



DOCUMENTACIÓN SPELUNKY

Entrega cocos2d

Javier Díez García – UO250728

Víctor Suárez Fernández – UO250790

David Ferreiro Fernández – UO250757

INTRODUCCIÓN

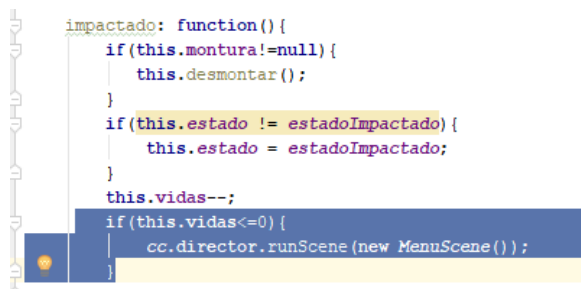
Este es un breve informe de cómo se juega y cómo se creó nuestra propuesta de juego en Cocos2D y Chipmunk. Para jugar, los controles son:

- Desplazamiento: Flechas de dirección
- Trepar: Flecha direccional estando sobre una cuerda/escalera
- Desplazar cámara: Teclas WASD
- Salto: Barra Espaciadora
- Disparo: X una vez obtenido el arma
- Bomba: Z Si se han recogido bombas
- Agachar: Flecha de dirección hacia abajo.

Para pasarnos el juego, debemos:

1. Obtener las 3 llaves de cada nivel.
2. Llegar a la puerta de fin de nivel.
3. Seleccionar mejora de habilidad.
4. Superar el nivel 3 (último).

El juego tiene muchos enemigos y trampas por lo que es complicado de jugar, recomendamos para poder probarlo a fondo comentar en la clase Jugador.js las líneas marcadas:



```
impactado: function() {  
    if (this.montura !== null) {  
        this.desmontar();  
    }  
    if (this.estado !== estadoImpactado) {  
        this.estado = estadoImpactado;  
    }  
    this.vidas--;  
    if (this.vidas <= 0) {  
        cc.director.runScene(new MenuScene());  
    }  
}
```

Además, recordamos que al ser las paredes verticales consideradas por el juego como parte del suelo, se puede subir una pared en vertical saltando si se mantiene el contacto del jugador con la pared.

El juego está desplegado permanentemente en: <https://javicodema.github.io/SpelunkyCocos2D/>

Tenemos unos cuantos errores derivados del uso del motor físico Chipmunk, como podría ser que el personaje entrara en una de las paredes, el propio crasheo del juego al contactar el jugador con una cuerda... La mejor solución sería recargar la página y empezar de nuevo.

Funcionalidades

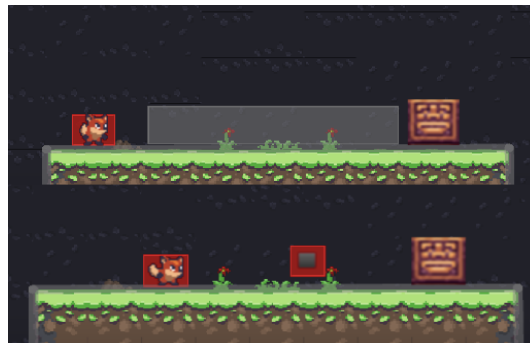
Trampas

Las trampas se colocan por medio del creador de mapa Tiled y en cocos2d por los medios habituales.

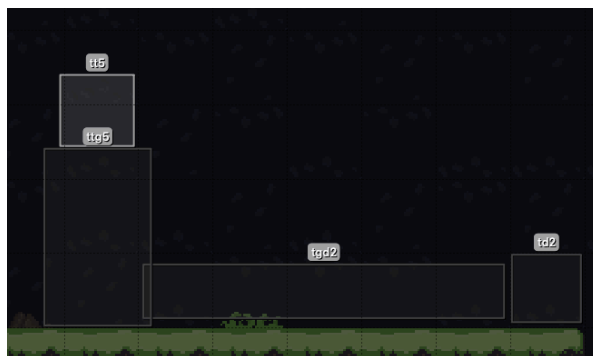
Las trampas tienen una función con la cual se activan o aplican un efecto. La activación se suele realizar al detectarse la colisión del jugador con la trampa.

```
//Colisiones trampa ralentizar
this.space.addCollisionHandler(tipoJugador, tipoTrampaRalentizar,
    this.colisionTrampaRalentizar.bind(this), null, null,
    this.finColisionTrampaRalentizar.bind(this));
```

Existen dos casos en los que la activación no se realiza por contacto directo con la trampa: TrampaDisparo y TrampaCaerEncima, en estos casos la activación se realiza a través de un trigger colocado en otro lugar.



La colocación de estos triggers se realiza también a través del editor Tiled, no obstante, siguiendo una nomenclatura en los nombres tanto de la trampa como de los triggers para posteriormente poder emparejarlos por medio de JavaScript. En la imagen, el prefijo es para determinar el tipo de trampa, el número es que se utiliza posteriormente para el emparejamiento.



Menú inicial y HUD

Para la creación del menú inicial bastó con añadir un MenuLayer en app.js que tenga un botón para comenzar a jugar y al que se redirige al jugador cada vez que pierde.

Para el HUD de jugador, añadimos una serie de etiquetas en ControlesLayer para los puntos, la vida, las bombas y las llaves que el jugador lleve en cada momento, y con unas funciones sencillas permitimos a otras partes del código actualizar estas etiquetas cuando deba.

Gestión de la puntuación y sistema de vida.

Para el sistema de puntuación, se suma a una variable guardada en el jugador con la puntuación 10 puntos cuando un disparo elimina a un enemigo y 50 puntos cuando se recoge un recolectable opcional. Estos opcionales están representados en la clase Opcional.js con el Sprite de un diamante que se recoge al pasar el jugador por encima de ellos. Cada vez que se actualiza la puntuación se llama a ControlesLayer para actualizar la etiqueta.

Para gestionar la vida del jugador, este parte con 5 vidas (se guardan en una variable en Jugador.js) y cada vez que se llama a la función impactado del jugador, este pierde una vida y si se queda sin ellas muere y vuelve a empezar el juego en el menú. A esta función se le llama por ejemplo cuando un disparo impacta con un jugador, un enemigo colisiona con el jugador (le sumamos una vida al jugador cuando salta encima de un enemigo, ya que a pesar de que elimina al enemigo colisiona antes con él, por lo que le resta una vida) ...

Arma y disparo del jugador.

Tras crear una clase Disparo a la que añadimos una orientación para indicarle en qué dirección x debe ir, en la GameScene tendremos un array de disparos al cual en actualizar() estaremos dando constantemente una velocidad $v_y=0$ para que no le afecte la gravedad.

Además, creamos una serie de collisionHandlers para ver cómo afecta el choque de un disparo con suelos o escaleras(desaparece) o a un enemigo o un jugador (elimina y resta una vida respectivamente).

Tras esto, cargamos del mapa el Arma y con un collisionHandler del arma y el jugador, llamamos a una función armar del Jugador para que este sepa que tiene un arma (se guardará un objeto vacío para indicarle que tiene un arma) y le pondremos un delay de disparo para que no pueda disparar a cada tick del juego.

Ya solo faltaría añadir en GameScene que cada vez que se active el control de atacar, se compruebe que tiene arma y ha pasado el intervalo de disparo para poder crear un nuevo disparo.

Creación de enemigos

Existen tres tipos de enemigos:

Los EnemigosPatrulla simplemente van de un lado a otro en el nivel de altura al que se les sitúa y cada vez que detectan que ya no hay suelo a un lado, cambian su orientación para continuar su movimiento infinito.

Los EnemigosTiradores comprueban si el jugador está a un rango determinado y a su misma altura y disparan si es que está en ese rango en la orientación hacia la que está el jugador. El disparo se genera en condiciones parecidas a las del jugador (salvo que este enemigo siempre está armado, luego no hace falta comprobar que tiene un arma).

Los EnemigosPerseguidores comprueban el mismo rango que los tiradores, pero en vez de disparar, se lanzan en la dirección del jugador hasta que lo pierden de vista, en su código vemos que modifica su dirección x en función de la localización del jugador.

Para los enemigos, es común una shape sobre su cabeza, llamada ShapeArriba, que es la que comprueba si colisiona con otra shape (ShapeAbajo) creada en el jugador. Cuando estas dos shapes colisionan, es cuando el jugador elimina a un enemigo saltando sobre él. Cada enemigo tiene apariencias diferentes, los patrullas y los perseguidores además tienen diferentes animaciones de movimiento.

Gestión de monturas

Para las monturas, añadimos un nuevo .js, Montura, y añadimos de nuevo un collisionHandler para que cuando el jugador pase sobre ella se monte, llamando a la función montar(). Esto provoca que las animaciones Idle, caminando y saltando se modifiquen por unas en las que el jugador está sobre la montura. Además, cuando está montado, el jugador adquiere unas bonificaciones de salto y de velocidad. El jugador perderá su montura cuando reciba un impacto (aunque sea saltando sobre un enemigo).

Escaleras

A las escaleras se les dota de body y shape, cuando se detecta una colisión del jugador el estado del jugador pasa a “trepando”, en el que el jugador puede desplazarse libremente por el cuerpo de la escalera.

Cuerdas

Funcionan del mismo modo que las escaleras, salvo que estas van rotando. Al detectar una colisión se crean dos elementos “Joint” entre el jugador y la cuerda, según la API de chipmunk esto hace que la distancia entre ambos objetos se mantenga constante, pero en el juego no se mantiene, simplemente ralentiza la caída. Creemos que esto es producto de la gravedad

aplicada pero no hemos podido solucionarlo. Para evitar caer se puede desplazar el jugador con las flechas junto a la rotación de la cuerda y moviéndolo ligeramente hacia arriba.

```
collisionCuerdaJugador: function(arbitrer, space){  
    if(this.jugador.puntoAnclaje == null){  
        this.jugador.puntoAnclaje = pinjoint;  
        collision_p = arbitrer.contacts[0].p;  
        var pinjoint = new cp.SlideJoint(arbitrer.body_a, arbitrer.body_b,  
arbitrer.body_a.p, collision_p, 0, 2);  
        var pinjoint2 = new cp.PivotJoint(arbitrer.body_a, arbitrer.body_b,  
collision_p, collision_p);  
        pinjoint.errorBias = 0.99;  
        pinjoint2.errorBias = 0.99;  
        space.addPostStepCallback( () => {  
            space.addConstraint(pinjoint)  
            space.addConstraint(pinjoint2)  
        } );  
        this.jugador.saltosAcutales = 0;  
        this.jugador.trepar();  
        this.jugador.puntoAnclaje = pinjoint;  
        this.jugador.puntoAnclaje2 = pinjoint2;  
    }  
},
```

Bombas

El jugador podrá recoger bombas del suelo, una vez disponga de ellas podrá soltarlas El jugador podrá recoger bombas pulsando la tecla Z, esto generará una nueva bomba en el mapa, que tras un intervalo de tiempo “explota”, creando dos disparos uno hacia cada lado que dañará tanto a enemigos como jugador, estos disparos al contrario que los de las armas o enemigos tienen un rango fijo.

Llaves y Gestión de Niveles

El jugador deberá recoger tres (3) llaves repartidas por el nivel para desbloquear la puerta que conduce hasta el siguiente. Al igual que los recolectables opcionales, las llaves tienen shape y body y un CollisionHandler para colisionar con el jugador. Cuando el jugador colisiona con la llave, esta se elimina y aumenta el contador de llaves del jugador.

Al colisionar el jugador con la puerta, si se han obtenido las 3 llaves de nivel (en realidad con más de 3 llaves funciona, pero solamente hay 3 por nivel), se cargará una nueva layer mostrando la puntuación del nivel y permitiendo al jugador elegir entre 3 mejoras:

- Un bonificador de velocidad
- Una vida extra
- Un arma

Tras elegir su mejora, se carga el siguiente nivel, creándose todo desde cero y se copian las estadísticas del jugador desde una variable que contiene el jugador en el momento que colisionó con la puerta.