

Punch Man

Video game for Mobile and Desktop

Objetivo de la presentación

Hablaremos sobre las herramientas necesarias y su utilización para crear un videojuego, así como también analizaremos los aspectos relevantes a considerar cuando queremos iniciar un proyecto de éstas características..

Introducción 1/3

Comercialización.

Admob

AdMob es una plataforma publicitaria, con la cual se puede generar dinero a través de la publicación de aplicaciones gratuitas.

Codecanyon

Codecanyon se trata de una página de comercialización de Software, donde se puede vender el código de las aplicaciones.

Introducción 2/3

Engine.

Antes de iniciar, debemos definir que es lo que queremos lograr.

- Juego 2d vs Juego 3d.
- Engine o Framework.
- Qué Engine o qué Framework.

Introducción 3/3

Videogame.

Un videojuego se trata de un conjunto de aspectos que por sí solos no funcionan, sino que el conjunto de éstos aspectos es el encargado de brindar experiencias al usuario.

- Programación.
- Arte.
- Música.
- Historia.
- Diseño del juego(o Game Design).

Herramientas

Todas las herramientas utilizadas son open source.

- Libgdx : framework encargado de manejar la conexión entre el usuario y el videojuego.
- Box2d : motor de físicas sobre el cual será construido el videojuego.
- Tiled : generador de mapas para desarrollo de niveles.

División en secciones

Dividiremos la presentación en múltiples secciones con el fin de poder aprender más eficazmente las reglas que se rigen en el mundo que crearemos.

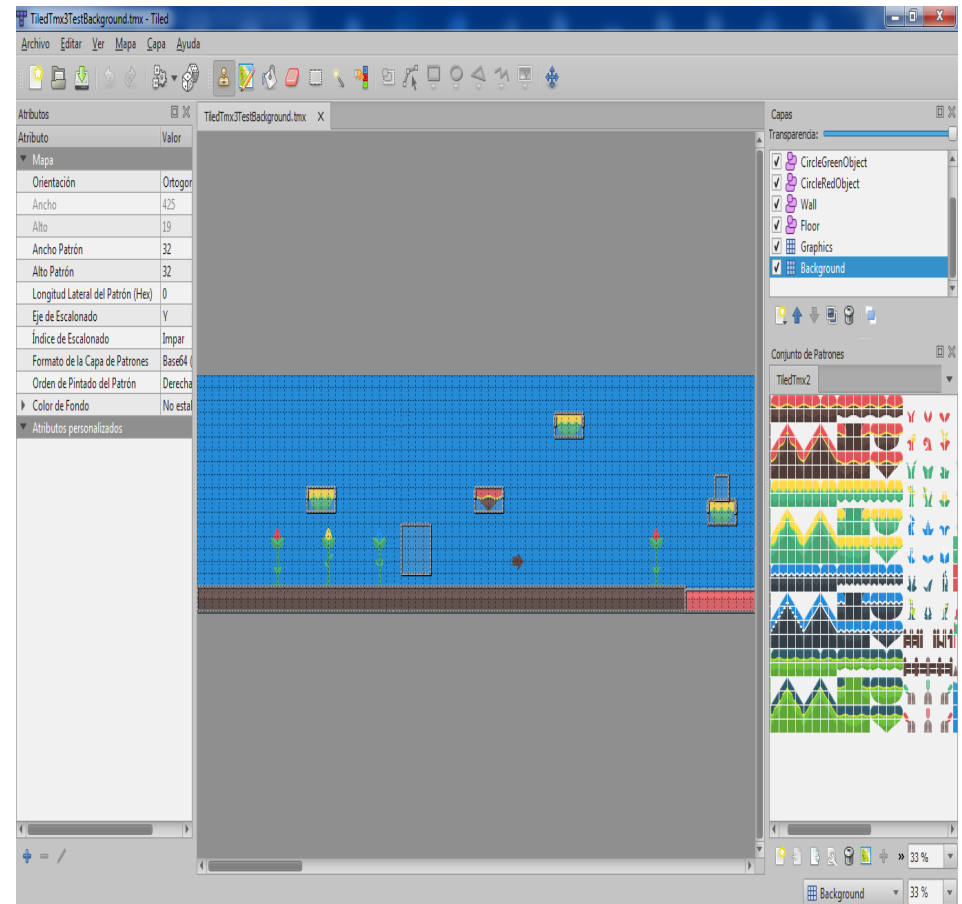
- Arte del juego.
- Física del juego.
- Lógica del juego.
- Interfaz del juego.

Arte del juego 1/7

Tiled

Mediante la utilización de tiles(cuadraditos 16x16 píxels), podemos generar gráficamente el mundo de nuestro videojuego.

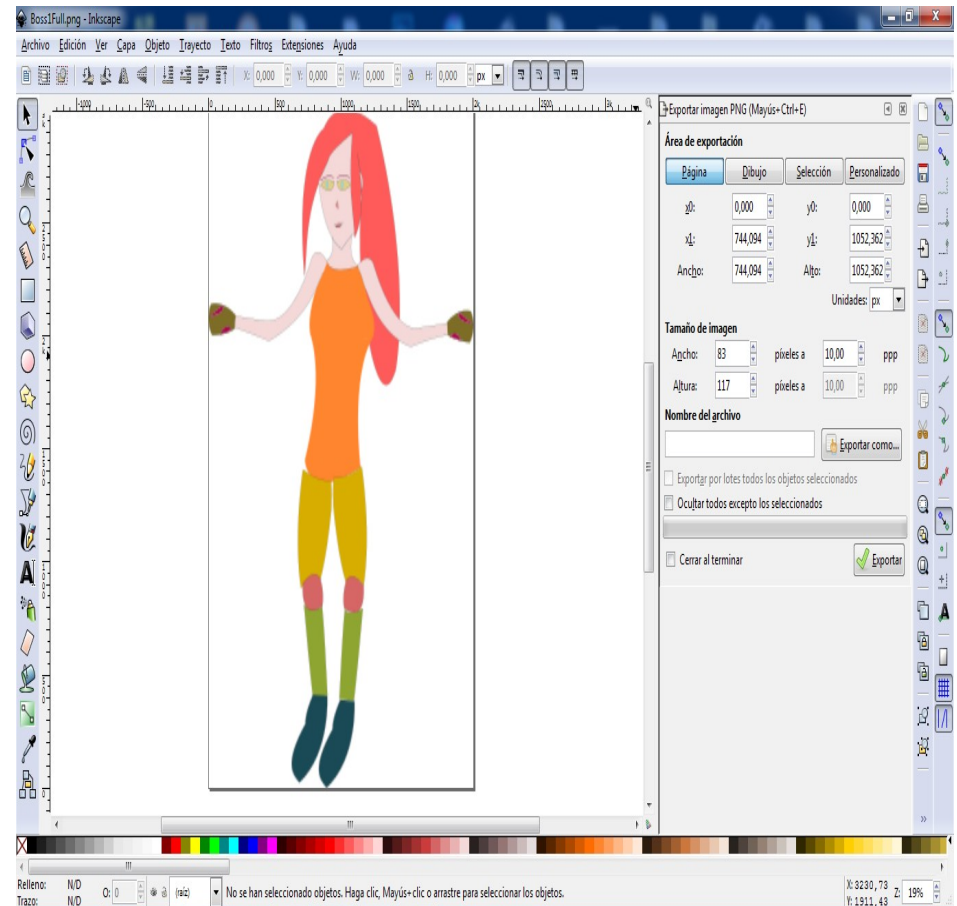
También podemos generar objetos vacíos, los cuales luego rellenaremos con enemigos.



Arte del juego 2/7

Inkscape

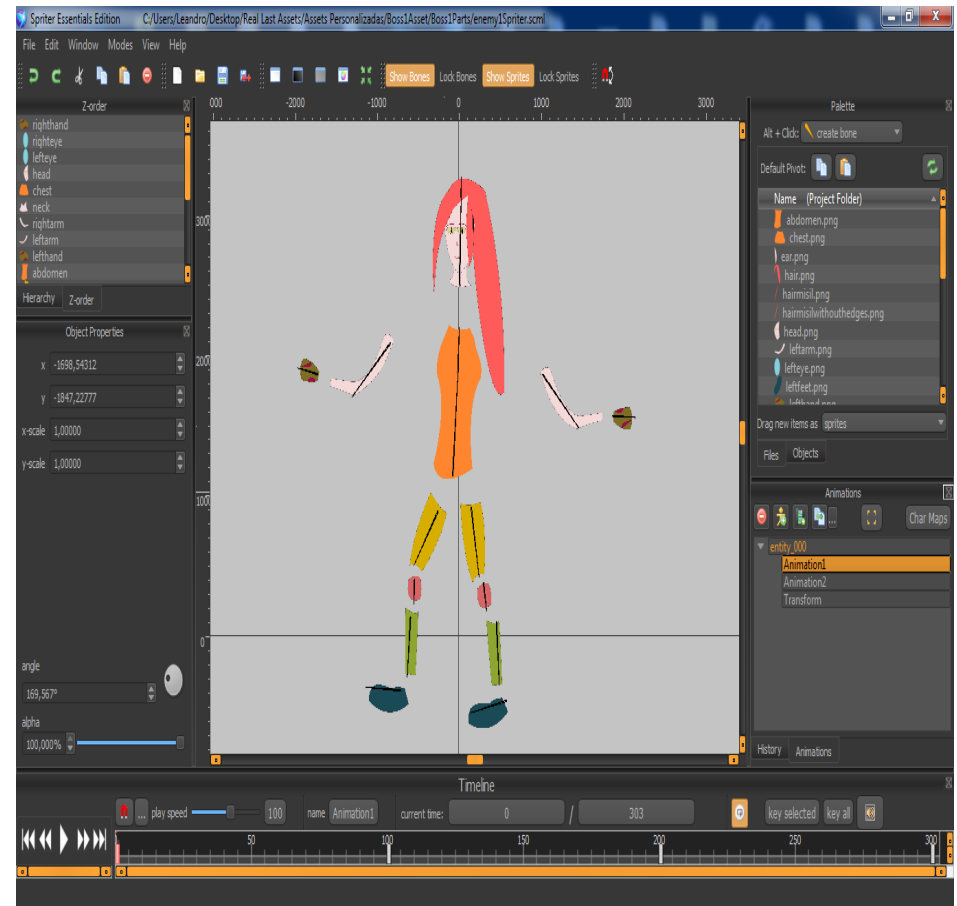
- Utilizando una técnica de gráficos vectoriales, podemos crear las imágenes que queramos incluir en nuestro juego.
- De esta manera, podemos hacer uso de múltiples gráficos para implementar lo que deseemos.



Arte del juego 3/7

Spriter

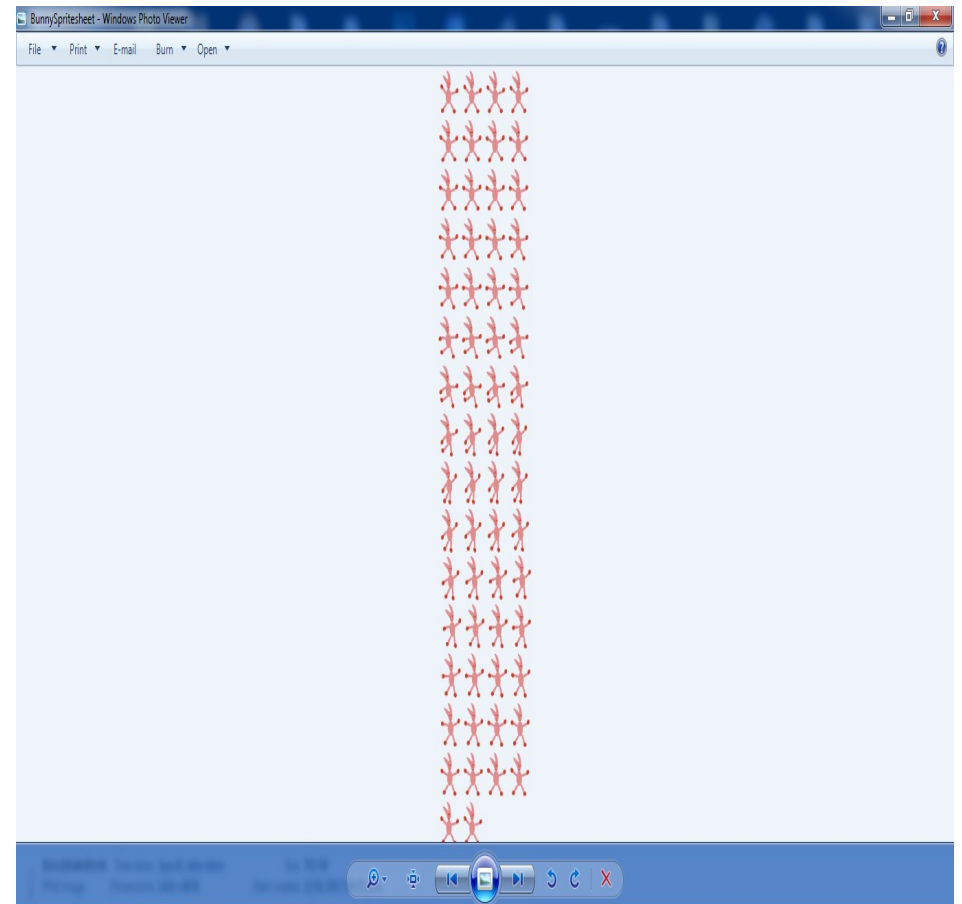
- Una forma muy útil de generar animaciones es, implementar una jerarquía de huesos, según la cual, cada imagen se encuentra unida a un hueso correspondiente.
- De esta manera, podemos exportar dichas animaciones en forma de Spritesheets, los cuales luego pueden ser utilizados en el videojuego para simular movimiento de los sprites.



Arte del juego 4/7

Spritesheet

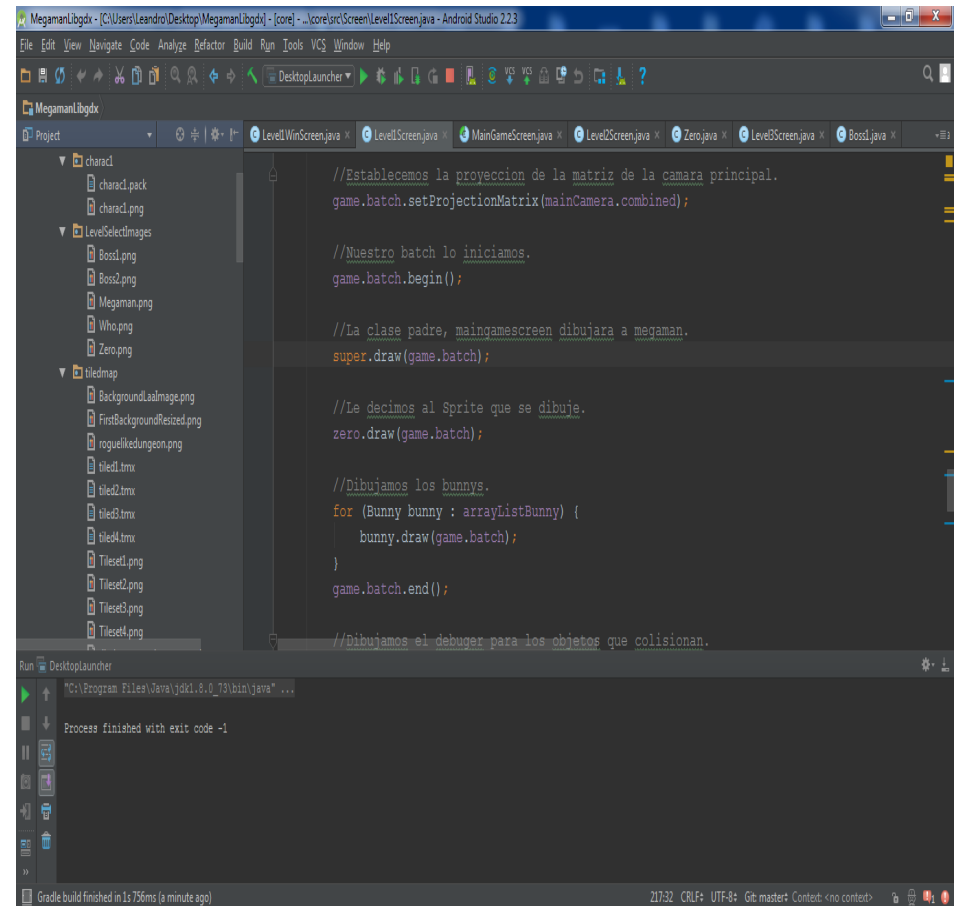
- Los spritesheets son imágenes de gran tamaño, que contienen dentro de si misma múltiples imágenes mas chicas, que corresponden a ciertas frames que se mostrarán en una animación dentro del juego.
- De esta manera se realizan las animaciones de juegos 2d, debido a que de lo contrario, cargar cada imagen por separado, es un gasto de recursos importante.



Arte del juego 4/7

Batch

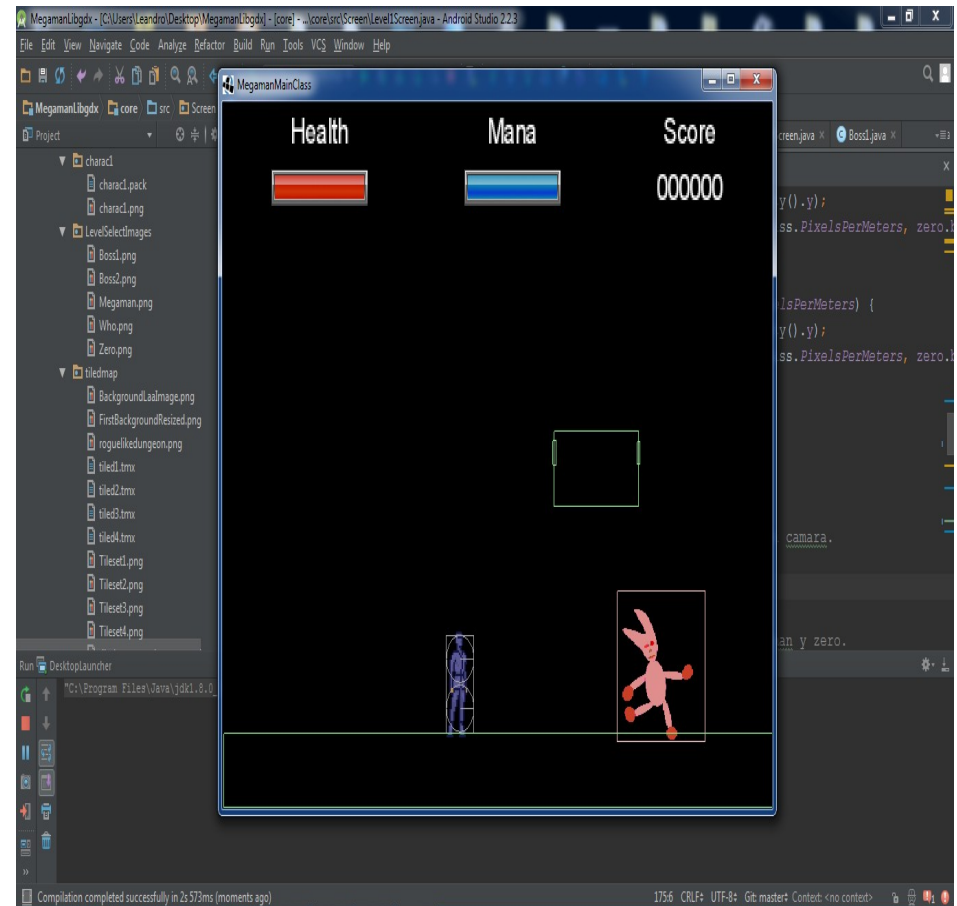
- Un batch se trata de una interfaz de Libgdx, mediante la cual se optimiza el proceso de dibujado de la aplicación.
- Para dibujar algo en un batch, primero se debe llamar a `batch.begin`, luego se dibuja lo que se requiere, y al finalizar se llama a `batch.end`.
- Solo debe haber un batch para todo el juego, debido a que se trata de un objeto muy pesado.



Arte del juego 5/7

Box2dDebugRenderer

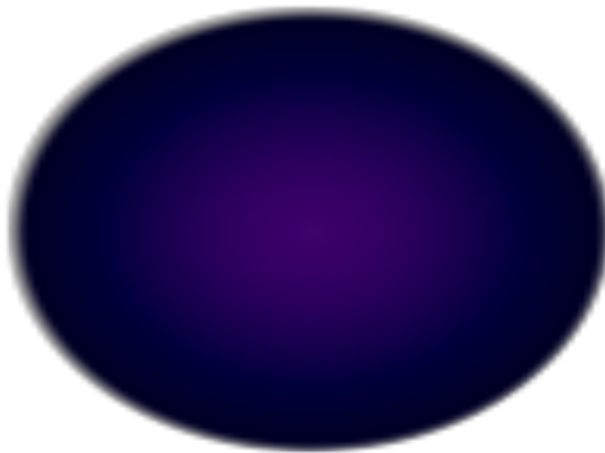
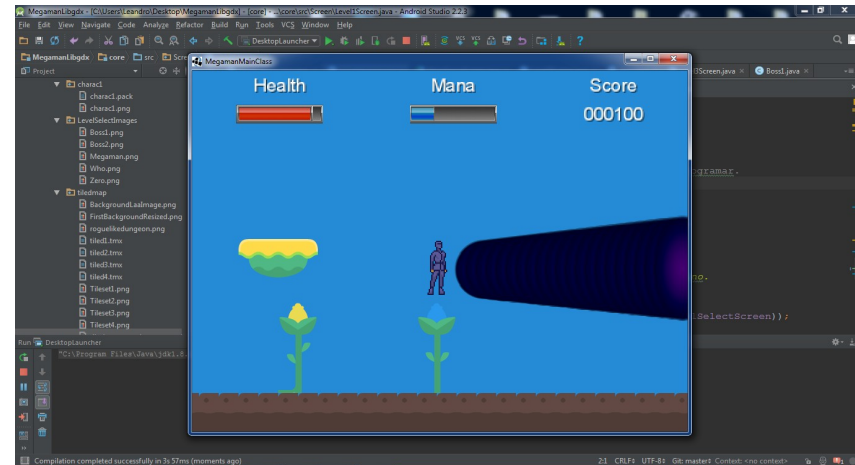
- El modo debug de Box2d es muy útil, ya que nos permite ver los verdaderos cuerpos de los objetos creados en el juego.
- De esta manera, podremos efectuar muchas comprobaciones, tales como el correcto funcionamiento de las colisiones, posiciones de los objetos, etc.



Arte del juego 6/7

Particles

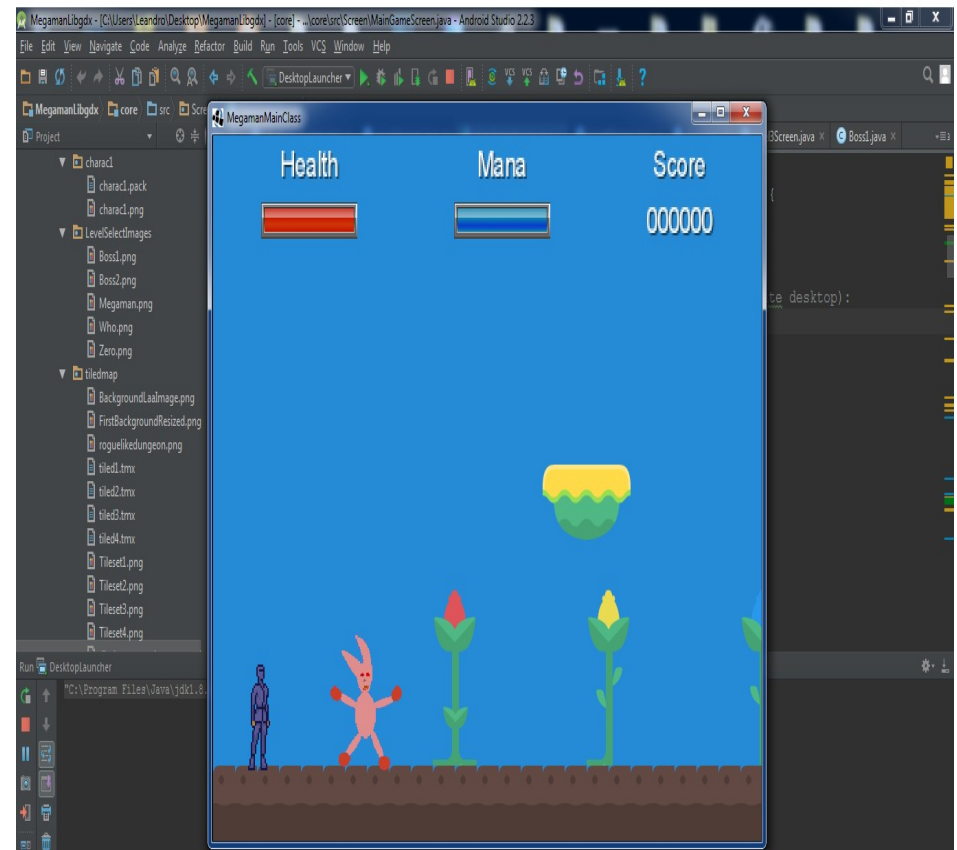
- En Libgdx lo llaman partículas, yo utilicé una técnica artesanal para convertir una esfera de color violeta, en un rayo que se desplaza cierta distancia.
- La idea es utilizar un sprite, y dibujarlo cierta cantidad de veces continuamente en cierta fracción de segundos, el algoritmo lo veremos en la sección lógica del juego.
- Esto nos permite crear numerosas variantes, solo limitadas por nuestra imaginación.



Arte del juego 7/7

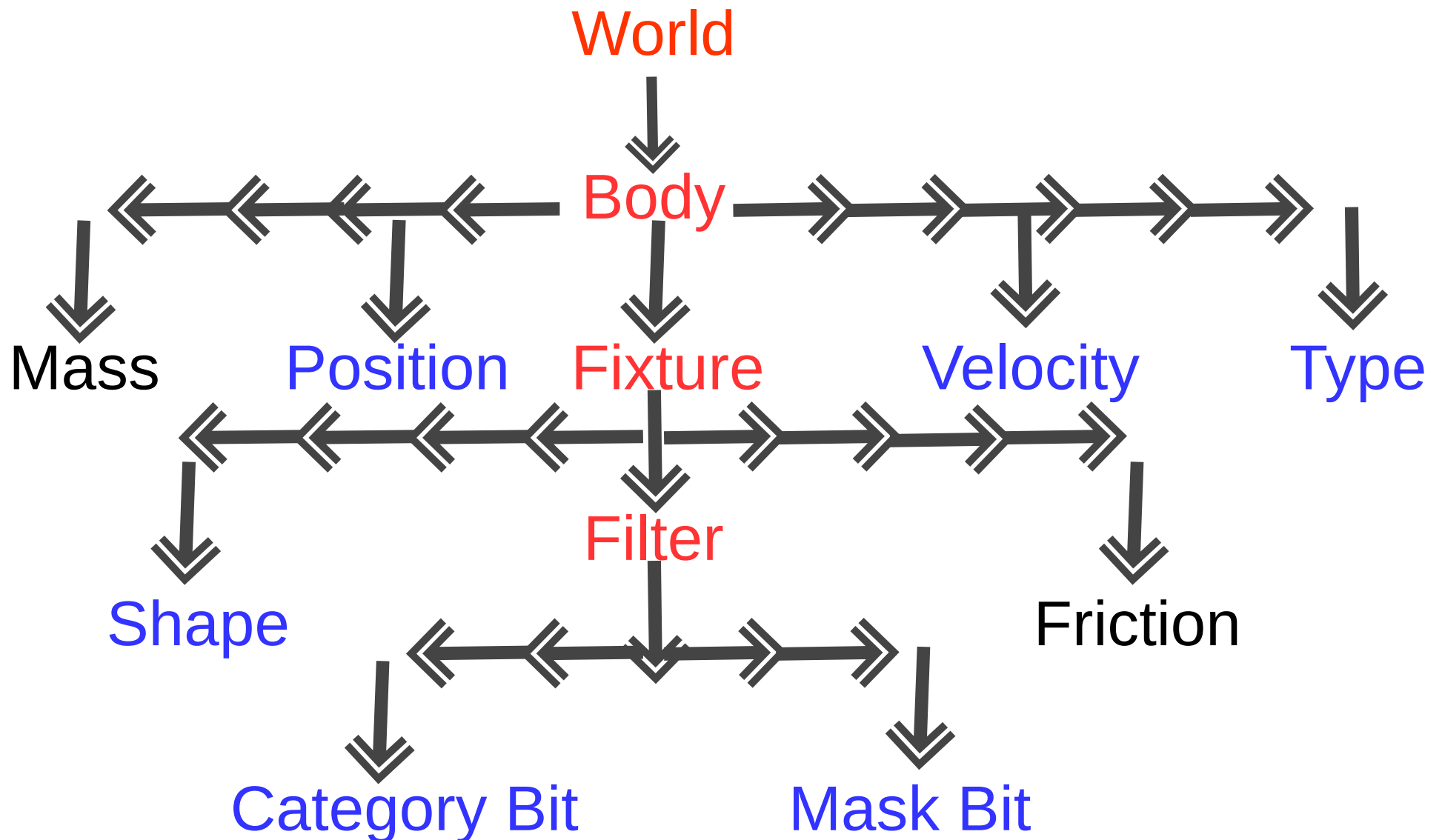
Dibujo de Sprites

- Lo que hacemos para dibujar los personajes del juego es lo siguiente: primero verificamos las posiciones de los cuerpos existentes en el mundo, y luego, en esas mismas posiciones dibujamos los sprites.
- Para las animaciones, realizamos lo mismo, solo que los sprites que mostramos cada segundo, van variando.



Física del juego 1/8

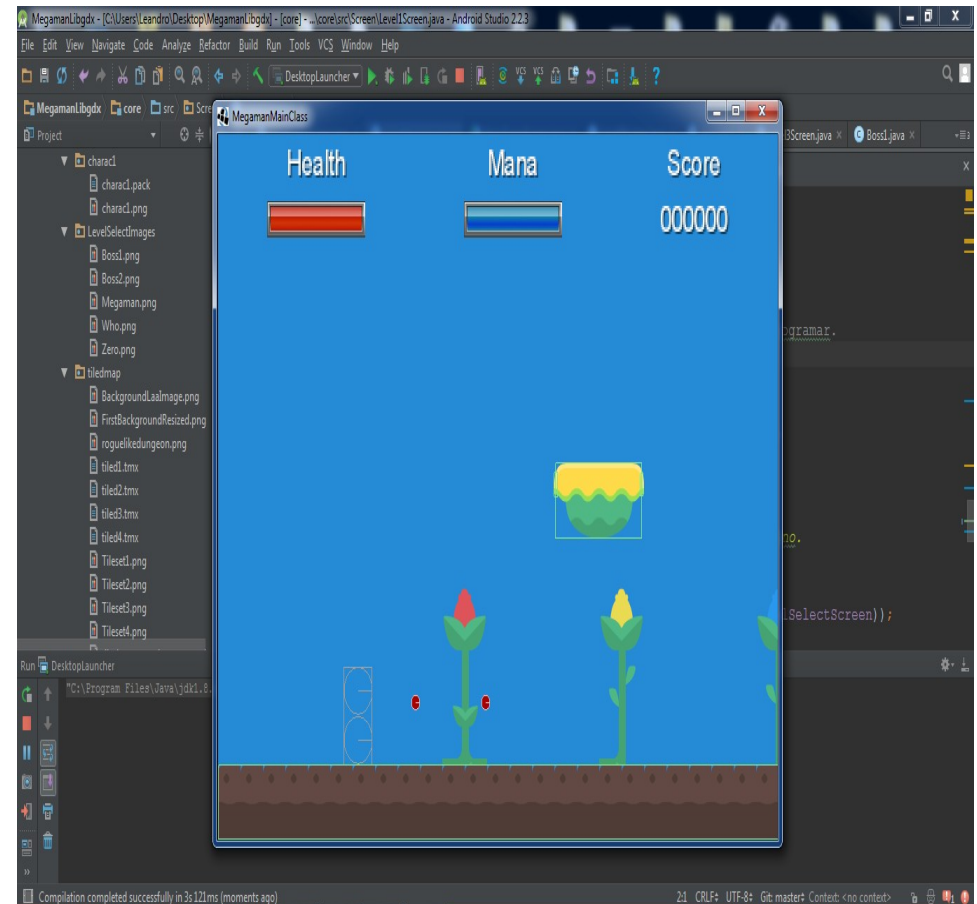
Intro to Box2d



Física del juego 2/8

FixtureShape

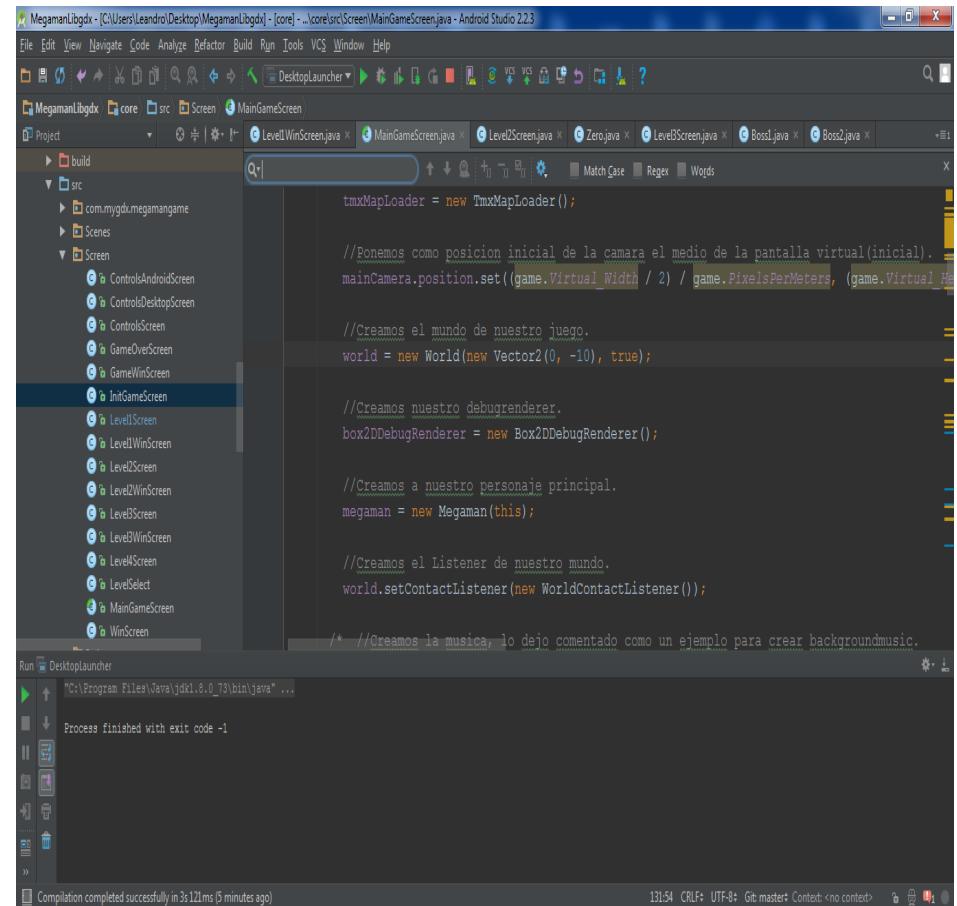
- Como se puede apreciar en el esquema anterior, el shape(forma) de un fixture será una forma geométrica la cual formará parte del cuerpo.
- Hay cuerpos que pueden tener un solo fixture, mientras que hay otros cuerpos que pueden tener más.
- Se puede apreciar en la imagen que, el cuerpo del personaje principal contiene 2 círculos y un rectángulo.



Física del juego 3/8

WorldGravity

- Al crear el mundo, podemos asignarle la gravedad que queramos. En nuestro juego, por ej, la gravedad está signada por el vector (0,-10).
- En cualquier momento del juego, nosotros podemos reajustar la gravedad como nos plazca.



Física del juego 4/8

Impulse/force

- El movimiento de cada cuerpo en nuestro juego, esta dado por impulsos o fuerzas.
- Un impulso es similar a una fuerza, pero la diferencia radica en que este se aplica de inmediato, mientras que la fuerza, se va aplicando de a poco(linealmente).
- Box2d se encarga de calcular las colisiones entre cuerpos, esto es, luego de un choque, box2d calcula las magnitudes finales a partir de las iniciales.

Física del juego 5/8

Category/Mask Bit Filter

- Un category bit es un short que permite asignar una identidad a un fixture en box2d.
- Un mask bit es un short que permite asignar un objeto a colisionar a determinado fixture en box2d.
- De esta manera, podemos controlar las colisiones en el mundo como queramos, podemos decidir si queremos que ocurra o no una colision modificando estos bits.

Física del juego 6/8

Category/Mask Bit Filter example

```
public final static short WALL_BIT = 1;  
public final static short FLOOR_BIT = 2;  
public final static short PERS1_BIT = 4;
```

Fixture Definition del personaje principal:

```
fixtureDef.filter.categoryBits = PERS1_BIT;  
fixtureDef.filter.maskBits = FLOOR_BIT | PERS1_BIT;
```

De esta manera, el personaje principal es identificado como PERS1_BIT, y puede colisionar con FLOOR_BIT y otros PERS1_BIT, pero no con WALL_BIT.

Física del juego 7/8

BodyType

Un cuerpo puede ser de 3 tipos distintos:

- Static Body: cuerpo que no puede moverse, y que no se calculan las físicas para posicionar el cuerpo, solo esta quieto.
- Kinematic Body: es un cuerpo que solo puede moverse siguiendo un patron fijo, y no es afectado por la gravedad.
- Dinamic Body: es un cuerpo cuyas propiedades se modifican con el movimiento, y es afectado por las fuerzas que se rigen en el mundo.

Física del juego 8/8

Sensor

- Un fixture puede ser un sensor, esto quiere decir, que puede servir para avisar cuando ocurre una colisión, sin modificar las físicas del juego.
- Se trata de una herramienta muy útil, ya que podemos implementarla de varias maneras diferentes y creativas, veremos ejemplos de esto en la sección Lógica del juego.
- Por ejemplo: los personajes enemigos simples tienen sensores para ver si colisionan con un proyectil.

Lógica del juego 1/16

Intro to Algorithms

Cada parte del juego está programada, lo que quiere decir que cada cosa que puede hacerse o no, está pensada con cierta lógica.

Esto quiere decir que tuve que implementar numerosos algoritmos para resolver numerosas situaciones diferentes.

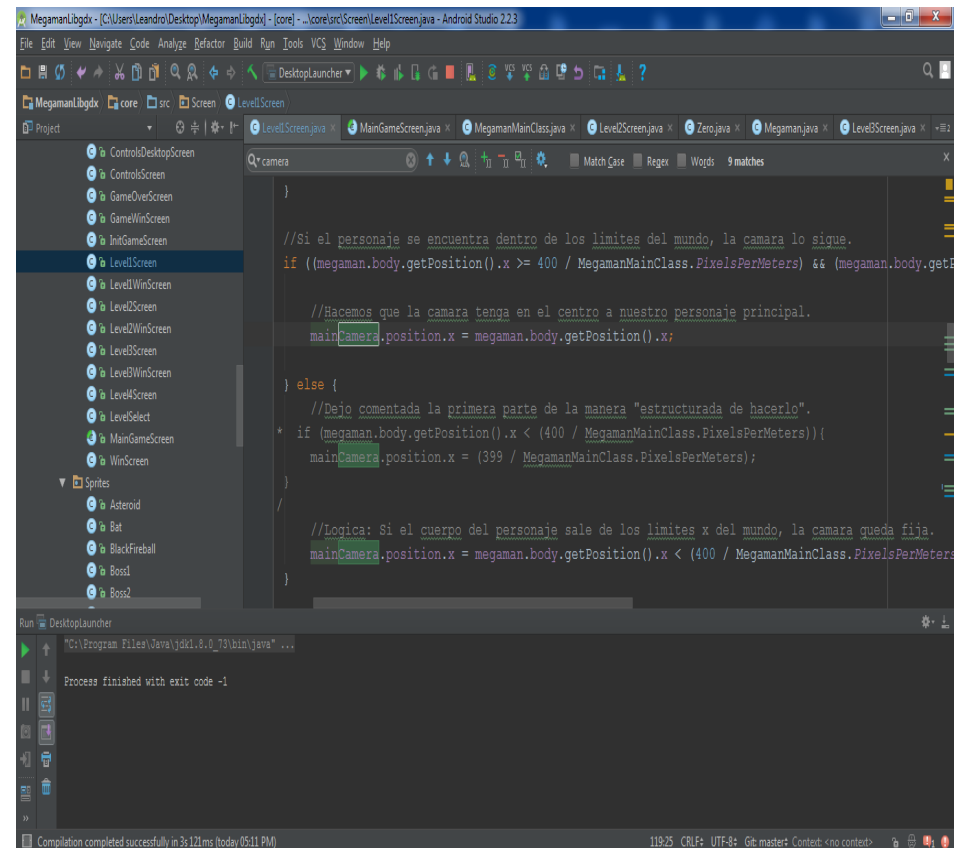
Voy a explicar los algoritmos que más me costaron o los más interesantes en el juego.

Lógica del juego 2/16

Camera Movement

La camara sigue al personaje principal, siempre y cuando este se encuentre dentro del mundo, si el personaje sale del límite del mundo, entonces la camara deja de seguirlo.

Si el personaje entra en la sección de la batalla final de cada nivel, entonces la camara se bloquea.



```

//Si el personaje se encuentra dentro de los limites del mundo, la camara lo sigue.
if ((megaman.body.getPosition().x >= 400 / MegamanMainClass.PixelsPerMeters) && (megaman.body.get...

//Hacemos que la camara tenga en el centro a nuestro personaje principal.
mainCamera.position.x = megaman.body.getPosition().x;

} else {
    //Dejo comentada la primera parte de la manera "estructurada de hacerlo".
    * if (megaman.body.getPosition().x < (400 / MegamanMainClass.PixelsPerMeters)){
        mainCamera.position.x = (399 / MegamanMainClass.PixelsPerMeters);
    }

    //Logica: Si el cuerpo del personaje sale de los limites x del mundo, la camara queda fija.
    mainCamera.position.x = megaman.body.getPosition().x < (400 / MegamanMainClass.PixelsPerMeters;
}

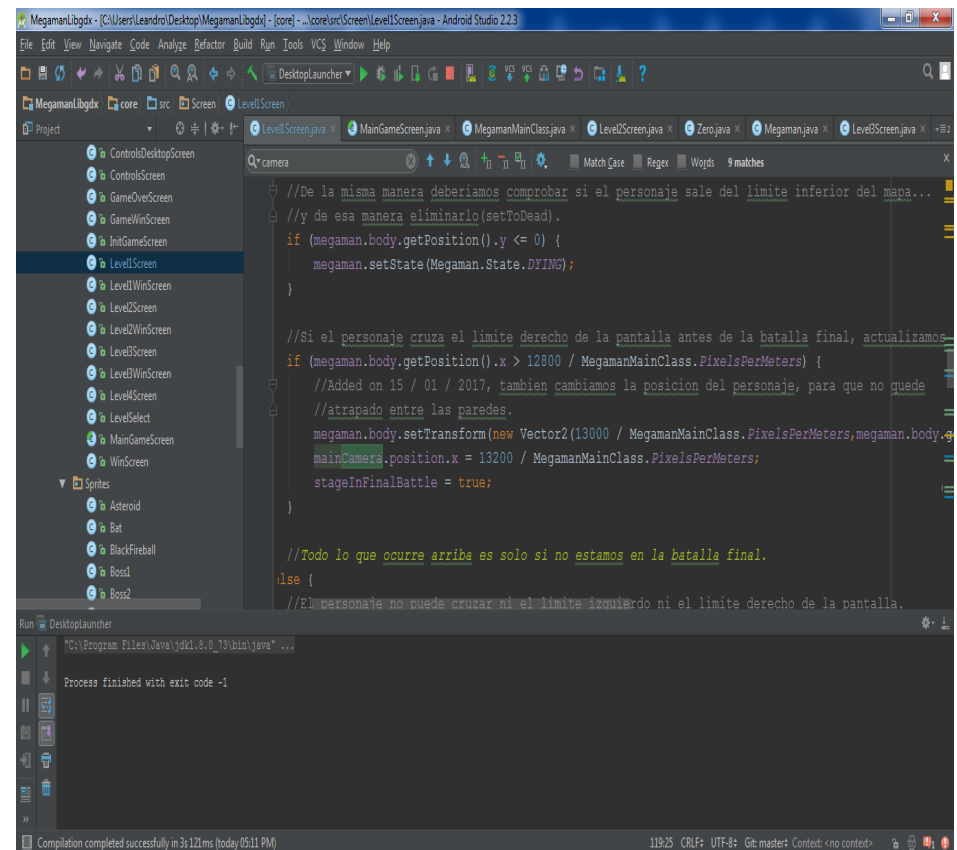
```

Lógica del juego 3/16

Main Character Movement

Si la posición del personaje es < 0 (si cae por debajo del suelo) entonces el personaje muere.

Si el personaje cruza los límites izquierdos o derechos de la pantalla cuando no debe hacerlo, entonces reposicionamos su cuerpo con una transformación (`body.transform`), para que parezca que no puede cruzar esa parte de la pantalla.



```

MeganLibdx - [C:\Users\Leandro\Desktop\MeganLibdx] - [core] - core\src\Screen\LevelScreen.java - Android Studio 223
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help

MeganLibdx core src Screen LevelScreen
Project
  ControlDesktopScreen
  ControlScreen
  GameOverScreen
  GameWinScreen
  InitGameScreen
  LevelScreen
  LevelWinScreen
  Level2Screen
  Level2WinScreen
  Level3Screen
  Level4Screen
  LevelWinScreen
  LevelSelect
  MainGameScreen
  WinScreen
  Sprites
    Asteroid
    Bat
    BlackFireball
    Boss1
    Boss2

camera
//De la misma manera deberiamos comprobar si el personaje sale del limite inferior del mapa...
//y de esa manera eliminarlo (setToDead).
if (megaman.body.getPosition().y <= 0) {
    megaman.setState(Megaman.State.DYING);
}

//Si el personaje cruza el limite derecho de la pantalla antes de la batalla final, actualizamos
if (megaman.body.getPosition().x > 12800 / MegamanMainClass.PixelsPerMeters) {
    //Added on 15 / 01 / 2017, tambien cambiamos la posicion del personaje, para que no quede
    //atrapado entre las paredes.
    megaman.body.setTransform(new Vector2(13000 / MegamanMainClass.PixelsPerMeters, megaman.body.y));
    mainCamera.position.x = 13200 / MegamanMainClass.PixelsPerMeters;
    stageInFinalBattle = true;
}

//Todo lo que ocurre arriba es solo si no estamos en la batalla final.
else {
    //El personaje no puede cruzar ni el limite izquierdo ni el limite derecho de la pantalla.
}

Run DesktopLauncher
"C:\Program Files\Java\jdk-8.0_73\bin\java" ...
Process finished with exit code -1

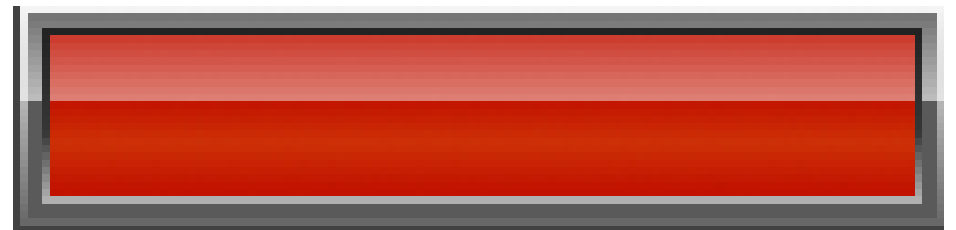
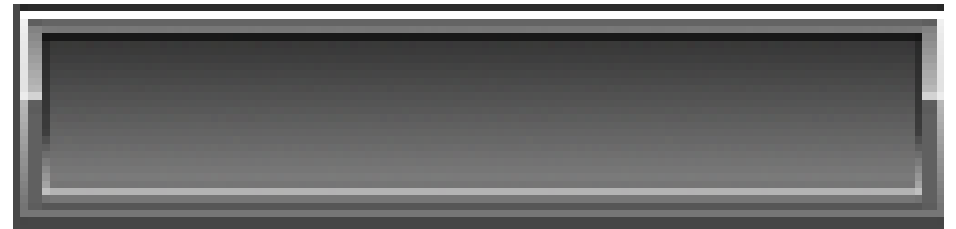
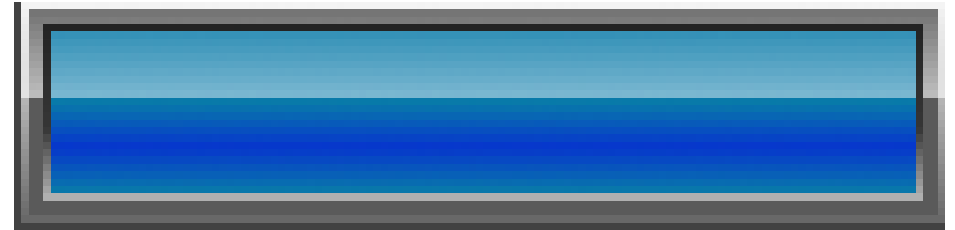
Compilation completed successfully in 3s121ms (today 05:11 PM) 118:25 CRLF UTF-8 Git: master Context: <no context>
```

Lógica del juego 4/16

Life/Mana Bar

Las barras de vida y de mana, están compuestas por dos imágenes cada una: un knob y un background.

El background siempre está fijo, lo que hacemos, es ir recortando la longitud de la imagen del knob para que se vaya visualizando el background(que es la imagen de fondo).

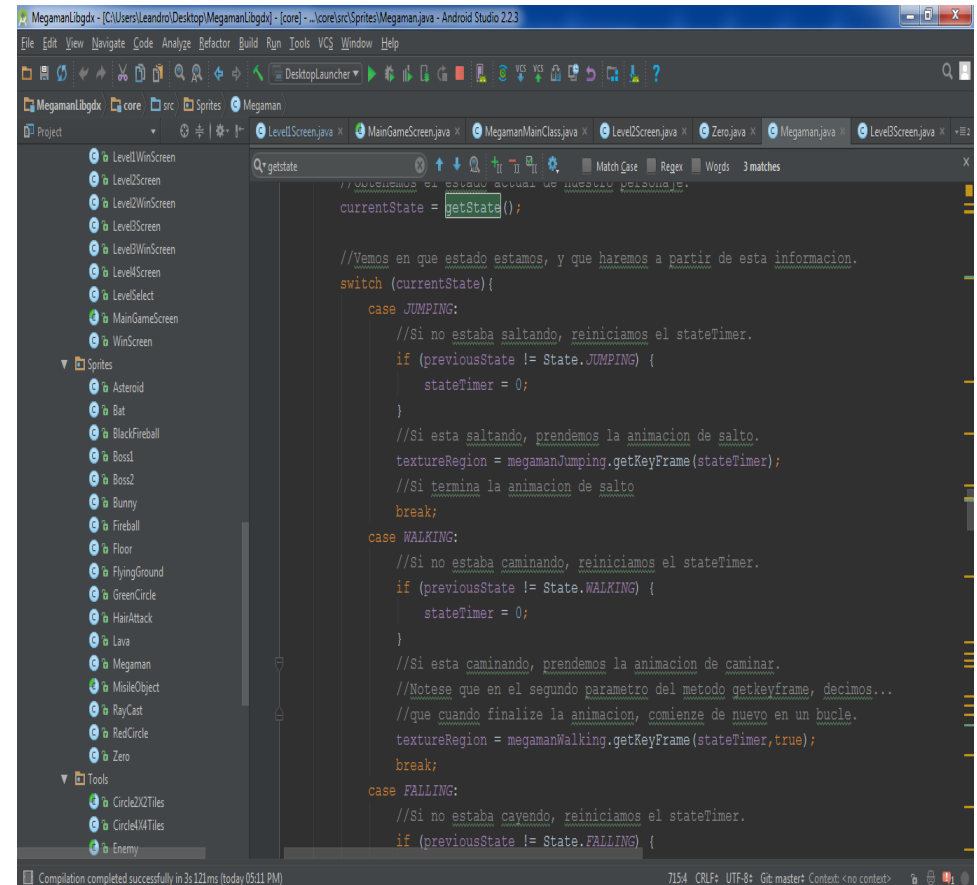


Lógica del juego 5/16

Main Character States(1/2).

El personaje principal puede tener los siguientes estados:

- State.Standing(el personaje esta quieto)
- State.Jumping(el personaje salta)
- State.Dying(el personaje muere)
- State.Hitting(el personaje pega)
- State.Sliding(el personaje se agarra a la pared)
- State.GettingHit(el personaje es golpeado)
- State.Slashing(el personaje se desliza por el suelo)
- State.Crouching(el personaje se agacha)
- State.Falling(el personaje cae).



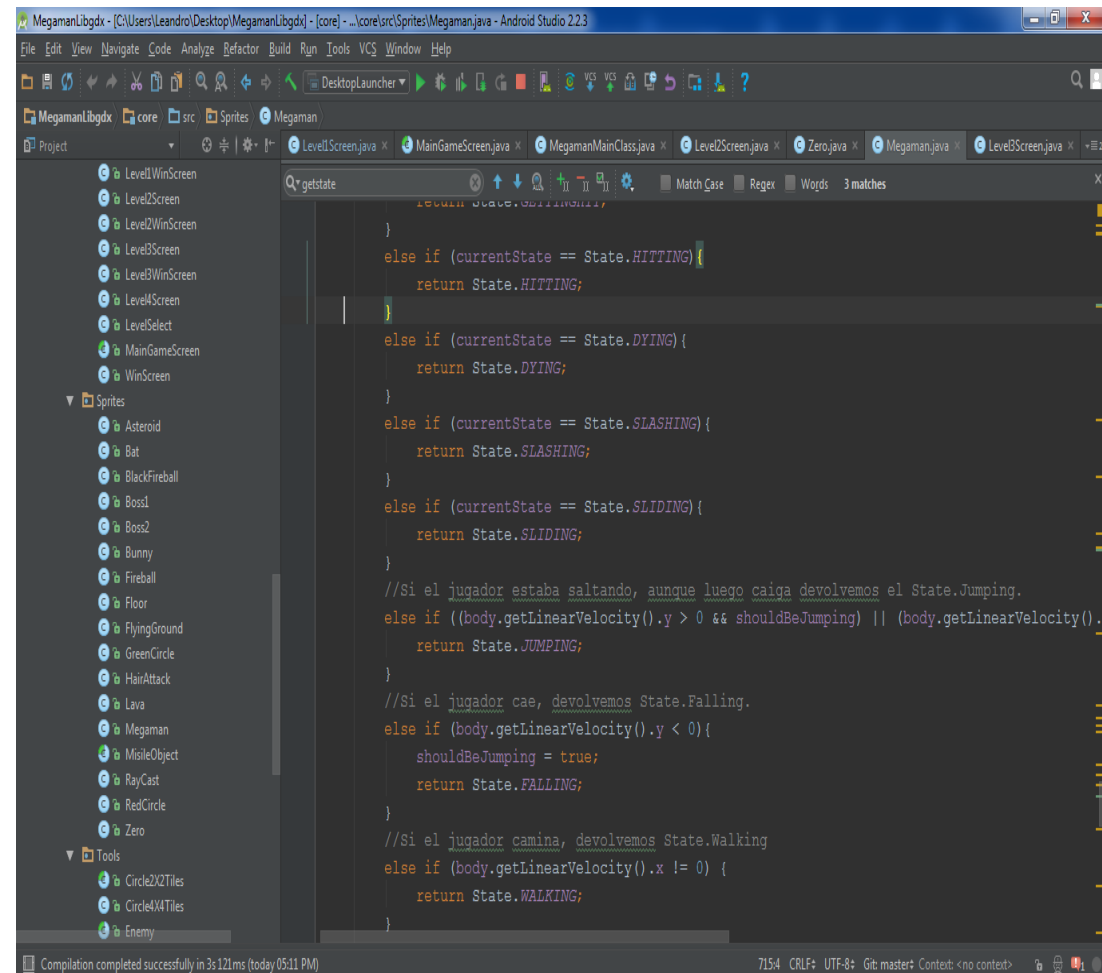
```
//Obtenemos el estado actual de nuestro personaje.
currentState = getState();

//Vemos en que estado estamos, y que haremos a partir de esta informacion.
switch (currentState){
    case JUMPING:
        //Si no estaba saltando, reiniciamos el stateTimer.
        if (previousState != State.JUMPING) {
            stateTimer = 0;
        }
        //Si esta saltando, prendemos la animacion de salto.
        textureRegion = megamanJumping.getKeyFrame(stateTimer);
        //Si termina la animacion de salto
        break;
    case WALKING:
        //Si no estaba caminando, reiniciamos el stateTimer.
        if (previousState != State.WALKING) {
            stateTimer = 0;
        }
        //Si esta caminando, prendemos la animacion de caminar.
        //Notese que en el segundo parametro del metodo getKeyframe, decimos...
        //que cuando finalice la animacion, comienze de nuevo en un bucle.
        textureRegion = megamanWalking.getKeyFrame(stateTimer,true);
        break;
    case FALLING:
        //Si no estaba cayendo, reiniciamos el stateTimer.
        if (previousState != State.FALLING) {
```

Lógica del juego 6/16

Main Character States(2/2).

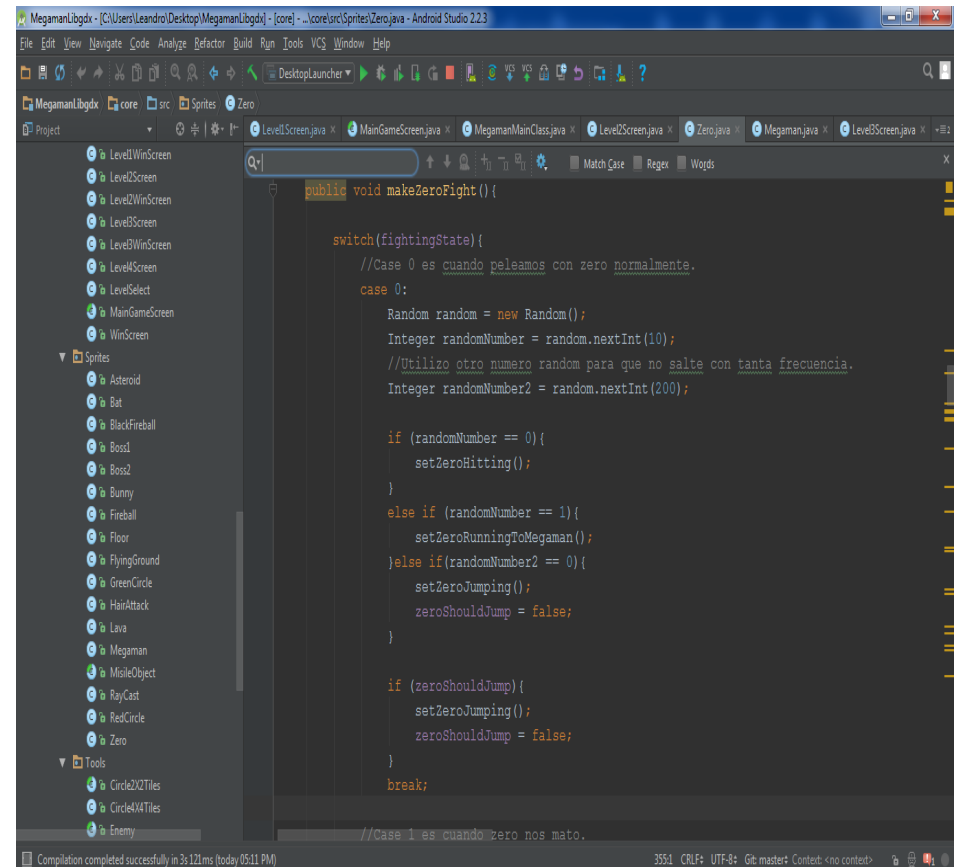
Cada uno de estos estados posee una imagen o animación asociada, que se activa cuando el usuario realiza ciertas acciones o si el juego indica que el personaje debe estar en cierto estado.



Lógica del juego 7/16

Boss1 Character IA.

- Se han utilizado numeros aleatorios para que no sea posible predecir el movimiento del enemigo.
- Mediante la selección de numeros al azar el jefe final obtiene ciertas acciones que debe realizar en la pelea.
- Si el numero random es 1, entonces el jefe dispara, si es 2, el jefe salta, si es 3, el jefe se acerca.
- También hago uso de un sensor rectangular gigante para detectar disparo del personaje principal.



```
public void makeZeroFight(){
    switch(fightingState){
        //Case 0 es cuando peleamos con zero normalmente.
        case 0:
            Random random = new Random();
            Integer randomNumber = random.nextInt(10);
            //Utilizo otro numero random para que no salte con tanta frecuencia.
            Integer randomNumber2 = random.nextInt(200);

            if (randomNumber == 0){
                setZeroHitting();
            }
            else if (randomNumber == 1){
                setZeroRunningToMegaman();
            }
            else if (randomNumber2 == 0){
                setZeroJumping();
                zeroShouldJump = false;
            }

            if (zeroShouldJump){
                setZeroJumping();
                zeroShouldJump = false;
            }
            break;

        //Case 1 es cuando zero nos mato.
```

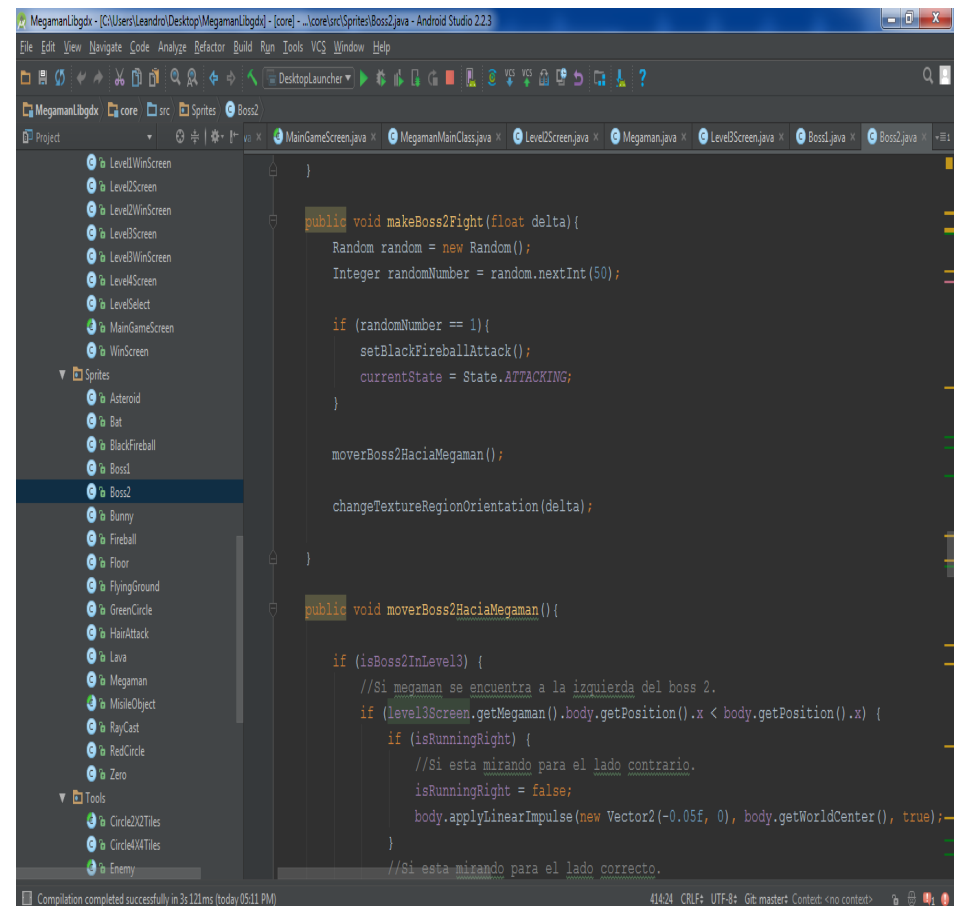
Lógica del juego 8/16

Boss2 Character IA.

Al igual que el jefe 1, el jefe 2 pasa por varios estados en el momento de la pelea.

- Si el random es 1, ataca con el ataque 1.
- Si el random es 2, ataca con el ataque 2.
- Si el random es 3, se teletransporta a una posición cercana.

Es importante remarcar, que hay varios numeros random calculados, es decir, para el algoritmo de ataque1, se busca un numero random 1:10, para el de ataque 2 se busca un numero random 1:50, y asi con otras acciones.



```
public void makeBoss2Fight(float delta){
    Random random = new Random();
    Integer randomNumber = random.nextInt(50);

    if (randomNumber == 1){
        setBlackFireballAttack();
        currentState = State.ATTACKING;
    }

    moverBoss2HaciaMegaman();

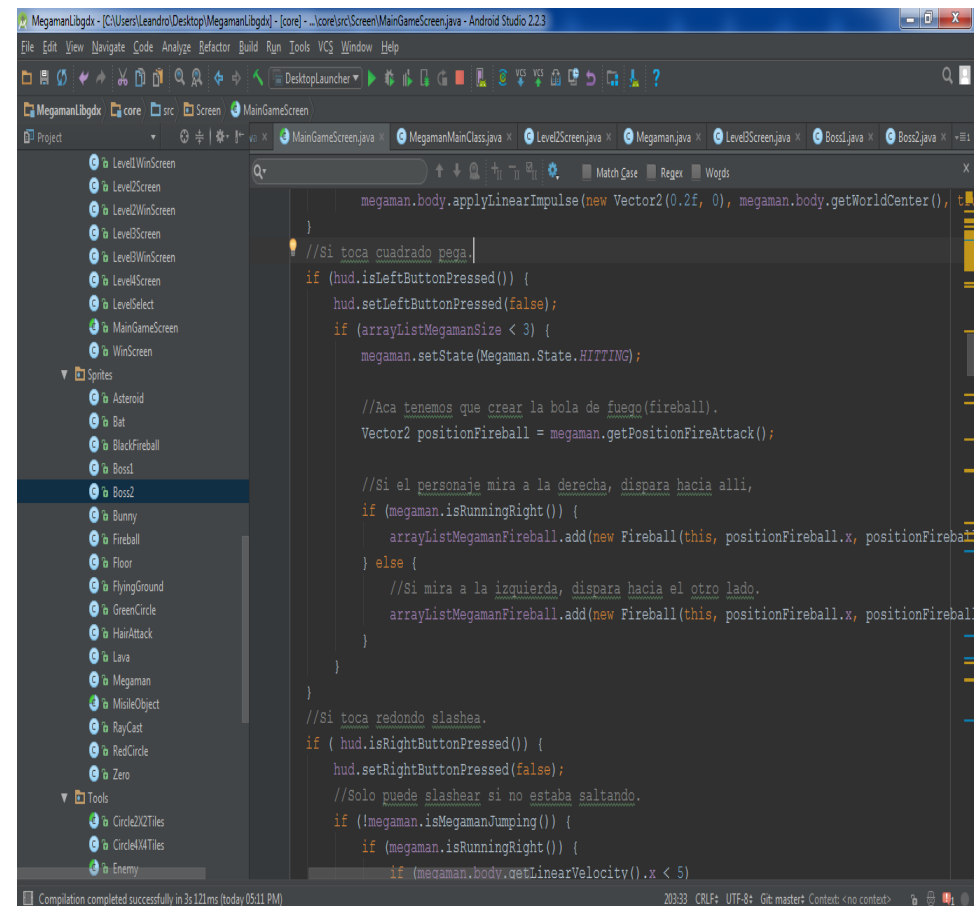
    changeTextureRegionOrientation(delta);
}

public void moverBoss2HaciaMegaman(){
    if (isBoss2InLevel3) {
        //Si megaman se encuentra a la izquierda del boss 2.
        if (level3Screen.getMegaman().body.getPosition().x < body.getPosition().x) {
            if (isRunningRight) {
                //Si esta mirando para el lado contrario.
                isRunningRight = false;
                body.applyLinearImpulse(new Vector2(-0.05f, 0), body.getWorldCenter(), true);
            }
            //Si esta mirando para el lado correcto.
        }
    }
}
```

Lógica del juego 9/16

FireBalls.

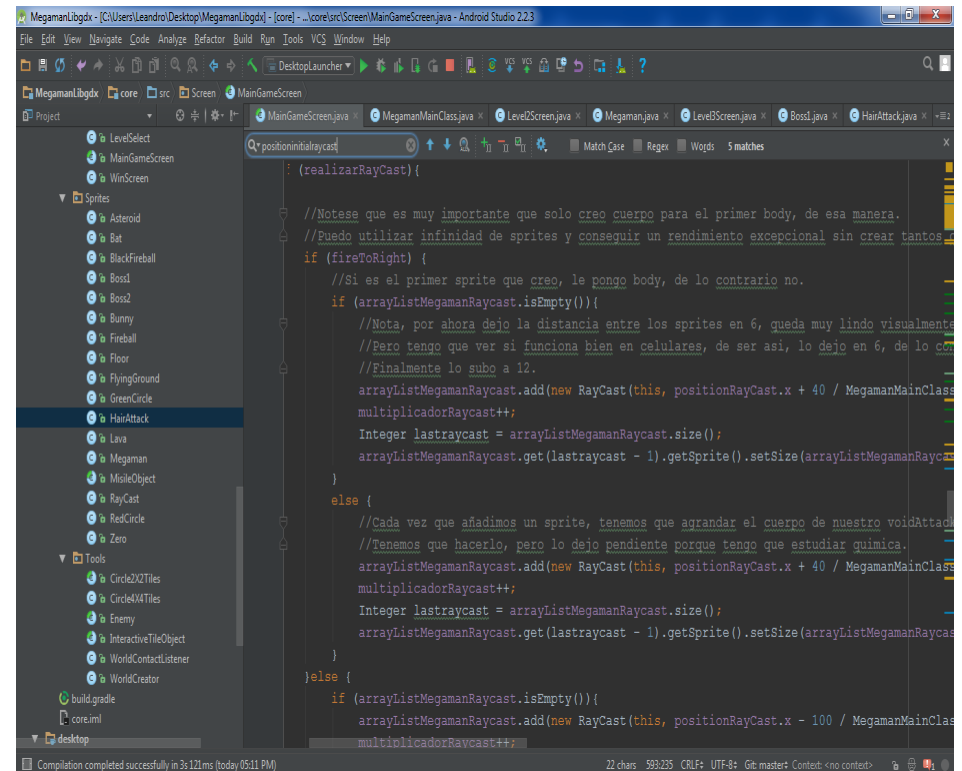
- Se crea un ArrayList, y cuando el usuario toca cierto botón, se añade un fireball al arraylist, se crea el cuerpo del fireball con un impulso en la orientación adecuada y se asigna el sprite de fireball al cuerpo.
- Cuando cada fireball sale de la pantalla, entonces lo eliminamos del ArrayList, para que no quede en memoria.
- Este funcionamiento esta expandido a todos los objetos misiles creados, por lo que tenemos la clase MissileObject y todos los objetos que se lanzan heredan propiedades de esta.



Lógica del juego 10/16

Gravity Power.

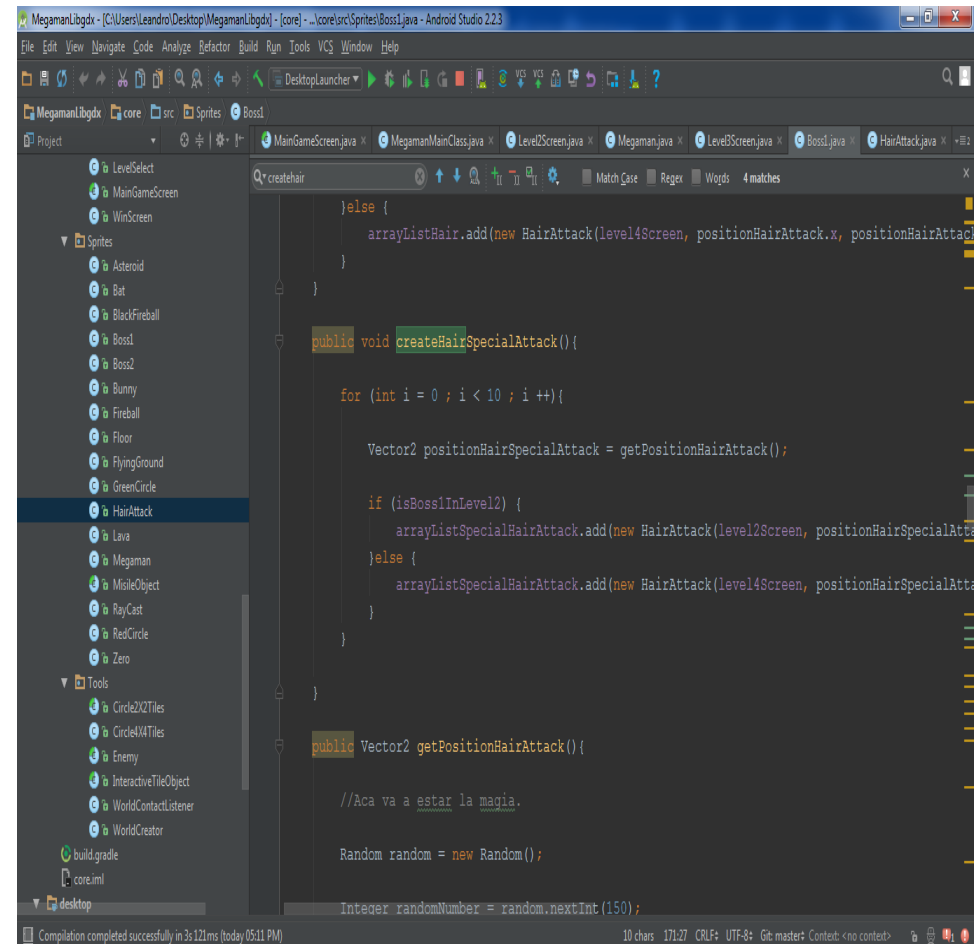
- El primer sprite lo dibujamos normalmente en la pantalla, y luego por cada fracción de tiempo(1/60fps), se va dibujando un sprite al lado del otro, hasta que se llega a cierta distancia entre el último sprite y la posición del sprite inicial.
- Cada vez que se dibuja un nuevo sprite, el tamaño del sprite inicial aumenta ligeramente, y la distancia entre los sprites también es modificada.
- De esta manera, se pueden crear efectos impresionantes, y muy variables modificando la forma en la que van apareciendo y desapareciendo los sprites.



Lógica del juego 11/16

Hair Attack.

- Lo que cambia es la dirección del impulso, es decir, no se trata de un lanzamiento horizontal, sino que se calcula la trayectoria del objeto.
- Está pensado como si fueran vectores, la fórmula es la siguiente: $\text{vectorTrayectoriaMisión} = \text{vectorPosiciónVictimaMisión} - \text{vectorPosiciónMisión}$.
- El impulso de los asteroides utiliza el mismo algoritmo.



```
public void createHairSpecialAttack() {
    for (int i = 0; i < 10; i++) {
        Vector2 positionHairSpecialAttack = getPositionHairAttack();

        if (isBoss1InLevel2) {
            arrayListSpecialHairAttack.add(new HairAttack(level2Screen, positionHairSpecialAttack));
        } else {
            arrayListSpecialHairAttack.add(new HairAttack(level4Screen, positionHairSpecialAttack));
        }
    }
}

public Vector2 getPositionHairAttack() {
    //Aca va a estar la magia.

    Random random = new Random();

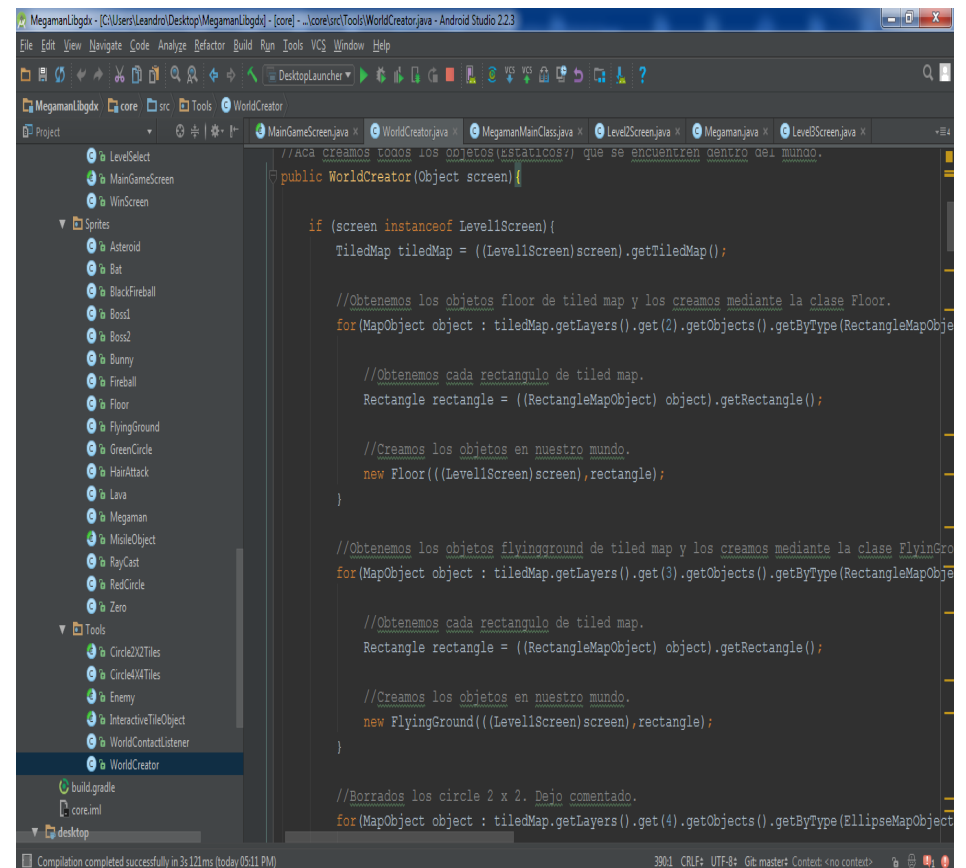
    Integer randomNumber = random.nextInt(150);
}
```

Lógica del juego 12/16

Creación de objetos en el mapa.

Cuando desde Tiled creamos objetos, solo son posiciones, luego tenemos que desde android studio, crear los cuerpos respectivos y asignarles los sprites correspondientes.

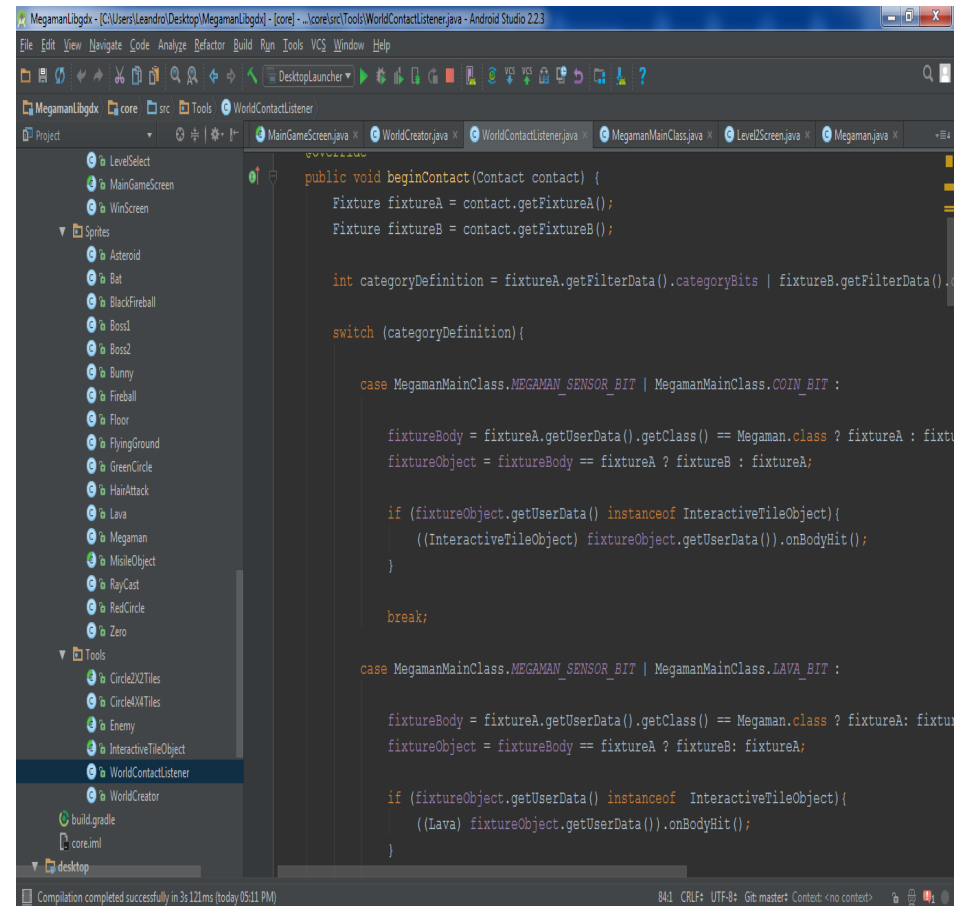
- Para eso tenemos una clase llamada WorldCreator, que dependiendo del nivel en el que nos encontramos, crea los objetos que puede tener cada nivel.
- WorldCreator se encarga de crear todos los enemigos intermedios, y luego cederle el ArrayList al respectivo nivel.



Lógica del juego 13/16

WorldContactListener.

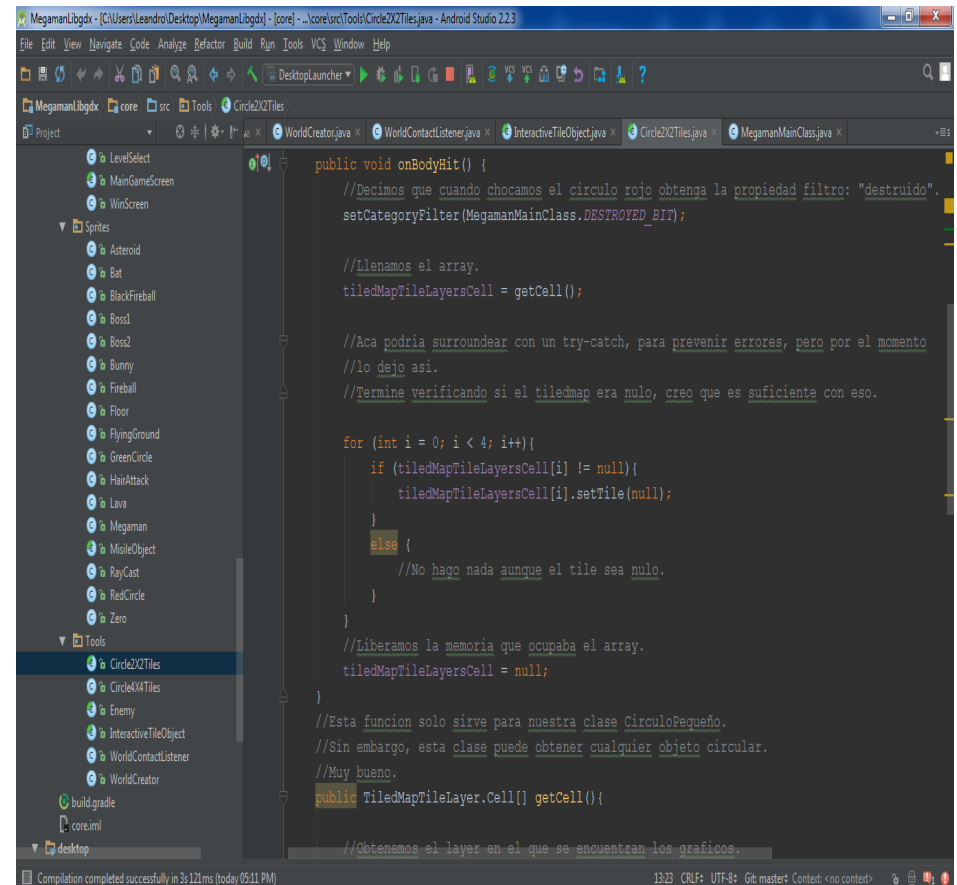
- Si bien ya sabemos como detectar y elegir las colisiones, falta decidir que hacer cuando ocurren. La clase `WorldContactListener` se encarga de esto, activa todos los eventos correspondientes a cada colisión.
- Cuando ocurre una colisión, en esta clase se distingue qué es lo que chocó con qué, y se puede realizar cualquier cambio que se desee.



Lógica del juego 14/16

Graphics Erasing

- Eliminación de ciertos gráficos de Tiled o desactivación cuando se hace contacto con ellos no es cosa simple.
- Por ejemplo, cuando agarramos un círculo de vida, y lo queremos eliminar, tenemos que crear una función que elimine los tiles que se encuentran alrededor del respectivo círculo.

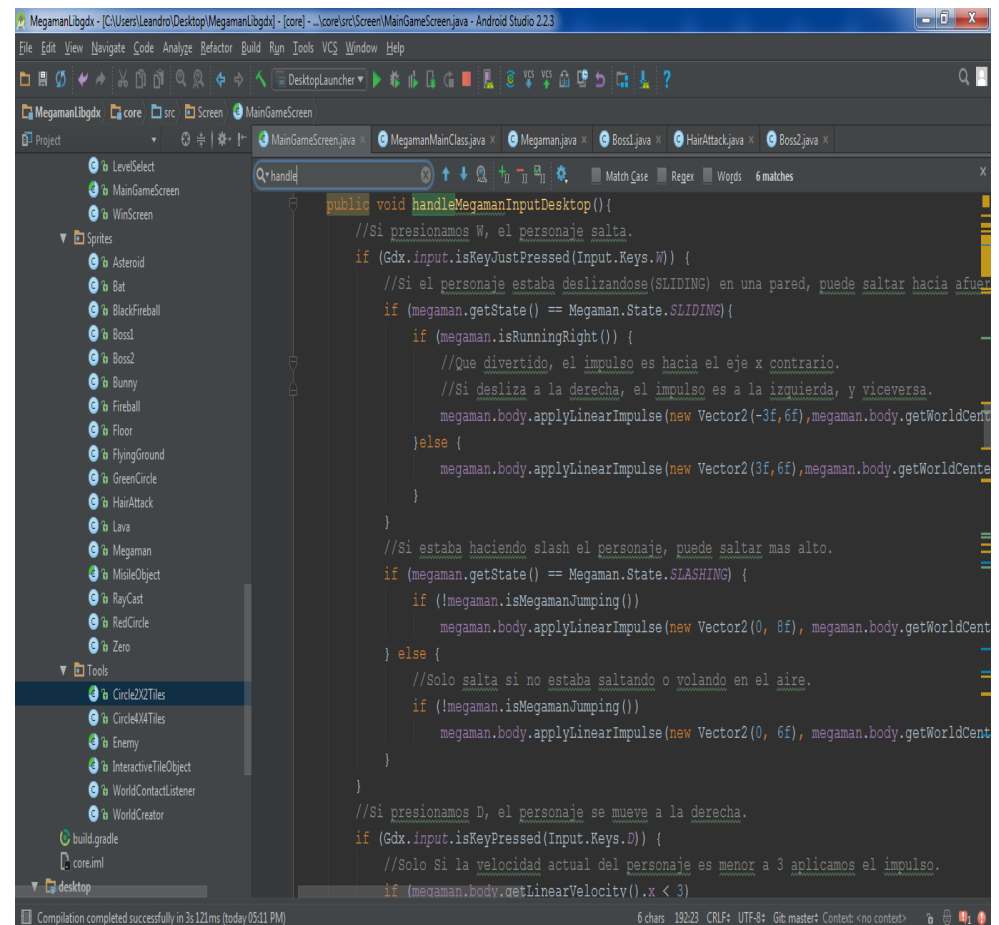


```
public void onBodyHit() {  
    //Decimos que cuando chocamos el círculo rojo obtenga la propiedad filtro: "destruido".  
    setCategoryFilter(MegamanMainClass.DESTROYED_BIT);  
  
    //Llenamos el array.  
    tiledMapTileLayersCell = getCell();  
  
    //Aca podría surroundear con un try-catch, para prevenir errores, pero por el momento  
    //lo dejo así.  
    //Termino verificando si el tiledMap era nulo, creo que es suficiente con eso.  
  
    for (int i = 0; i < 4; i++){  
        if (tiledMapTileLayersCell[i] != null){  
            tiledMapTileLayersCell[i].setTile(null);  
        }  
        else {  
            //No hago nada aunque el tile sea nulo.  
        }  
    }  
    //Liberamos la memoria que ocupaba el array.  
    tiledMapTileLayersCell = null;  
}  
  
//Esta función solo sirve para nuestra clase CírculoPequeño.  
//Sin embargo, esta clase puede obtener cualquier objeto circular.  
//Muy bueno.  
public TiledMapTileLayer.Cell[] getCell(){  
    //Obtenemos el layer en el que se encuentran los gráficos.
```

Lógica del juego 15/16

Handling Input

- Libgdx nos da el manejo del input del usuario, sin embargo, nosotros debemos utilizar los datos de la posición que nos brinda y utilizarlo para realizar cambios en el juego.
- Usualmente, como el personaje principal es lo que es modificable, el ingreso de un comando por parte del usuario lo que hace es cambiar el estado del personaje, ya sea aplicando un impulso o aplicando un evento.
- También se ha creado el Hud para android, el cual reacciona al input del usuario.

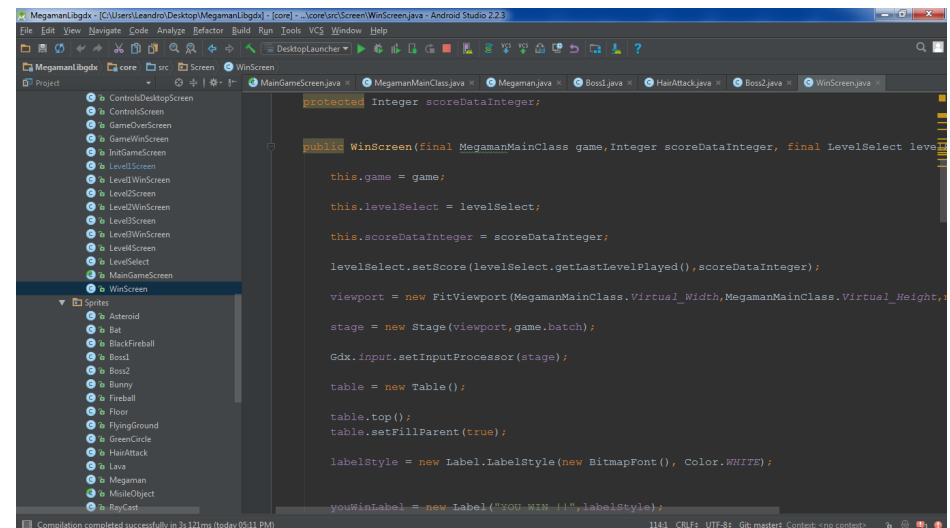


```
public void handleMegamanInputDesktop() {
    //Si presionamos W, el personaje salta.
    if (Gdx.input.isKeyJustPressed(Input.Keys.W)) {
        //Si el personaje estaba deslizando (SLIDING) en una pared, puede saltar hacia afuera.
        if (megaman.getState() == Megaman.State.SLIDING) {
            if (megaman.isRunningRight()) {
                //Que divertido, el impulso es hacia el eje x contrario.
                //Si desliza a la derecha, el impulso es a la izquierda, y viceversa.
                megaman.body.applyLinearImpulse(new Vector2(-3f, 6f), megaman.body.getWorldCenter());
            } else {
                megaman.body.applyLinearImpulse(new Vector2(3f, 6f), megaman.body.getWorldCenter());
            }
        }
        //Si estaba haciendo slash el personaje, puede saltar mas alto.
        if (megaman.getState() == Megaman.State.SLASHING) {
            if (!megaman.isMegamanJumping()) {
                megaman.body.applyLinearImpulse(new Vector2(0, 8f), megaman.body.getWorldCenter());
            } else {
                //Solo salta si no estaba saltando o volando en el aire.
                if (!megaman.isMegamanJumping()) {
                    megaman.body.applyLinearImpulse(new Vector2(0, 6f), megaman.body.getWorldCenter());
                }
            }
        }
        //Si presionamos D, el personaje se mueve a la derecha.
        if (Gdx.input.isKeyPressed(Input.Keys.D)) {
            //Solo si la velocidad actual del personaje es menor a 3 aplicamos el impulso.
            if (megaman.body.getLinearVelocity().x < 3) {
                megaman.body.applyLinearImpulse(new Vector2(1, 0), megaman.body.getWorldCenter());
            }
        }
    }
}
```

Lógica del juego 16/16

Screens

- Se han utilizado múltiples Screens en el juego, para la pantalla de inicio, victoria, game over, controles, selección, etc.
- Estas pantallas utilizan Stage, mediante el cual poseen actores y estos actúan.
- El actor principal usualmente es un table, en el cual se agregan todos los labels y objetos necesarios.
- Este Stage, usualmente tiene un InputProcessor, para detectar cuando un usuario pulsa cierto label.



Interfaz del juego 1/3

LevelSelect

LevelSelect será la clase encargada de administrar las múltiples pantallas del juego.

- Aquí es también donde se guarda toda la información volátil correspondiente al juego.
- También esta clase maneja el guardado de datos y estados que deben permanecer estables aunque el juego se cierre.
- Esta clase siempre permanece en memoria cuando la aplicación está corriendo, ya que solo contiene lo necesario y no significa una gran pérdida de rendimiento.

Interfaz del juego 2/3

Score

- Cada vez que eliminamos a un enemigo, el puntaje obtenido aumenta.
- Tenemos dos tipos de puntaje, el que vemos todo el tiempo en pantalla, cuando jugamos los niveles, y el de la tabla de puntajes, que se encuentra en la pantalla de inicio.

Interfaz del juego 3/3

Preferences

Un preference es una herramienta similar a utilizar una base de datos, o archivos txts, para guardar información aunque se cierre la aplicación.

- Utiliza un sistema de mapeo, en el cual, le debemos mandar un string asociado a un valor.
- Los archivos creados se guardan en la carpeta **C://USERS/USER/.prefs.**

Finalizando 1/2

Reutilización del código.

El hecho de decidir utilizar un framework en vez de un engine me permitió obtener las siguientes ventajas:

- Cada minúscula parte del juego es modificable.
- Entiendo como funciona el juego, qué características hacen perder rendimiento y cuáles no.
- Puedo reutilizar el código, ahorrando mucho tiempo, ya que tengo una base de la cual partir.
- Terminé creando un simil Game Engine.
- El límite es la imaginación.

Finalizando 2/2

Finalizando.

- Cada una de las secciones de lógica del juego, debe implementarse, y aquí es donde ocurren los problemas. En la implementación, pueden ocurrir errores, o simplemente no encontrar forma de hacer funcionar una idea.
- Siempre hay una solución para poder realizar lo que uno quiere, solo se trata de tener paciencia y buscar una y otra vez información.
- Una de las herramientas muy útiles en el desarrollo fue Github, ya que pude tener un respaldo del código fuente.