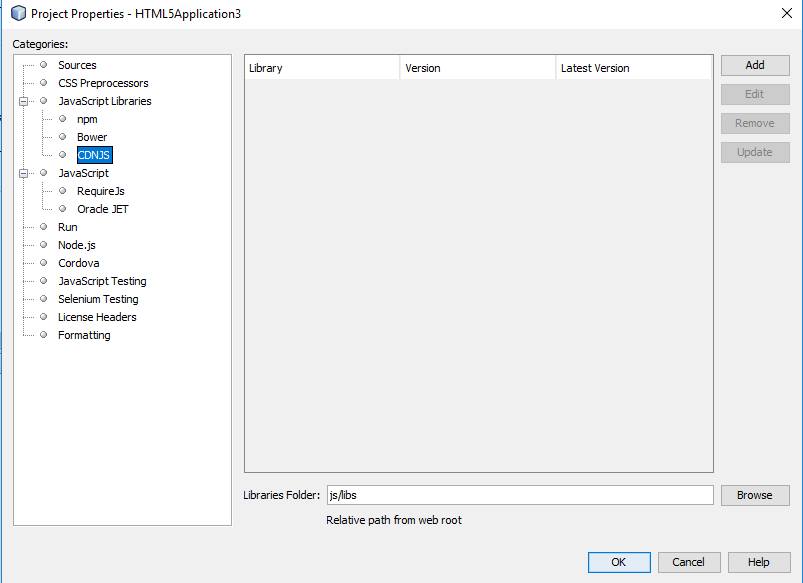


Dale a *Next* en todo hasta llegar al *step 4 (tools)*. Ahí deseleccionas todo y pulsas *Finish.*

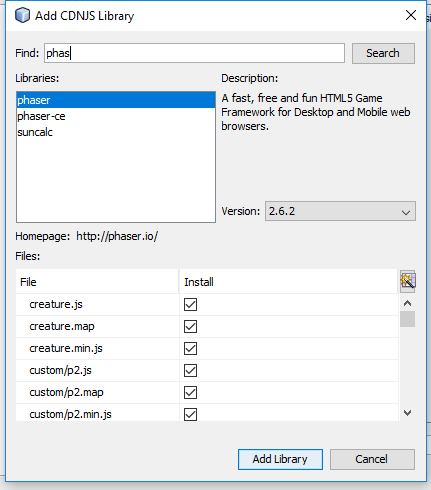


Luego haces click derecho en la pestaña principal de tu nuevo proyecto (la del globo terráqueo en miniatura) y seleccionas *properties* (la última opción que se te ofrece en la barra que se te abre).

A continuación pulsamos donde pone *CDNJS* y luego al botón *Add* de la esquina superior derecha.



Se te abrirá una ventana como esta:



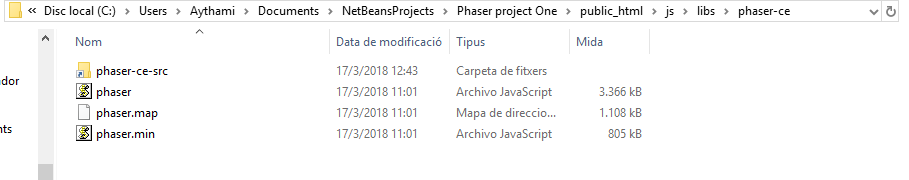
Tendrás que poner en donde pone *Find:* la palabra *phas* y darle a *search*. Entonces aparecerán unas opciones en la caja de texto que pone *Libraries*. Ahí debemos coger la que pone *phaser-ce*. Cuando hagas click aparecera una opción que dice *Add Library*. Aceptas y ya estaría… casi.

Faltaría hacer un enlace de manera simbólica lo que hemos descargado con el proyecto. Para ello tocará entrar en la consola de tu ordenador. Como yo soy un puto weaboo de mierda que usa windows, explicaré lo que se hace para este sistema operativo.

De primeras tocará abrir el *cmd* en **modo administrador**. Para ello ponemos en el buscador cmd y en lo primero que salga pulsamos click derecho. La primera opción que te dan es la de ejecutar el cmd como administrador. Le damos a ese y aceptamos cuando nos pregunten si estamos seguros. Ahora ya podemos crear cosos con la consola c:

Como ya sabemos por los vastos conocimientos que Espinosa nos legó antes de unirse con la fuerza, la creación de un vínculo simbólico se realiza mediante la orden ***mklink***. Para ello primero debemos ponernos en la localización donde queremos tener el acceso directo a nuestra carpeta referenciada.

Vamos a donde tengamos guardado el proyecto y abrimos la carpeta *phaser-ce*.



Apretamos dentro de la barra de arriba (donde se ven las carpetas) y copiamos la ruta que aparece.

Ahora vamos al *cmd* y escribimos la orden ***cd***, así como la ruta previamente copiada.

Luego toca colocar la orden *mklink*, que sigue esta estructura:

**mklink /D phaser-ce-src** *ruta de la carpeta que hemos descargado antes.*

Lo ejecutas y en teoría debería irte bien. Para comprobarlo puedes ir a la carpeta *phaser-ce* de tu proyecto y comprobar si, en efecto, hay una carpeta con un icono de flechita que te lleve a *phaser-ce-src*. Este proceso debe repetirse para todos los trabajos hechos en este entorno, así que hay que saber seguir estos pasos de forma correcta, no sea que luego haya moraleja.

A continuación veremos la estructura clásica de un programa básico de *Phaser*:

**// game instance**

**var game = new Phaser.Game(400, 400, Phaser.CANVAS, ’game’);**

**// states**

**var playState = { preload: preload,**

**create: create,**

**update: update };**

**game.state.add(’main’, playState);**

**game.state.start(’main’);**

**// methods**

**function preload() {**

**game.load.image(’logo’, ’pics/uji.png’)**

**}**

**function create() {**

**game.stage.backgroundColor = "#4488AA";**

**logoImg = game.add.image(0, 0, ’logo’); logoImg.scale.setTo(0.2);**

**}**

**function update()**

**{**

**}**

Como es tradición, veamos cacho a cacho qué carajo está pasando:

De primeras, creamos la variable que almacenará nuestro juego, que será *game*. Básicamente será un *Phaser.Game()* cuyos parámetros son, respectivamente: alto, ancho, el contexto de renderizado (que será un *canvas*) y el identificador del juego en el .*html*. Por esto último me refiero a que, cuando pongamos la etiqueta en el archivo *.html* tocará colocar lo siguiente:

**<div id="game"> </div>**

A continuación creamos los estados de *preload*, *create* y *update*, que quedan inicializados en la variable ***playState***. Éstos básicamente son las pantallas de las que se disponen (de carga, de curso, etc.). Cada una a su vez está asociada a una función que es la que determina las acciones a realizar para cada uno de los casos presentados.

A su vez, la variable *game* acepta los estados a través de su objeto *state*, que gastaremos mediante las dos siguientes líneas:

**game.state.add()**

**game.state.start()**

La primera añade un estado (en este caso el formado por *playState*) y le pone un nombre para su identificación. Por su parte, *game.state.start()* se dedica a inicializar un estado, al cual accede mediante el nombre que le dimos como parámetro al anterior método.

Volviendo a *playState*, estamos observando que hay tres funciones definidas en la variable. ¿Qué hace cada una? Pues, dentro de lo que cabe, su explicación es sencilla:

* *preload()* básicamente prepara todos los *assets* que emplearemos en el juego.
* *create()* los inicializará para poder comenzar el juego.
* *update()*, como su propio nombre indica, se encarga de cambiar los elementos que durante el proceso de juego. Básicamente esta es la manera mediante la cual se implementará el *game loop* (osea, el bucle mediante el cual ocurren los sucesos del juego).

Vale, a continuación veremos los elementos básicos mediante el uso de un sencillo juego en donde el usuario introducirá letras, éstas se moverán de manera aleatoria y se eliminarán por medio de hacer *click* sobre ellas. El número de aciertos se mostrará por pantalla bajo el título de *Hits*.

**Eventos de teclado y ratón.**

Iremos denagrando cada parte en función de lo que hay que meter en cada función (*create, preload* y *update*).

Así, dentro de la función *create* tocará introducir la siguiente sentencia:

**game.input.keyboard.onDownCallback = getKeyboardInput;**

Básicamente, con esa línea estamos indicando que al presionar una tecla hay que recurrir a la función *getKeyboardInput()*, la cual definiremos ahora.

**function getKeyboardInput(e) {**

**if (e.keyCode >= Phaser.Keyboard.A && e.keyCode <= Phaser.Keyboard.Z)**

**var a = game.add.text (Math.random() \* game.width,**

**Math.random() \* game.height, e.key,**

**{fontSize: ’40px’, fill: ’#FA2’}, letters); // group to add to**

**}**

Atentos al condicional de la función. En este caso estamos aplicando sobre el atributo *e* su método *keyCode*. Esa *e* realmente es un objeto de tipo evento (como tal no hace falta que vaya declarado en la llamada). El *keyCode* lo que hace es indicar el código con el que *Phaser* conoce la tecla pulsada. La disposición de las letras del teclado es como ocurre en el código *ASCII* (donde se sigue el orden alfabético). Así, podemos cercar las teclas que queremos mediante el uso del formato *Phaser.Keyboard.[letra]*.

Finalmente, ***game.add.text()*** lo que hace es imprimir las teclas pulsadas. Atento a que se determina de manera aleatoria (con *Math.random()*) los valores de anchura y altura de la letra, mientras que el estilo tipográfico se especifica mediante los parámetros *fontSize* y *fill*.

Ahora bien, para que este código funcione hace falta que pulamos unos detalles importantes, como el tema de *letters*, que aparece referenciado pero que nunca se llega a especificar nada acerca de cómo funciona. Básicamente es un grupo donde se introducen los valores que hemos pasado por teclado. Para que eso sea factible debemos, de primeras, declarar la variable en *create* mediante la siguiente línea:

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

Por su parte, también habrá que activar la opción de que cada letra que se introduzca actúe a modo de hijo. Para ello habrá que poner a *true* el valor de acceso de hijos del objeto:

**letters.inputEnableChildren = true;**

Y ya estaría: ahora las letras que coloques aparecerán por pantalla :-)

Como hemos dicho antes, también queremos que se muevan las letras. Para ello habrá que tirar de una nueva función, que colocaremos en ***update()***.

La función **moveLetters()** funciona de la siguiente manera:

**function moveLetters() {**

**for (var i = 0, len = letters.children.length; i < len; i++)**

**{**

**letters.children[i].x += Math.random() \* 2 - 1;**

**letters.children[i].y += Math.random() \* 2 - 1;**

**letters.children[i].angle += Math.random() \* 10 - 5;**

**}**

**}**

No es muy complicado entender lo que está haciendo: está calculando con aleatoriedad los tres aspectos básicos de la posición del objeto: su situación en *xy* y el ángulo que toma en su avance. Todo esto se facilita con el *for* que recorre las letras que hemos colocado en pantalla.

Como ya hemos dicho, la llamada a esta función va en *update*, porque esto es algo que iremos actualizando constantemente y tal.

Ahora veamos qué carajo hacemos con el último aspecto general que nos queda por tocar: la interacción con el ratolí y las letras. Para ello, en el *update* habrá que definir un nuevo evento que se sustente con una función que haremos más tarde:

**letters.onChildInputDown.add(processLetter);**

La función en cuestión, como podemos observar, se llama *processLetter*, que consta de dos parámetros que, como son fruto de un evento, no hace falta colocar como tal:

**function processLetter(item, pointer) {**

**item.destroy();**

**}**

Atentos a que estamos usando el *destroy* para eliminar el elemento, que en este caso será una letra.

A continuación veamos cómo hacer un contador de cuántas letras hemos quitado con el clickeo. Para ello deberemos introducir en el *create()* el contador.

**var numHits=0;**

**scoreText = game.add.text(**

**15, // x**

**logoImg.y+logoImg.height, // y**

**’Hits: ’+numHits, // text**

**{fontSize: ’32px’, fill: ’#000’});**

Y ya estaría. Como se puede observar, la concatenación se efectúa pese a contener un valor entero. Puta madre para nosotros, loco.

**Los sprites.**

Ahora veamos la parte más gráfica de toda esta movida, empezando por los *sprites*. Éstos se forman a partir de un *sprite sheet*, que como ya sabemos, se trata del conjunto de imágenes que conforman la animación de un determinado personaje. De primeras veamos un ejemplo y ya luego metámonos en el rollo de cómo carajo hacer el *spritesheet*:

**var game = new Phaser.Game(800, 600, Phaser.CANVAS, 'phaser-example', { preload: preload, create: create });**

**function preload() {**

**game.load.spritesheet('mummy', 'assets/sprites/metalslug\_mummy37x45.png', 37, 45, 18);**

**}**

**function create() {**

**var mummy = game.add.sprite(300, 200, 'mummy');**

**var walk = mummy.animations.add('walk');**

**mummy.animations.play('walk', 30, true);**

**}**

Como podemos observar, la animación de esta momia se divide en dos funciones que, al igual que en el ejemplo anterior, sirven para cargar y crear los elementos de los que vamos a disponer.   
Atentos a cómo en *preload* se carga el *spritesheet*, gracias a la función ***game.load.spritesheet()*** que tiene como parámetros, en este orden:

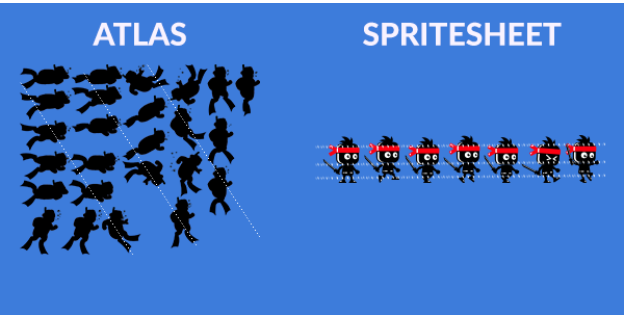
1. El nombre con el que se referenciará el *spritesheet*.
2. Su localización.
3. El largo.
4. El ancho.
5. El número de *frames* que contiene.

Después tenemos la función *create*, que lleva a otro nivel lo que ya hemos cargado. Dentro tenemos una nueva variable llamada *mummy* donde se guarda la partición de la imagen que ya hemos subido en un tamaño especificado en los dos primeros parámetros del método *sprite*.

Llegados a este punto ya tenemos todas las imágenes, pero no las tenemos colocadas como si fuera una animación. Para ello hay que tirar de crear una nueva variable (que se ha llamado *walk*) que contenga el resultado de aplicar el método *.animations.add()*, con la etiqueta (“walk” en este caso) como único parámetro.

Finalmente, para poner en marcha la animación, se hace uso del *.animations.play()*, cuyos tres parámetros son la etiqueta (“walk”), los *fps* y un booleano que indique la repetición de la animación cuando ésta se haya acabado.

Luego están los *texture atlases*, que son algo más complejos en cuanto a su uso, pero suelen ser preferibles de cara a hacer cosas un poco más profesionales. La diferencia entre el *spritesheet* es que el *Atlas* no necesariamente debe seguir un orden lineal, siendo más fácil componer la imagen. Vean este ejemplo:



Para crear este coso toca trabajar en un nuevo formato de archivo: el **JSON**. No hay que preocuparse, pues su implementación es relativamente sencilla. Básicamente nuestra estrategia a seguir va a ser tener una imagen *PNG* a modo de tablero y un JSON para trabajar encima (como si fuese papel cebolla, vamos). En este JSON lo que haremos será delimitar cada frame y nombrarlo para poder trabajar con él más tarde. Eso está de puta madre por una razón de peso: si lo que hacemos es nombres cada cachito, eso significa que a posteriori no será necesario utilizar todo lo que hemos extraído para lo mismo (como sí ocurría en el *spritesheet*)... entonces nos viene de puta madre para gastar, en un mismo *PNG*, todos los dibujos que se muestren por pantalla (es decir: fondos, personajes, etc.).

Veamos un ejemplo del *JSON*:

**{"frames": {**

**"background":**

**{**

**"frame": {"x":2,"y":2,"w":800,"h":400},**

**"rotated": false,**

**"trimmed": false,**

**"spriteSourceSize": {"x":0,"y":0,"w":800,"h":400},**

**"sourceSize": {"w":800,"h":400}**

**},**

**"capguy/walk/0001":**

**{**

**"frame": {"x":967,"y":2,"w":158,"h":316},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":15,"y":3,"w":158,"h":316},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0002":**

**{**

**"frame": {"x":1850,"y":2,"w":168,"h":303},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":0,"y":8,"w":168,"h":303},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0003":**

**{**

**"frame": {"x":1553,"y":2,"w":152,"h":307},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":26,"y":2,"w":152,"h":307},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0004":**

**{**

**"frame": {"x":1707,"y":2,"w":141,"h":306},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":42,"y":1,"w":141,"h":306},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0005":**

**{**

**"frame": {"x":1275,"y":2,"w":139,"h":311},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":33,"y":3,"w":139,"h":311},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0006":**

**{**

**"frame": {"x":1416,"y":2,"w":135,"h":311},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":29,"y":8,"w":135,"h":311},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0007":**

**{**

**"frame": {"x":1127,"y":2,"w":146,"h":314},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":32,"y":2,"w":146,"h":314},**

**"sourceSize": {"w":187,"h":324}**

**},**

**"capguy/walk/0008":**

**{**

**"frame": {"x":804,"y":2,"w":161,"h":319},**

**"rotated": false,**

**"trimmed": true,**

**"spriteSourceSize": {"x":22,"y":1,"w":161,"h":319},**

**"sourceSize": {"w":187,"h":324}**

**}},**

**"meta": {**

**"app": "http://www.codeandweb.com/texturepacker",**

**"version": "1.0",**

**"image": "cityscene.png",**

**"format": "INDEXED",**

**"size": {"w":2020,"h":404},**

**"scale": "1",**

**"smartupdate": "$TexturePacker:SmartUpdate:3f95b1feed2fbfc4385b49348c54e958:fed0f696c0a1349f4c5839d5f65ab643:292542e6853f316339413b9243f95b10$"**

**}**

**}**

Claro, esto para ustedes resulta una parrafada muy basta, pero si se dan cuenta, lo único que estamos haciendo es darle un punto de referencia y una altura/anchura para cortar un rectángulo y tratarlo como un *frame*, repitiendo este proceso toooodo el tiempo. La verdad es que no tiene más ciencia.

Ahora veamos la aplicación del *JSON* en la el fichero principal, pues para activar todo lo referido a los gráficos tocará tirar de ciertos métodos contenidos en el *game*.

Pongámonos en la siguiente situación: el *JSON* ya está creado y queremos colocar un fondo y un machango caminando (los sprites de todo esto ya están, claro). ¿Qué hacemos pues? Tocará colocar el siguiente código:

**<!doctype html>**

**<html>**

**<head>**

**<meta charset="UTF-8" />**

**<title>TexurePacker+Phaser Demo!</title>**

**<script src="phaser.min.js"></script>**

**</head>**

**<body>**

**<script>**

**window.onload = function() {**

**var game = new Phaser.Game(800, 400, Phaser.AUTO, '', { preload: preload, create: create, update: update });**

**var capguy;**

**function preload () {**

**game.load.atlasJSONHash('cityscene', 'cityscene.png', 'cityscene.json');**

**}**

**function create () {**

**var background = game.add.sprite(0, 0, 'cityscene', 'background');**

**capguy = game.add.sprite(800, 180, 'cityscene', 'capguy/walk/0001');**

**capguy.scale.setTo(0.5,0.5);**

**capguy.animations.add('walk', Phaser.Animation.generateFrameNames('capguy/walk/', 1, 8, '', 4), 10, true, false);**

**capguy.animations.play('walk');**

**}**

**function update() {**

**capguy.x += 3;**

**if(capguy.x > 800)**

**{**

**capguy.x = -50;}}}; </script> </body> </html>**

Parémonos detenidamente a ver qué ocurre con esté código.

En el **preload** hemos cargado un nuevo *atlas*, que hemos llamado *cityscene*, cuyo contenido está extraído de *cityscene.png* y su *json* correspondiente es *‘cityscene.json’*.

En el **create** creamos nuestros dos objetos: el *background*(Que se posiciona en el punto 0,0 y que corresponde al corte de *cityscene* correspondiente a *background*) y *capguy* (que inicialmente toma el valor de *capguy/walk/0001*).

Luego tocará hacer dos cosas: escalar y animar. El proceso de escalado, como ya sabemos, consiste en reducir o ampliar (en función del valor que le demos) el formato el tamaño de una imagen, tango en *x* como en *y*. En nuestro caso estamos reduciendo su tamaño a la mitad. En caso de querer, por ejemplo, que el muñeco mire hacia el otro lado, entonces tocaría poner en negativo el valor de la *x*. Pero bueno, eso ya es cosa de que trasteen.

Ahora bien, miremos cómo se constituye la animación en este caso. Para ello se emplea el ***.animations.add***, que lo que hace es crear un nuevo objeto de tipo animado. Para ello primero se coloca el nombre del objeto (en este caso *walk*) y a continuación los *frames* que componen la animación. Muy importante saber cómo colocarlos. La forma más fácil es la que se muestra en el trozo de código, que es utilizando el método ***.generateFrames***, cuyos parámetros son una cadena, un valor de inicio, otro de finalización y un último que indica el total de cifras que puede alcanzar (4 en este caso, pues se enumera del 0001 al 0008).

Finalmente, en **update** hemos colocado todo lo referido al manejo del movimiento del muñeco mediante el valor de x del objeto *capguy*. Como se puede observar, *update* funciona como un bucle, de tal forma que a cada interacción va sumando 3 o restaurando la cifra de partida del machanguito.

**Los putos sonidicos.**

Para implementar sonidos en nuestro juego toca hacer un par de cosas en nuestro juego. De primeras tocará cargar los audios. Eso se hace siguiendo la instrucción siguiente:

**game.load.audio('blaster', 'assets/audio/SoundEffects/blaster.mp3');**

Por supuesto, deberá ir en el *preload* del juego.

Luego, en el *create* tocará colocar un *add* para cada no de los elementos de sonido, tal y como muestra esta línea de código:

**blaster = game.add.audio('blaster');**

Acto seguido tocará colocar el *setDecodedCallback*, método que nos indica que el sonido está preparado. Funciona de la siguiente manera:

**game.sound.setDecodedCallback([ explosion, sword, blaster ], start, this);**

Luego tocará llamar a una función que maneje los eventos de los botones, que será *start*.

**keys = game.input.keyboard.addKeys({ blaster: Phaser.Keyboard.ONE, explosion: Phaser.Keyboard.TWO, sword: Phaser.Keyboard.THREE })**

En *start* tenemos esta bonita línea, que es la que indica qué sonido está asociado a qué tecla… y básicamente así funciona esto, en verdad.

**Implementar botones**

Para implementar un botón, se sigue la siguiente estructura:

**game.add.button(x, y, ’image’, handler);**

Donde *x* e *y* corresponden a la posición del botón en la pantalla, *image* al dibujín empleado para la parte gráfica y *handler* a la función que debe utilizarse cuando sea pulsado el coso.

**Hacer que corra un vídeo**

La estructura que sigue es la siguiente:

**game.load.video(...);**

Mediante esta función se carga el archivo de vídeo.

**video=game.add.video(...);**

Con esto se crea el objeto de tipo vídeo como tal.

**video.addToWorld(...);**

Mediante esto se añade al mundo del juego.

**video.play();**

Y ya esto lo que hace es poner en activo el vídeo.

**video.stop();**

Bueno, un poco evidente… pero esto para el vídeo.

***Tweens***

*Inbetweening* o *tweening* es el proceso mediante el cual se generan los frames intermedios que hay entre dos imágenes. Para realizar esto se ha de tener en cuenta aspectos fundamentales correspondientes a las propiedades de los elementos, tales como su rotación, opacidad, tamaño, color,...

El ejemplo clásico es el siguiente:

**tween\_mute = game.add.tween(buttonMute);**

**tween\_mute.to({ y: 0, angle: 90}, 500);**

**tween\_mute.to({ y: buttonPlay.y, angle: 0 }, 500);**

**tween\_mute.start();**

La primera línea lo que hace es constituir el *tween*. Para ello se emplea el *add.tween()*, donde su único parámetro es el objeto sobre el que se van a aplicar los cambios.

En la segunda, mediante el uso del *to* se especifica el cambio, que en este caso es en *y* y en *angle*, así como la duración de cada paso (que en este caso es de 500 milisec, ½ segundos).

La tercera línea es más de lo mismo y en la última lo que se hace es iniciar el *tween*.

**Partículas**

Las partículas son elementos muy chicos que, en gran cantidad, producen efectos visuales. Para hacer partículas en *Phaser* se suelen seguir los siguientes pasos:

**// 1. Define an emitter at postion (x,y)**

**// and given width w and height h**

**emitter = game.add.emitter(x,y,w,h);**

**// 2. Set the images of the particles**

**emitter.makeParticles([’star’,’ball’]);**

**// 3. Define properties of the particles:**

**// gravity, speed, opacity, scale, etc.**

**emitter.setAlpha(0.6, 0.8); // transparency range**

**// 4. Start emitting**

**emitter.start(true, // all particles at once**

**1000, // lifespan of each particle**

**null, // ignored in "explode" mode**

**15); // # particles in this burst**

La primera línea define *emitter*, que será nuestro cuadradito invisible donde se ejecutarán las acciones propias de la emisión de las partículas.

La segunda crea las partículas a partir de imágenes que tú le des (básicamente tú le das la etiqueta del objeto).

La tercera altera los valores de transparencia de las partículas… pues para dar efectos guapos y todo eso.

Ya luego en la cuarta se empieza la emisión mediante el *start*.

*Y ya estaría esta práctica de mierda. Suscríbanse y denle a like para más vídeos como éste.*