Evaluación Cuatrimestral Adaptación del Ambiente de Trabajo

**Lautaro Cerruti** y **Facundo García Mata**, alumnos de 6to Informática

Link del trabajo: <https://github.com/LautaroCerruti/AAT.git>

Para completar la consigna de la evaluación decidimos desarrollar un personaje. Todas las funciones relacionadas al mismo se encuentran dentro de una clase que también creamos, llamada **GranDad**.  *GranDad* puede *correr*, *saltar*, hacer *un doble salto* y *pegar*. También cuenta con una animación de cuando está parado llamada *Idle* y otra en donde se desliza mientras desacelera.

Para facilitar el trabajo entre los dos utilizamos un sistema de control de versiones llamado **GitHub**.

Partimos desde la implementación de un ejemplo de [phaser.io](https://phaser.io/tutorials/making-your-first-phaser-3-game/phaser3-tutorial-src.zip), más específicamente, la parte 7.

Lo primero que hicimos fue crear una nueva clase con la siguiente estructura donde integramos varias cosas:

class GranDad {

   constructor () {...}

   playerconstructor (...) {...}

  modifvelizq () {...}

  modifvelder (){...}

  corrizq (...) {...}

  corrder (...) {...}

  saltar (...) {...}

  idle (...) {...}

   golpe (...) {...}

}

* el constructor: En este enunciamos los atributos que se utilizan en la clase, como por ejemplo, banderas o contadores a los que les asignamos valores iniciales, y además un atributo llamado **player**. A este lo definimos aquí dentro para luego usando una función llamada **playerConstructor** fuera del constructor asignarle distintos parámetros. Esto se expresa en código de la siguiente manera:   
  constructor() {

this.player;

this.dbljmp = false;

this.banim = false;

this.salto = false;

this.contadoractivo = false;

this.contador = 0**;**

          }

* función playerConstructor: Para empezar, esta función toma como parámetros  la posición X e Y donde será creado el jugador, el bounce que le queremos asignar al personaje (preferiblemente 0.0), un cuarto parámetro que es this, el cual hace referencia a phaser en sí, y por último el nombre que escribimos cuando cargamos el sprite del jugador. Esta función se tiene que ejecutar en la función *create* del juego, ya que el crear las animaciones si la ponemos fuera no funcionaria. En resumen, esta función es la que cumple el rol de constructor para el atributo *player*. La primer parte de esta es:

playerConstructor(X, Y, bounce, that, spritesheet) {  
 this.player = that.physics.add.sprite(X, Y, spritesheet);

this.player.setBounce(bounce);

this.player.setCollideWorldBounds(true);

that.anims.create({

key: 'left', frames:that.anims.generateFrameNumbers(spritesheet,{start:0,end:5}),

frameRate: 10,

repeat: -1

})

* métodos privados: Si bien no existe una forma de declarar un método privado en javascript nosotros creamos dos funciones que podrían considerarse privadas  ya que son llamadas y ejecutadas desde otras funciones de la clase, pero nada nos impide ejecutarlas desde fuera. Por esto, uno podría llamar a nuestros métodos “privados” desde la función **update** y ver que estos funcionan de todas formas.  
  Estas funciones son **modifvelizq** y **modifvelder** que se llaman desde **corrizq** y **corrder** respectivamente. Un ejemplo de esta es:

modifvelizq() {

if(this.player.body.velocity.x > -400){

if(this.player.body.velocity.x > 0){ this.player.body.velocity.x <= -10;}

else this.player.body.velocity.x -= -3;}

}

* métodos públicos: Todas las funciones que están declaradas dentro de la clase son métodos públicos. Estas toman como parámetros inputs de teclado, y con respecto a los mismos, ejecutan su código. A continuación se muestra a la función **corrder**:

corrder(presionando) {if (presionando) {

if (this.contadoractivo && this.contador < 35)

this.contador++;

                  else {

this.modifvelder();

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

this.player.anims.play('right', true);

else

if (this.player.body.velocity.y >= -100)

this.player.anims.play('caidad');

else

this.player.anims.play('saltod');

this.contador = 0;

this.contadoractivo = false;

                   }

}

              if (this.player.body.touching.down) this.salto = false;}

Como vemos en **corrder**, si la persona está presionando un botón de desplazamiento horizontal y el contador que inhabilita las acciones no está activado, se ejecuta la función **modifvelder**, y dependiendo de si está tocando el suelo o si está cayendo o saltando, reproduce diferentes animaciones. La función **corrizq** es muy parecida, tanto que solo cambia la función que ejecuta para modificar la velocidad y las animaciones que reproduce.

En la función **saltar** nuevamente, si se está presionando, y el contador está desactivado, se pregunta si el jugador está tocando el piso.  Si lo está, salta y la variable *dbljmp* cambia su valor atrue, permitiendo el doble salto. Si el jugador salta en el aire, y tiene el doble salto disponible, lo realiza y se cambia la variable *dbljmp* a false. Esta función tiene otra sección que siempre se ejecuta, que es que si la persona está en el aire y esta nunca saltó, tiene que tener la posibilidad de hacer el doble salto.

En la función **golpe** nos fijamos si el jugador está tocando el suelo, si presionó el botón y si la velocidad es 0. En caso de que así sea, dependiendo de qué lado estaba mirando, reproduce la animación de golpe y activa el contador. Este contador se ejecuta una vez por frame y cuenta el largo de la animación del golpe, ya que si no lo hacíamos de esta manera se nos pisaba la animación con otras que intentaba reproducir.

La función **idle** es la que realiza acciones cuando el jugador no está presionando ningún botón. Esta se encarga de reproducir las animaciones correspondientes y de frenar en caso de que el personaje se haya estado moviendo.

* instancia de clase: Finalmente, para crear variables de esta clase, escribimos lo siguiente:

let jugador = new GranDad();

let jugador2 = new GranDad();

Luego ejecutamos en la función **create** el **playerConstructor** en cada una:

jugador.playerConstructor(10, 450, 0.0, this, 'dude');

jugador2.playerConstructor(350, 450, 0.0, this, 'dude');

Y en **update** ejecutamos todas las funciones públicas, con el input correspondiente de cada jugador:

jugador.corrizq(cursors.left.isDown);

jugador.corrder(cursors.right.isDown);

jugador.idle(cursors.left.isDown, cursors.right.isDown, cursors.space.isDown);

jugador.golpe(cursors.space.isDown);

jugador.saltar(cursors.up.isDown);

jugador2.corrizq(a.isDown);

jugador2.corrder(d.isDown);

jugador2.saltar(w.isDown);

jugador2.idle(d.isDown, a.isDown, j.isDown);

jugador2.golpe(j.isDown);

**Principales Desafíos**

Como principal desafíos encontramos el manejo de animaciones del personaje, ya que nos pasaba que había errores o algunas animaciones que nunca se reproducían y esto era porque otra se estaba reproduciendo arriba. Esto se puede ver en el caso del *golpe*, que cuando lo accionábamos se reproducía solo el primer frame de su animación. Esto ocurría porque la de *Idle* se reproducía todo el tiempo y por lo tanto no le daba lugar a la del *golpe*. La solución que encontramos fue poner el contador que evitara cualquier acción durante el tiempo que durara la animación de golpe. Otra cosa que se presentó como una dificultad fue el hecho de no haber planificado todas las acciones desde un comienzo. Esto generó que pensemos las cosas que podíamos agregar sobre la marcha, haciendo que el código se complejice cada vez más y sea más difícil de entender.

**Mejoras no Implementadas**

Podríamos haber implementado un collider para el golpe, pensamos en un pequeño objeto que se generara cada vez que apretamos el botón, y lo que colisione con las cosas sea ese objeto y no todo el personaje. Este cambio le daría más realismo al golpe. De la mano de esto también pensamos en una animación de *stuneo* que se activaría cada vez que un *GranDad* golpeara al otro.

También pensamos algunas funcionalidades varias como poder correr más rápido apretando Shift, hacer un golpe aéreo y un sistema que al concatenar saltos haga efectos diferentes como saltar más alto.

También pensamos en poner una forma de personalizar más las funcionalidades como la posibilidad de elegir si se quiere acelerar y frenar deslizándose o no, la velocidad de movimiento, la gravedad del entorno entre otras cosas.