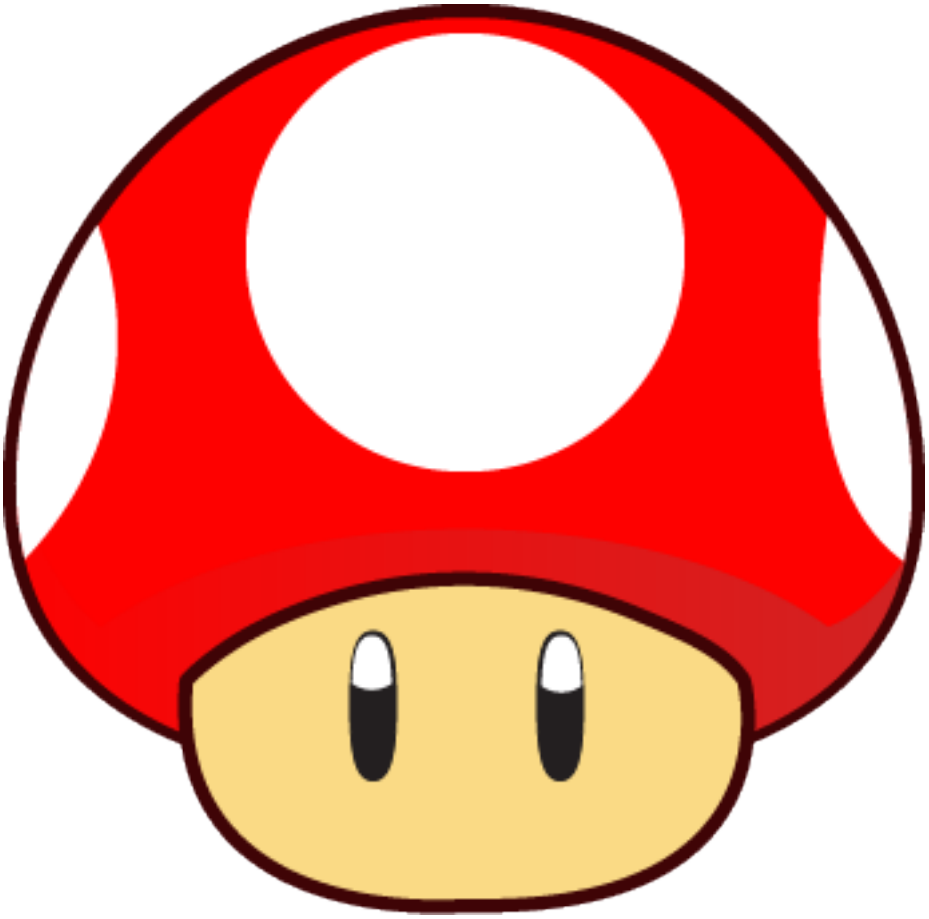
EL LABERINTO

PEPE LÓPEZ SALADO



26 / 05 / 2022

ÍNDICE

INTRODUCCIÓN

El Laberinto es un juego cuyo objetivo es llegar a la meta sin ser derrotado por los diferentes elementos que construyen el mapa.
También tendrás que recoger una serie de monedas y objetos como llaves para poder continuar en el mapa. Está compuesto por tres niveles.

En el modo historia, tendrás que pasar de nivel en nivel hasta llegar al último, el cuál, una vez superado te llevará a la pantalla final de créditos.

También se da la opción de jugar al nivel que quieras sin tener que superar los niveles anteriores.

Si eres eliminado en alguno de los niveles, te llevará a una pantalla donde podrás elegir si continuar o salir al menú principal. En caso de salir al menú principal, se te permitirá continuar el nivel donde abandonaste pulsando en la primera opción, continuar.

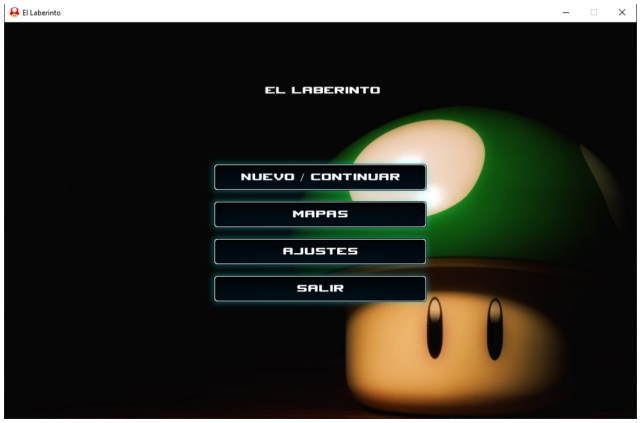
En cualquier momento del juego podrás pulsar “Esc” para salir del nivel.

En todo momento podrás visualizar el número de monedas que llevas en el nivel y las que te quedan para conseguirlas todas. No son necesarias para pasar de nivel.

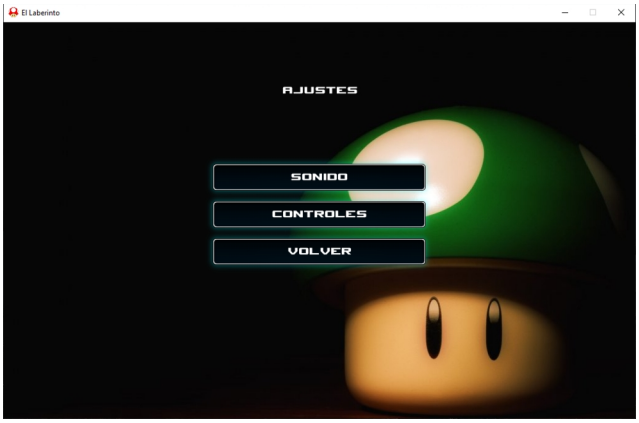
En el menú podrás encontrar las diferentes opciones, entre ellas, el apartado de ajustes, en el cuál podrás apagar o encender el volumen de la música y del sonido de los botones, y por otra parte, encontrarás un apartado con los controles del juego.

PANTALLAS

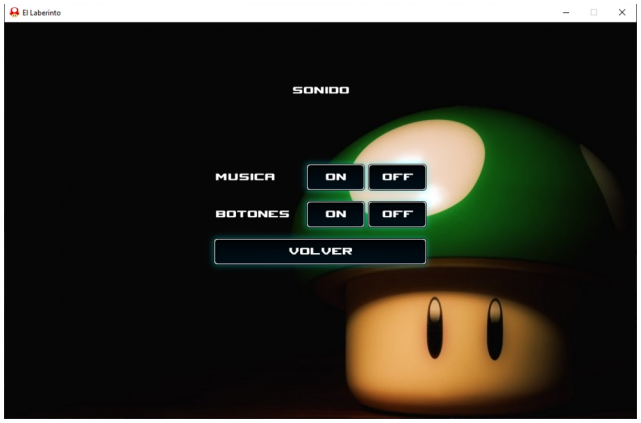
Menú Principal



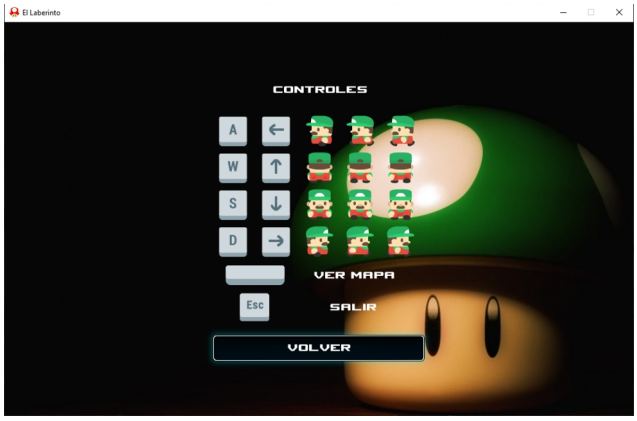
Ajustes



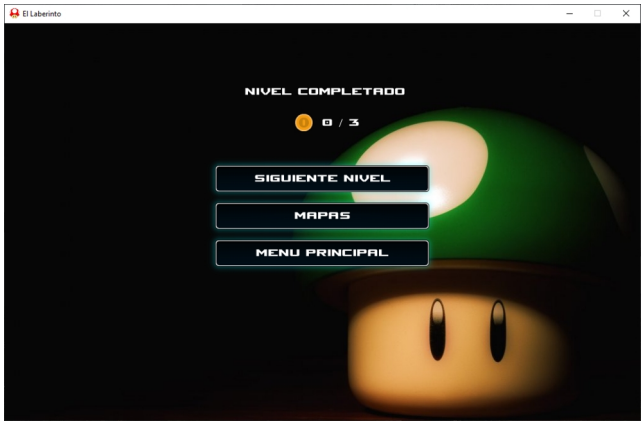
Sonidos



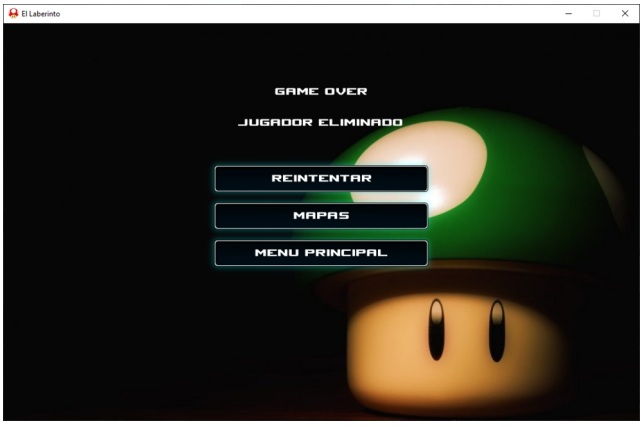
Controles



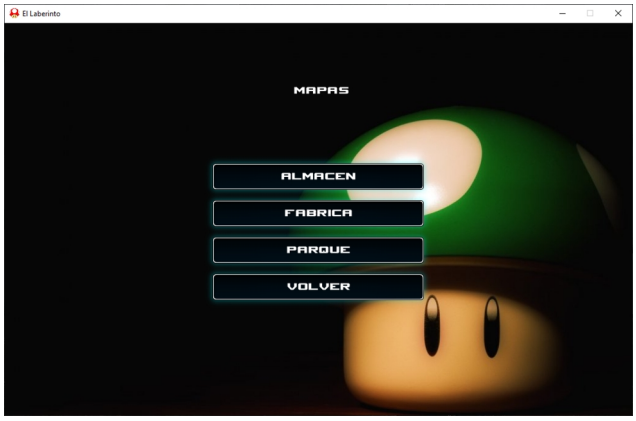
Victoria



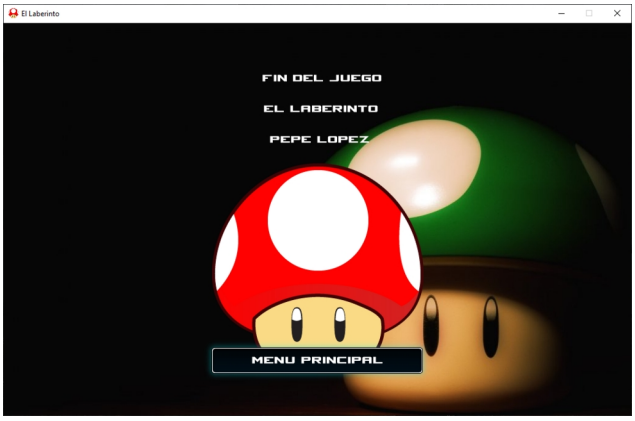
Eliminado



Mapas

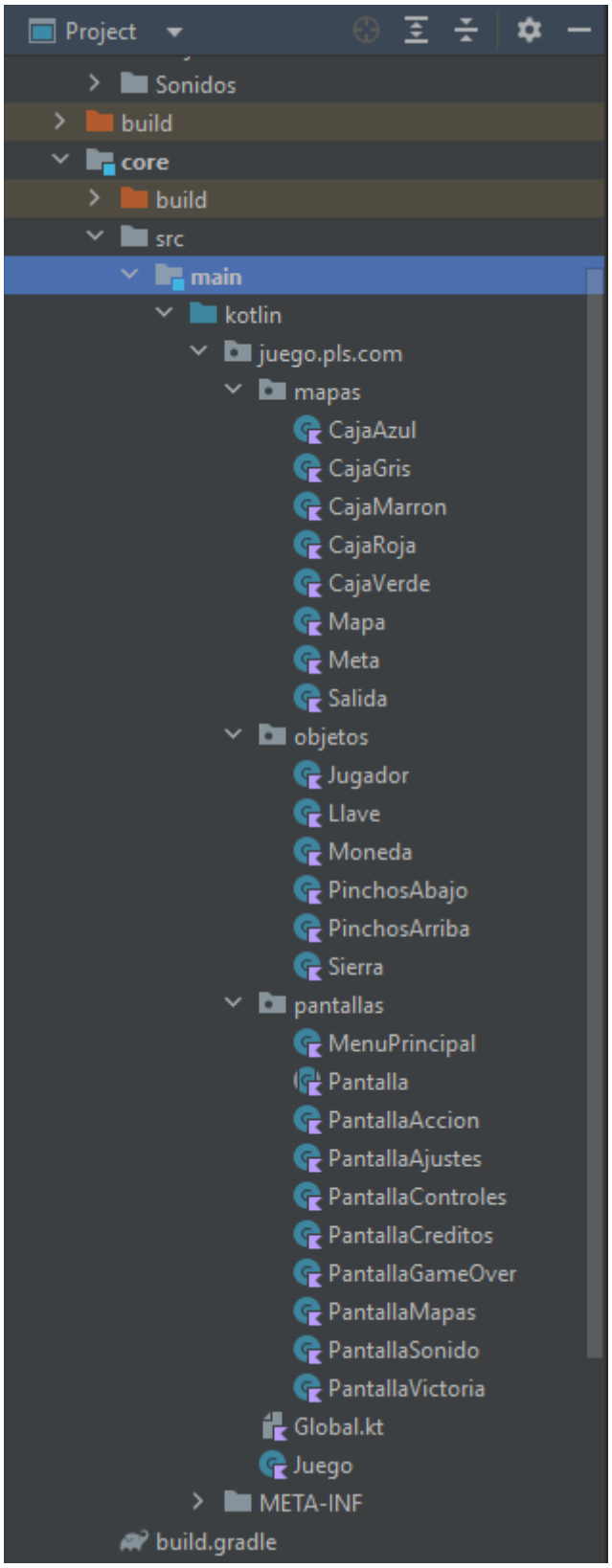


Créditos



Pantalla de Acción

ESQUEMA



Clases

El código del juego está formado por 25 clases, dividido en 3 secciones.

9 de ellas son las diferentes pantallas que conforman el menú y la pantalla de acción. Todas ellas extienden de Pantalla.
En la parte final del documento se explica que funcionalidad se aplica a todas estas pantallas.

Por otra parte, el juego consta de otras 13 clases que junto al jugador, son los diferentes objetos que aparecen en pantalla.

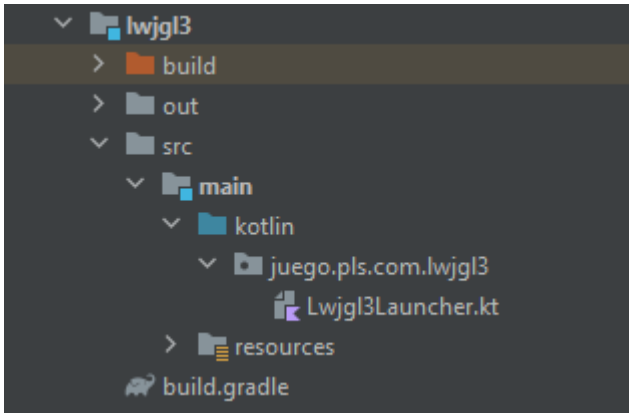
Más abajo se explica la funcionalidad y uso de cada uno de ellos.

También podemos diferenciar la clase “Mapa”, la cual obtiene un mapa de enteros, y a partir de ese mapa se encarga de crear los objetos en su lugar.

En “Global” encontramos parámetros que serán usados en muchas de las otras clases.

Por último tenemos la clase “Juego”, la cual se encarga de gran parte del funcionamiento de dicho juego, que ya veremos mas adelante.

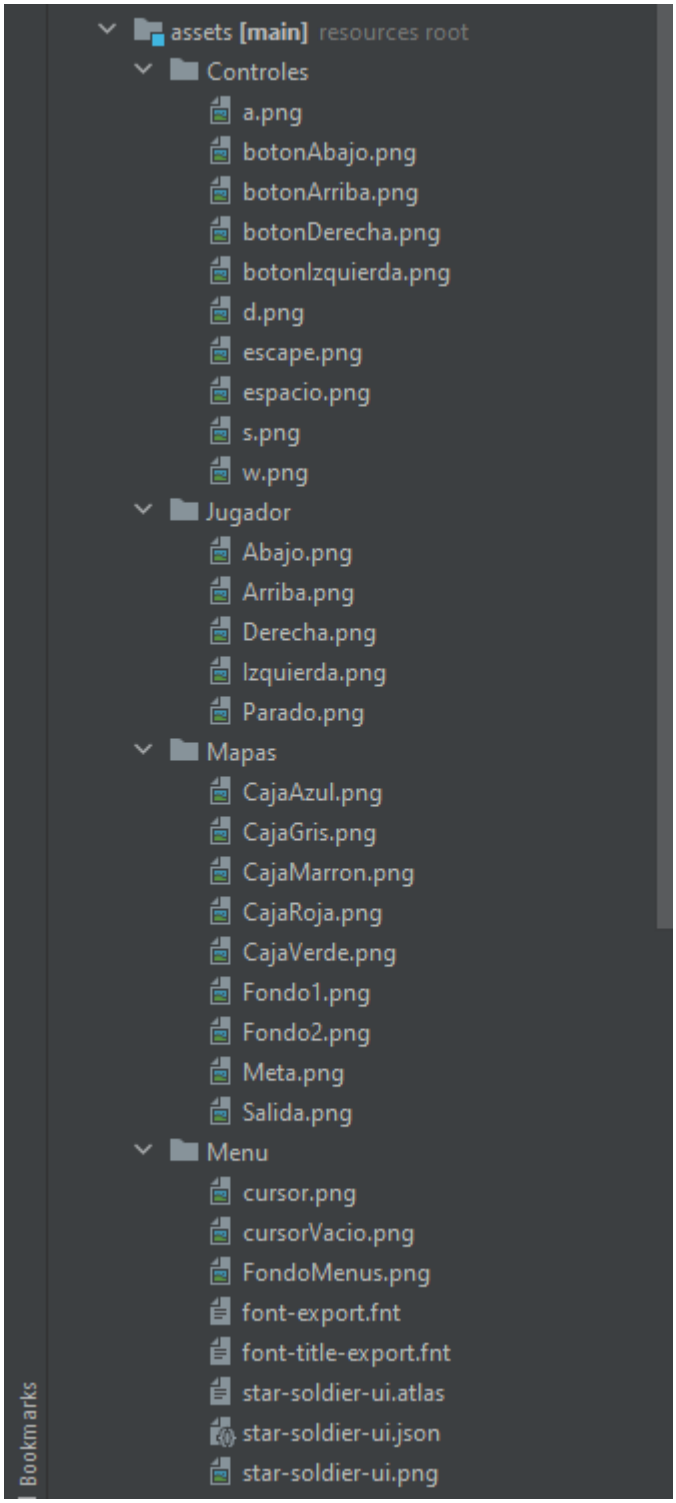
Y para que el juego sea lanzado, se hará uso del documento “Lwjgl3Launcher.kt”

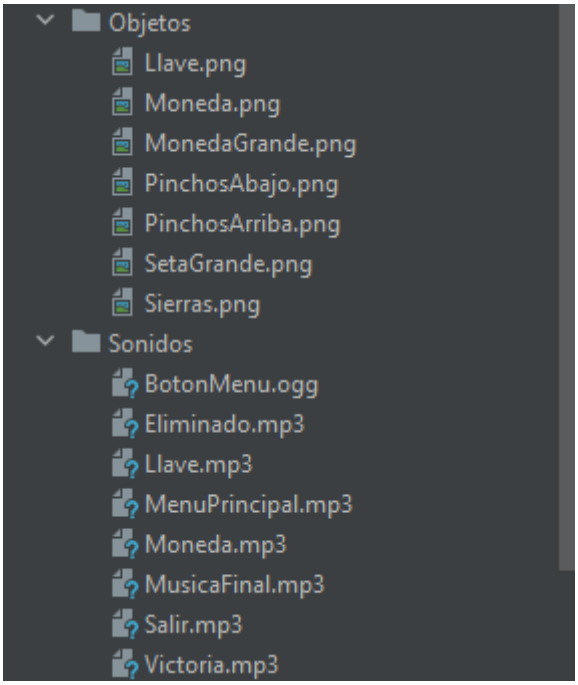


Assets

En la carpeta assets encontramos todos los archivos necesarios para nuestro juego, entre ellos, las diferentes imágenes que conforman los objetos, además de sus animaciones y fondos.

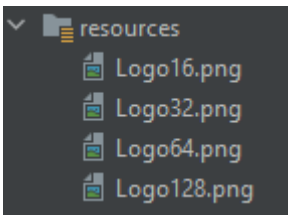
También incluye los sonidos y la música de fondo, además de la fuente utilizada para el menú y los archivos necesarios para su uso.





Resources

Para el logo del juego será necesario incluir las imágenes en los diferentes tamaños para que se adapte a lo que necesite, y todo esto se encuentra en el directorio “resources”.



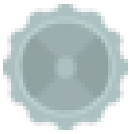
TUTORIAL / GUÍA DE USO

Jugador

Moneda / Llave (Abrir pinchos)

Pinchos

Meta / Salida



Sierra

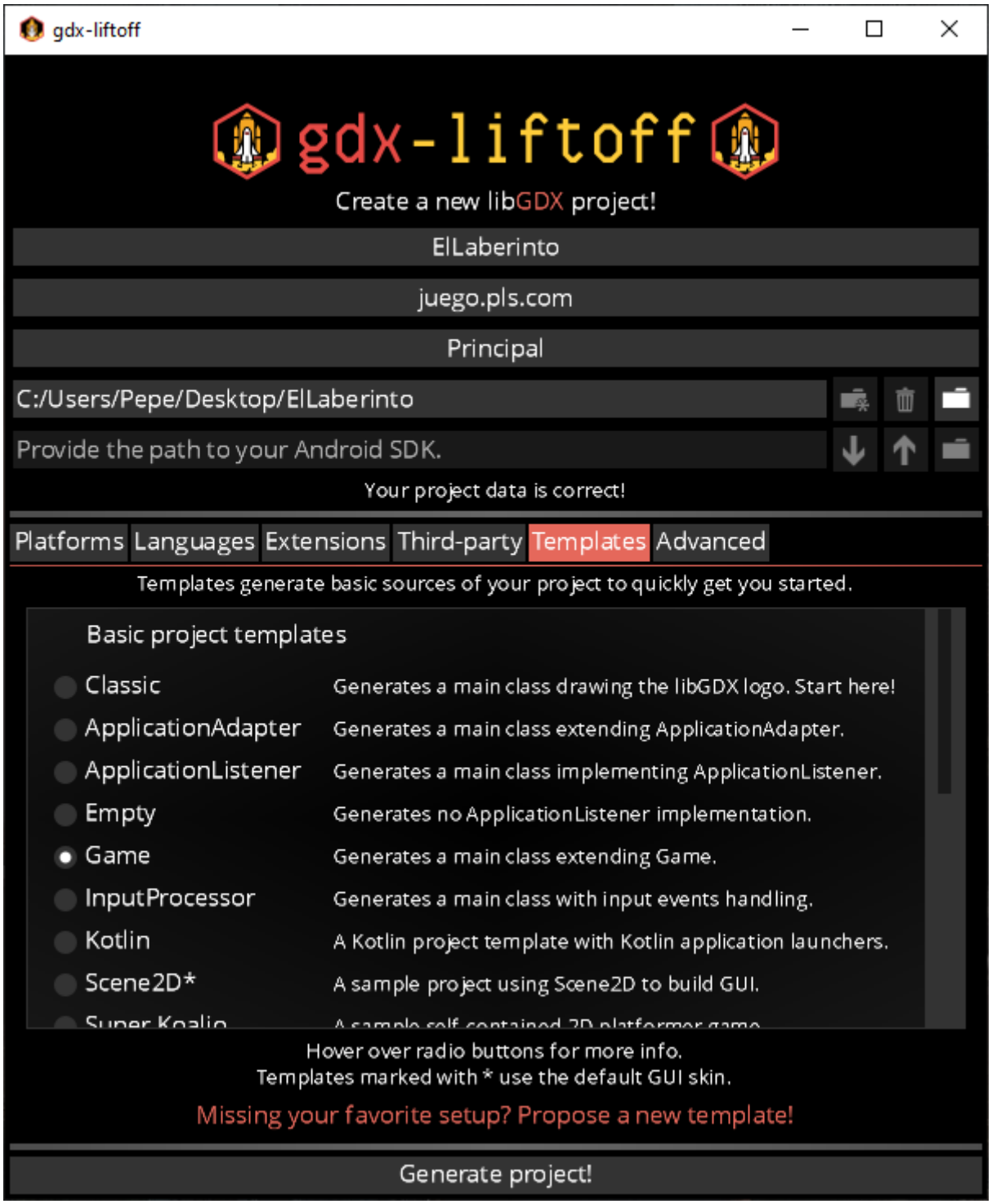
Ver mapa completo

Salir a menú principal

PROGRAMAS

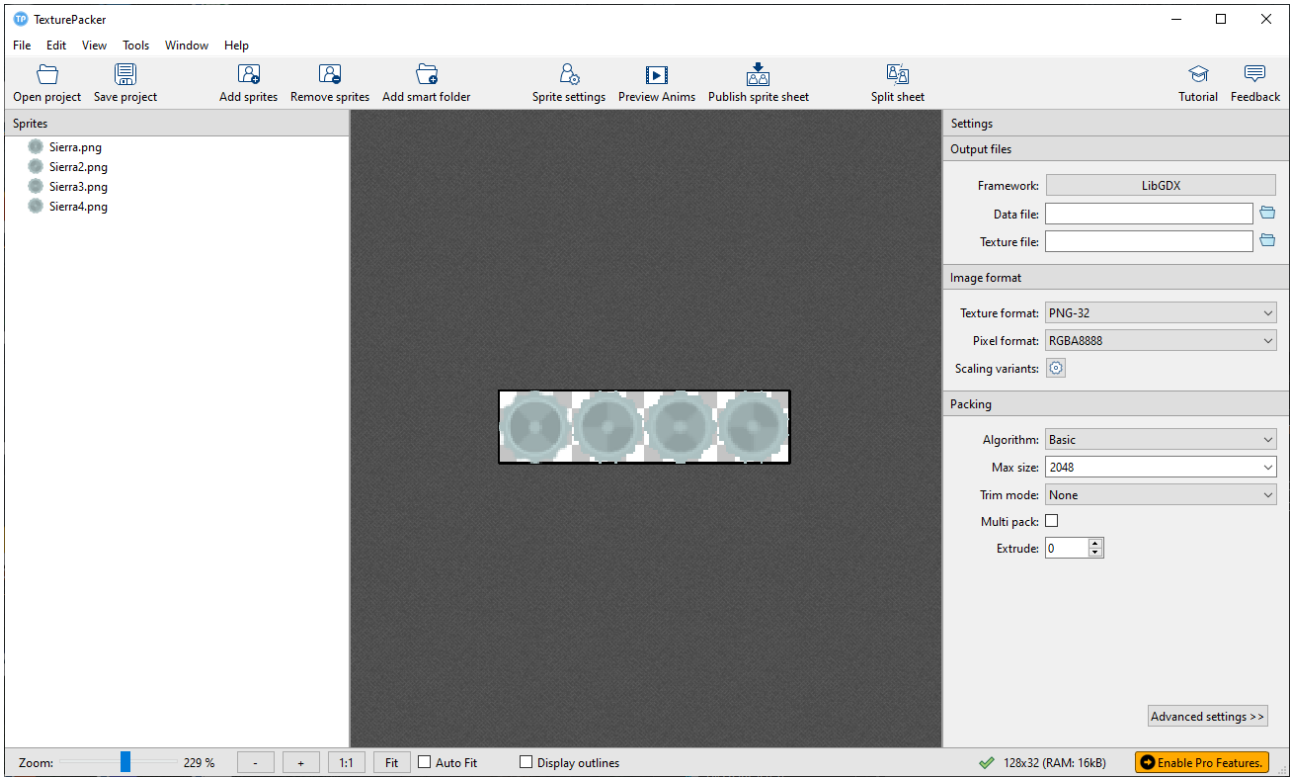
GDX-litoff

Usamos esta herramienta para generar el código inicial de nuestro juego, para ello tendremos que seleccionar en “Templates” la opción de “Game”.



TexturePackerGUI

Uno de los programas que he usado ha sido TexturePacker para crear las texturas que necesitaba para hacer las diferentes animaciones, como es el caso de la sierra, o del movimiento del personaje.



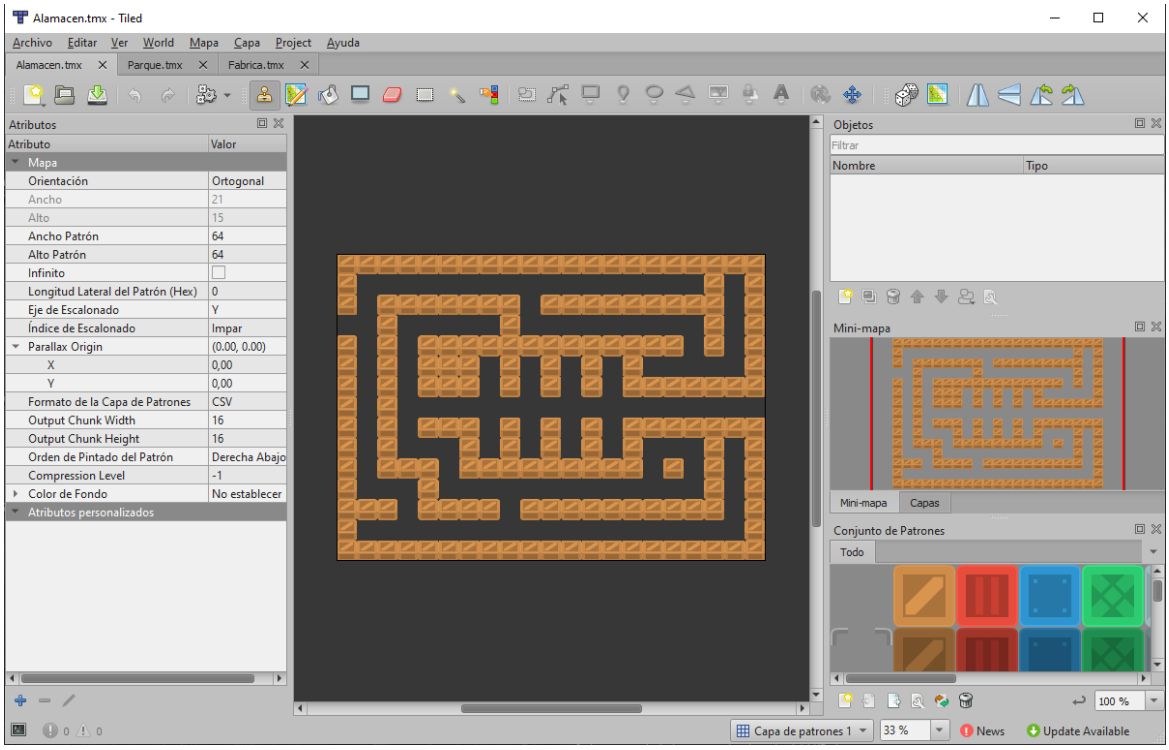
Para usar el programa, tendremos que añadir las imágenes por separado, y seleccionar el algoritmo básico.

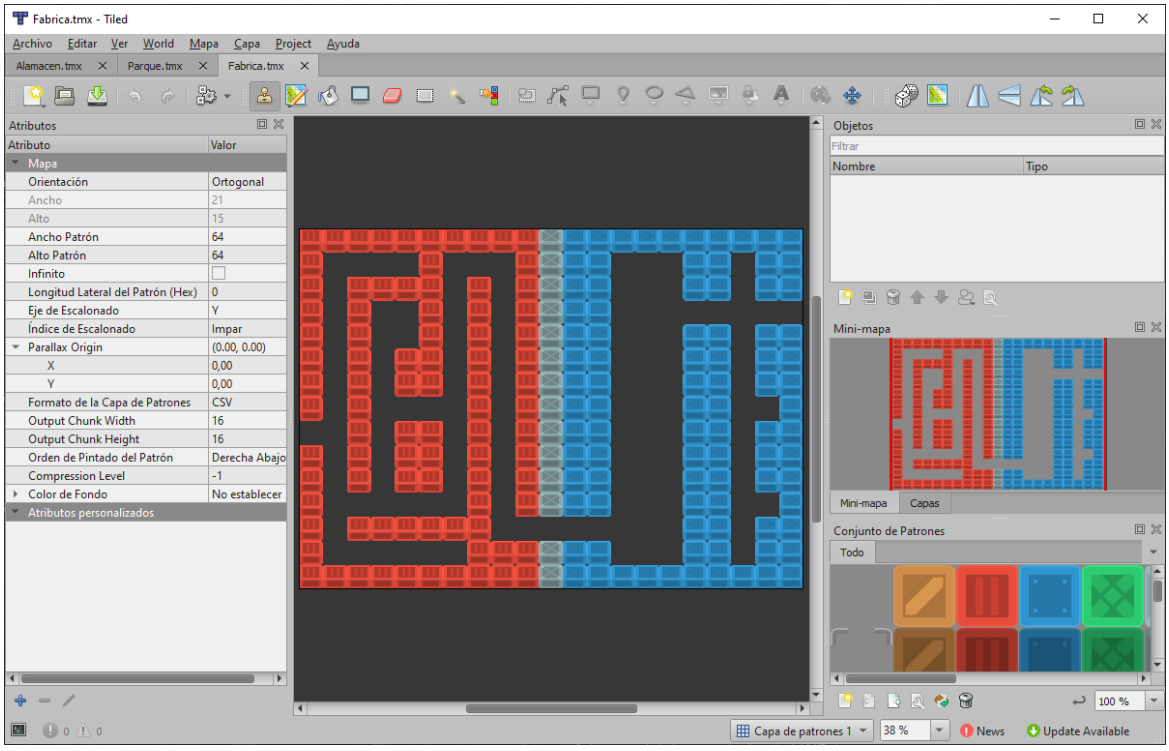
Ésto nos generará una imagen .png ademas de un archivo con todas las posiciones y tamaños de las diferentes imágenes.

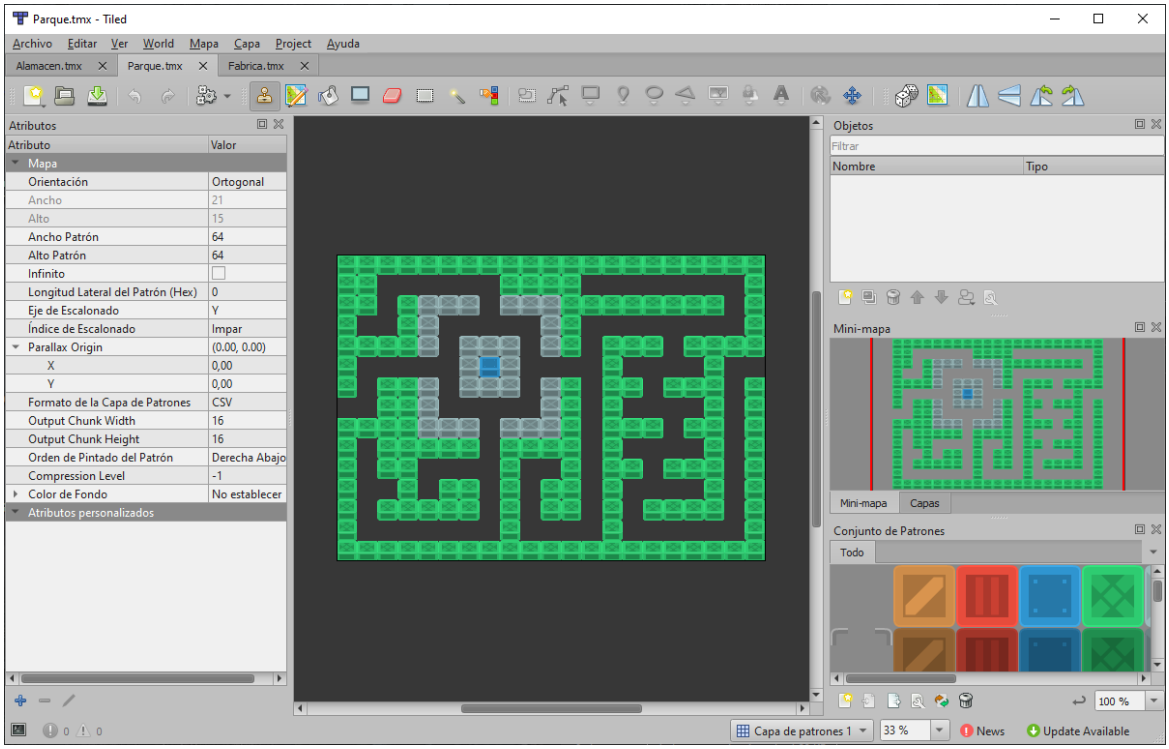
Mas adelante veremos como se implementa esto en el código del juego.

Tiled

Tiled es la herramienta que he usado para generar mis mapas, en mi caso no he trabajado en el código con los propios archivos que nos genera Tiled, pero he usado el mapeado que nos ofrece para crear mis mapas.







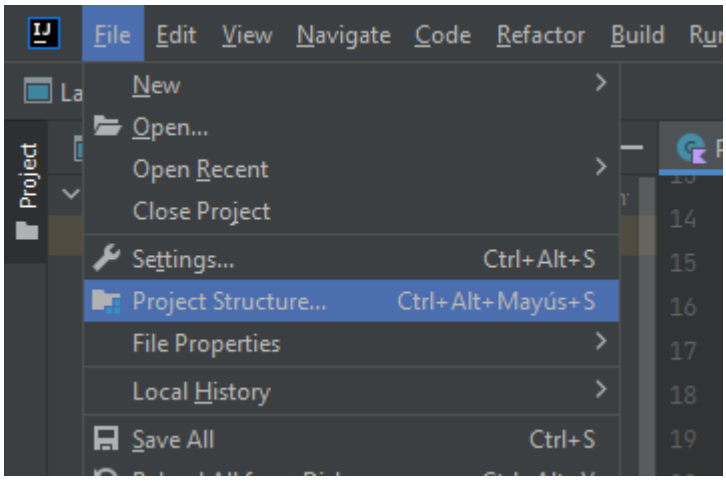
Para crear las colisiones tendremos que acceder a las propiedades de los objetos y añadir el atributo bloqueado, de esta forma, a través del código podemos indicar que elementos harán colisión.

Como indiqué anteriormente, en mi caso uso el mapeado que nos ofrece el archivo .tmx para crear el array que forma cada uno de mis mapas, donde cada número es un objeto diferente.

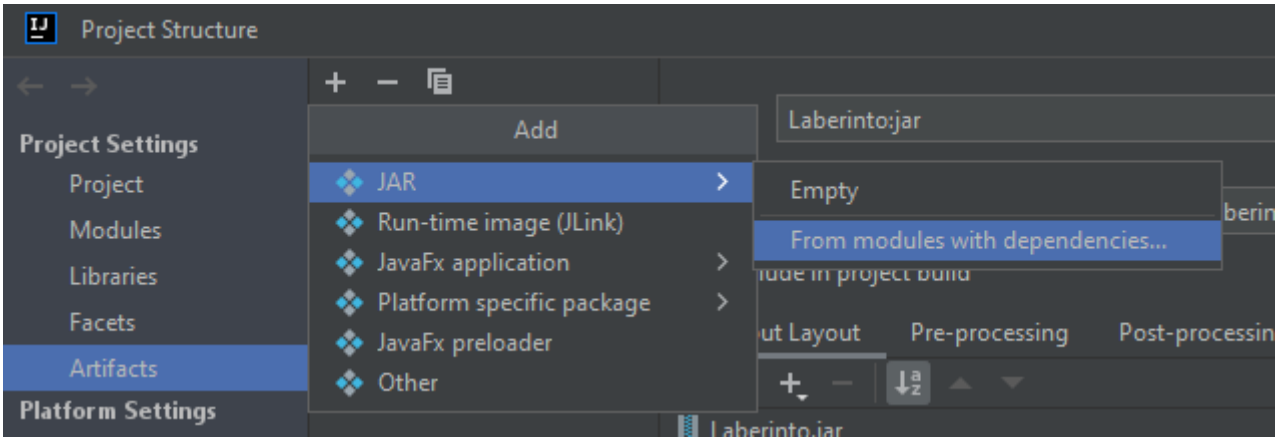
```
val mapa2 = intArrayOf(
    8,8,8,8,8,8,8,8,8,8,11,9,9,9,9,9,9,9,9,9,9,
    8,0,0,0,0,8,0,0,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,8,8,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,0,0,8,0,8,0,8,11,9,9,0,0,0,0,0,0,0,0,
    8,0,8,0,0,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,0,8,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,0,8,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,0,0,0,0,8,0,8,11,9,9,0,0,0,9,9,0,0,9,
    0,0,8,0,8,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,0,8,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,8,0,8,8,0,8,0,8,11,9,9,0,0,0,9,9,0,9,9,
    8,0,0,0,0,0,0,8,0,8,11,9,9,0,0,0,9,9,0,0,9,
    8,0,8,8,8,8,8,8,0,0,0,0,0,0,0,9,9,0,9,9,
    8,0,0,0,0,0,0,8,8,8,11,9,9,0,0,0,9,9,0,9,9,
    8,8,8,8,8,8,8,8,8,8,11,9,9,9,9,9,9,9,9,9,9)
```

Jar

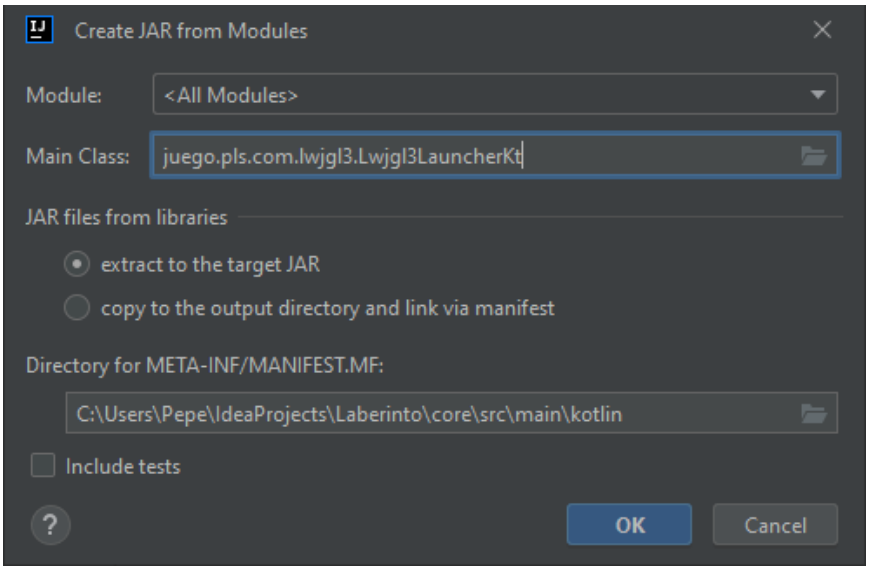
Para crear un archivo .jar de nuestro juego y que podamos ejecutarlo en nuestro equipo sin necesidad de IntelliJ IDEA, primero tendremos que acceder a “Project Structure…”



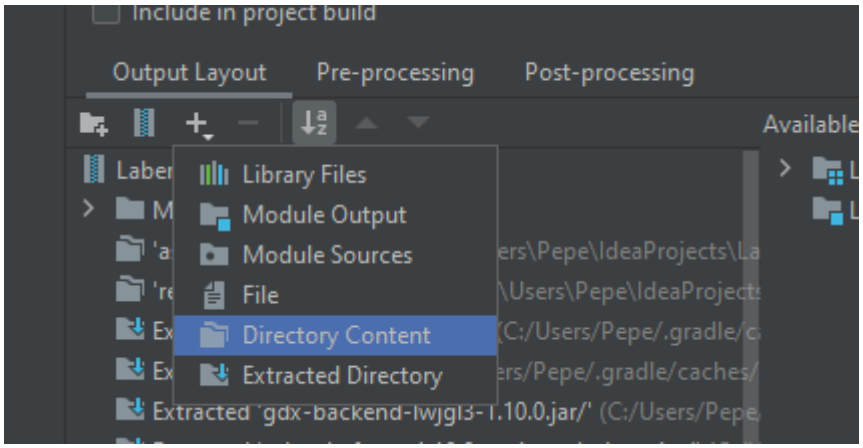
Luego accederemos a “Arifacts, JAR, From modules with dependencies…”



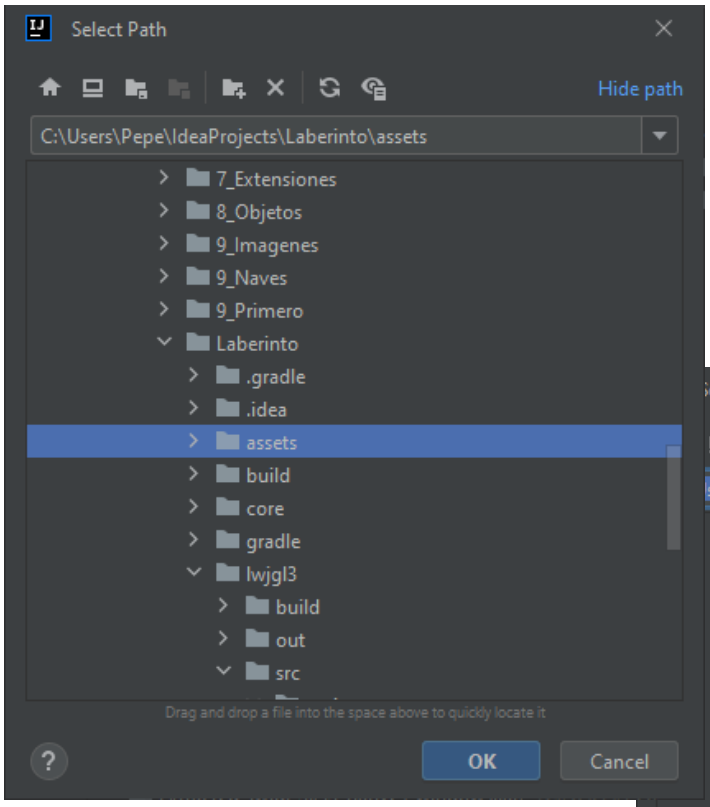
Añadimos la clase principal y pulsamos en “OK”.



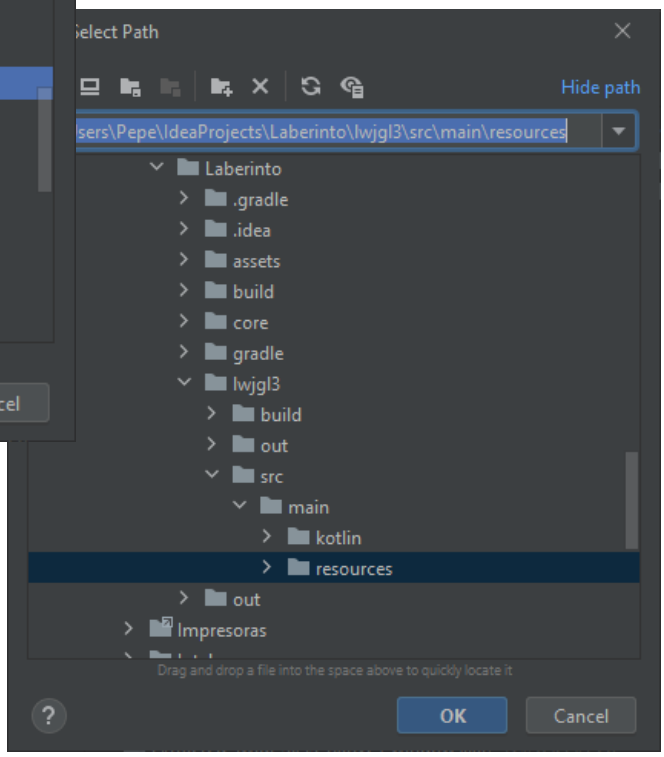
Importante: Como de esta forma no se añaden los assets, es importante que pulsemos en “+” y luego “Directory Content” para añadirlos.



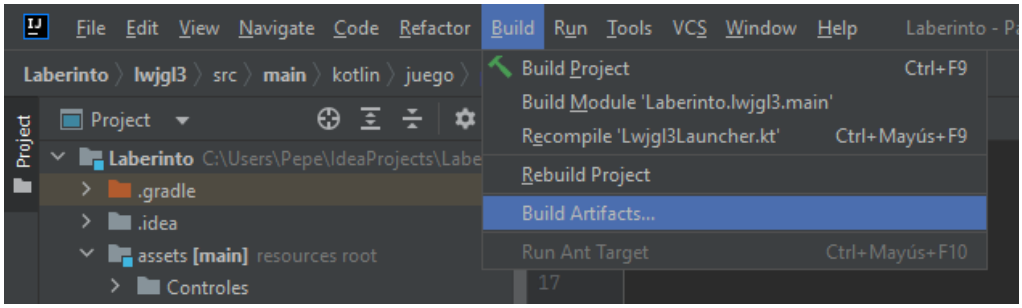
Seleccionamos todo lo que queramos que se añada a nuestro .jar y pulsamos en “OK”.



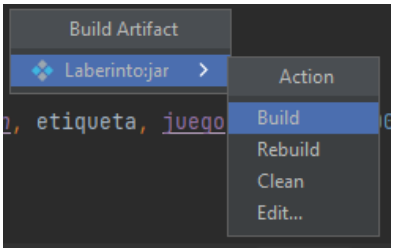
En mi caso también añadiré la carpeta “resources” para que tenga acceso a la imagen inicial de juego en sus diferentes tamaños.



Por último tendremos que pulsar en “Build”, “Build Artifacts…” y seleccionamos “Build”.



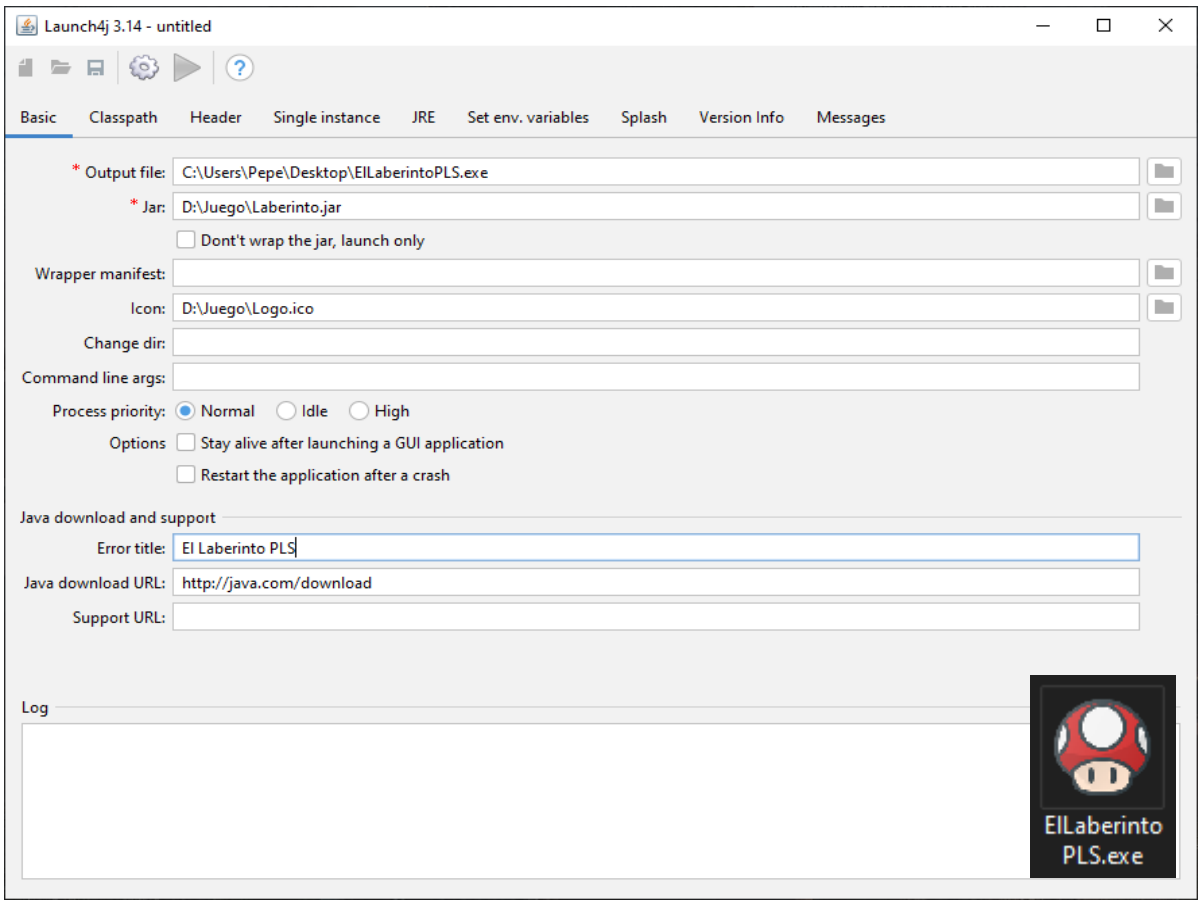
Para ejecutar el archivo .jar, usaremos el comando desde el CMD, **java -jar “nombre_archivo”**



Launch4j

Por último usaremos la siguiente aplicación para crear el ejecutable de nuestro juego, para ello necesitaremos el archivo .jar, un icono en formato .ico e indicar al versión de java necesaria para ejecutar nuestro programa.

En caso de no disponer de Java, nos redireccionará a la página oficial para que lo descarguemos



CÓDIGO

Lanzador

Destacar que en el archivo que lanza el juego he añadido una restricción de FPS, ya que una vez creado el ejecutable al pasar el juego a un equipo con mejor equipamiento la cámara de juego se movía a diferente velocidad.

También restrinjo que se pueda redimensionar la pantalla, ya que me interesa que la pantalla tenga las medidas indicadas.

Lwjgl3Launcher.kt

```kotlin
package juego.pls.com.lwjgl3

import com.badlogic.gdx.backends.lwjgl3.Lwjgl3Application
import com.badlogic.gdx.backends.lwjgl3.Lwjgl3ApplicationConfiguration
import juego.pls.com.ALTO
import juego.pls.com.ANCHO
import juego.pls.com.Juego

fun main() {
    Lwjgl3Application(Juego(), Lwjgl3ApplicationConfiguration().apply{ this: Lwjgl3ApplicationConfiguration
        setTitle("El Laberinto")
        useVsync( vsync: true)
        setForegroundFPS(Lwjgl3ApplicationConfiguration.getDisplayMode().refreshRate)
        setForegroundFPS(60)
        setWindowedMode(ANCHO, ALTO)
        setWindowIcon("Logo128.png", "Logo64.png", "Logo32.png", "Logo16.png")
        setResizable(false)
    })
}
```

Clase Pantalla

En esta clase controlamos la música del juego además de dividir la funcionalidad del render en otras cuatro funciones.

```kotlin
override fun render(delta: Float) {
    musica.play()
    if (!PantallaSonido.musicaEncendida) musica.stop()
    leerEntrada(delta)
    actualizar(delta)
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT) // Limpia la pantalla
    sb.begin()
    dibujar(delta)
    sb.end()
    depurar(delta)
}

abstract fun leerEntrada(delta:Float)
abstract fun actualizar(delta:Float)
abstract fun dibujar(delta:Float)
abstract fun depurar(delta:Float)
```

Global

En este archivo añado todos los parámetros que van a ser utilizados a lo largo de todas las clases.

Como puede ser, las dimensiones de la pantalla, la textura de lo menús y los botones utilizados, el volumen de la música, y las dos cámaras que he usado, que ya veremos mejor explicado más adelante.

```
// Dimensiones
const val ANCHO = 1024
const val ALTO =  640

// Menu
val fondoMenus = Texture( internalPath: "Menu/FondoMenus.png")

// Volumen de Juego
var volumenMusica = 0.5f
var volumenBotones = 1f

// Botones
var skinBoton = Skin(Gdx.files.internal( path: "Menu/star-soldier-ui.json"))
var sonidoBoton: Sound = Gdx.audio.newSound( Gdx.files.getFileHandle( path: "Sonidos/BotonMenu.ogg", Files.FileType.Internal))
var sonidoBotonSalir: Sound = Gdx.audio.newSound( Gdx.files.getFileHandle( path: "Sonidos/Salir.mp3", Files.FileType.Internal))

// Música Menú
val musica: Music = Gdx.audio.newMusic(Gdx.files.getFileHandle( path: "Sonidos/MenuPrincipal.mp3", Files.FileType.Internal))

// Cámara
var camaraAccion = OrthographicCamera()
var camaraHUD = OrthographicCamera()

// Viewport
val vista = FitViewport(ANCHO.toFloat(), ALTO.toFloat(), camaraAccion)

// SpriteBatch
val sb = SpriteBatch()
```

Clase Juego

Aquí creamos la instancia del juego para cambiar entre las diversas pantallas, además de crear la variable que guarda el último mapa jugado en todo momento, para poder usar el botón "Reintentar", “Siguiente Nivel” y “Continuar”

También hacemos que la música suene durante todo el juego, excepto cuando no la necesite que lo controlaremos en dicha pantalla.

```
class Juego: Game() {
    companion object{
        lateinit var instancia: Juego
        lateinit var ultimoMapaJugado: IntArray
    }

    override fun create() {
        ultimoMapaJugado = mapa1
        setScreen(MenuPrincipal(Juego()))
        instancia = this

        // Cursor
        mostrarCursor()

        // Música
        musica.volume = volumenMusica
        musica.isLooping = true
    }
```

Cámaras

Como indiqué anteriormente he usado dos cámaras, una que sigue al jugador y es donde se produce la acción, y otra estática en la que se muestra un contador de monedas, que es la cámara HUD.

Destacar que para dibujar en una u otra cámara, antes de dibujar hay que indicar que se va a dibujar en dicha cámara, de la siguiente manera.

```
override fun actualizar(delta: Float) {
    sb.projectionMatrix = camaraAccion.combined
    mapa.actualizar(delta)
    sb.projectionMatrix = camaraHUD.combined
    camaraAccion.update()
}

override fun dibujar(delta: Float) {
    sb.projectionMatrix = camaraAccion.combined
    mapa.dibujar(delta)
    sb.projectionMatrix = camaraHUD.combined
    monedaContador.dibujar()
    fuente.draw(sb,  str: "$contadorMonedas / 3",  x: 40f,  y: ALTO - 12f)
}
```

Al principio de la pantalla tendremos que indicar la posición donde queremos que se encuentren apuntando ambas cámaras, en el caso de la cámara de acción, le indicamos que se centre en las coordenadas de la salida, al igual que el jugador, y la cámara HUD la centramos en un plano centra, sin movimiento.

```
19    override fun show() {
20        ocultarCursor()
21        contadorMonedas = 0
22        musica.volume = 0.1f
23        mapa.show()
24        camaraAccion.setToOrtho( yDown: false, mapa.salida.x, mapa.salida.y)
25        camaraHUD.setToOrtho( yDown: false, ANCHO.toFloat(), ALTO.toFloat())
26    }
```

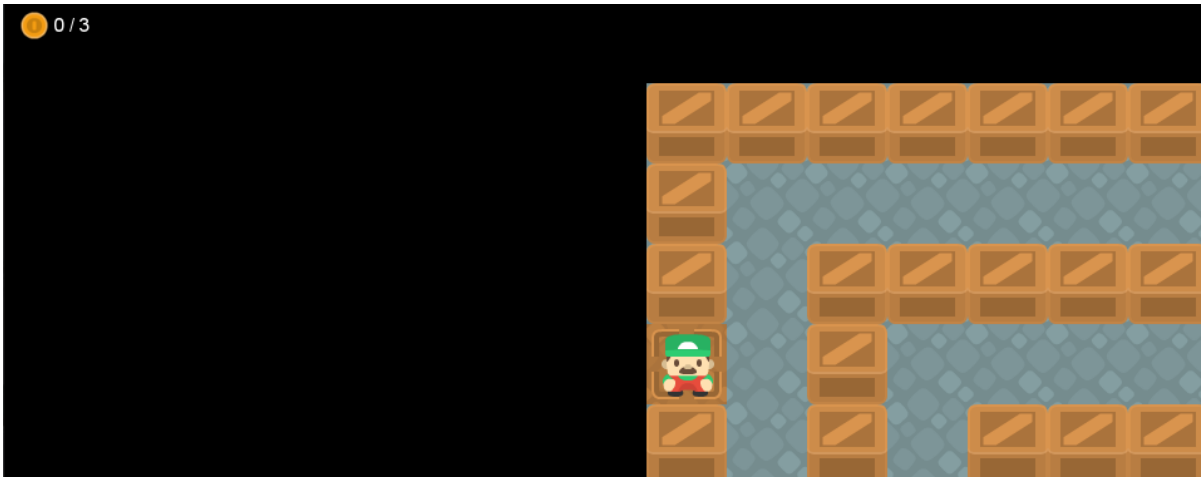
Movimiento Cámaras

Para el movimiento de las cámaras, que se moverá al mismo tiempo que el jugador, iremos trasladando su posición a medida que pulsemos las teclas de movimiento.

Para controlar que cuando el jugador choque, la cámara no siga moviéndose, he tenido que guardar la posición anterior de dicha cámara e ir modificando su valor. Lo mostraré mas adelante en el movimiento del jugador.

```
override fun leerEntrada(delta: Float) {
    mapa.leerEntrada(delta)
    if ((Gdx.input.isKeyPressed(Input.Keys.ESCAPE))) {
        Juego.instancia.screen.dispose()
        Juego.instancia.screen = MenuPrincipal(game = Juego())
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.SPACE))){
        camaraAccion.zoom = 2f
    } else {
        camaraAccion.zoom = 1f
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.LEFT) || Gdx.input.isKeyPressed(Input.Keys.A))) {
        camaraAccion.translate( x: - 5f,  y: 0f)
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.RIGHT) || Gdx.input.isKeyPressed(Input.Keys.D))) {
        camaraAccion.translate( x: 5f,  y: 0f)
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.UP) || Gdx.input.isKeyPressed(Input.Keys.W))) {
        camaraAccion.translate( x: 0f,  y: 5f)
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.DOWN) || Gdx.input.isKeyPressed(Input.Keys.S))) {
        camaraAccion.translate( x: 0f,  y: - 5f)
    }
    camaraAccion.update()
}
```

HUD



Zoom

Dejando presionada la tecla “espacio”, cambiaremos el zoom del mapa.

Cursor

Uso dos funciones para modificar el cursor, ya que no funciona la línea de codigo “None” para la versión que estamos utilizando, por lo que en una de las funciones cambio la imagen del cursor a una imagen vacía.

```
// Cursor
fun mostrarCursor() {
    val texturaRaton = Pixmap(Gdx.files.internal( path: "Menu/cursor.png"));
    val xPosicion = texturaRaton.width / 2 - 15
    val yPosicion = texturaRaton.height / 2 - 15
    val cursor = Gdx.graphics.newCursor(texturaRaton, xPosicion, yPosicion);
    Gdx.graphics.setCursor(cursor)
    texturaRaton.dispose()
}

fun ocultarCursor() {
    val texturaRaton = Pixmap(Gdx.files.internal( path: "Menu/cursorVacio.png"));
    val xPosicion = texturaRaton.width / 2 - 15
    val yPosicion = texturaRaton.height / 2 - 15
    val cursor = Gdx.graphics.newCursor(texturaRaton, xPosicion, yPosicion);
    Gdx.graphics.setCursor(cursor)
    texturaRaton.dispose()
}
```

Botones

Para el caso de los botones, para no tener que crear los botones uno a uno repitiendo código, uso una función que se encarga de darle estética y funcionalidad.

```
fun crearBoton(texto: String, skin: Skin, etiqueta: Label, juego: Juego, xPosicion: Int, yPosicion: Int, salir: Boolean, pantallaSiguiente: Pantalla?): TextButton {
    val boton = TextButton(texto, skin)
    boton.setPosition( x: etiqueta.originX - xPosicion,  y: etiqueta.originY + yPosicion)
    boton.width = 400f
    boton.height = 80f
    boton.addListener(object: InputListener() {
        override fun touchDown(event: InputEvent, x: Float, y: Float, pointer: Int, button: Int): Boolean {
            sonidoBoton.play(volumenBotones)
            return true
        }
        override fun touchUp(event: InputEvent, x: Float, y: Float, pointer: Int, button: Int) {
            Juego.instancia.screen.dispose()
            if (salir) {
                sonidoBotonSalir.play(volumenMusica)
                Thread.sleep( millis: 2 * 1000)
                exitProcess( status: 0)
            }
            Juego.instancia.screen = pantallaSiguiente
        }
    })
    return boton
}
```

Menú Principal

Para crear los menús, he utilizado un escenario en el cual creo una tabla en el centro y le añado los diferentes botones o etiquetas como actores.

Todos los menús están hechos de la misma forma, excepto el menú de sonido, en el cuál le añado más funcionalidad a los botones.

MenuPrincipal.kt

```
class MenuPrincipal(game: Juego): Pantalla() {
    var juego: Juego
    private var escenario: Stage = Stage()

    init {
        this.juego = game
    }

    private fun pantallaMenu() {
        val tabla = Table()
        tabla.setPosition( x: ANCHO / 2f,  y: ALTO / 2f)
        tabla.setFillParent(true)
        tabla.height = 400f
        escenario.addActor(tabla)

        // Etiqueta de texto
        val etiqueta = Label( text: "El Laberinto", skinBoton)
        tabla.addActor(etiqueta)
        etiqueta.setPosition( x: etiqueta.originX - 90,  y: etiqueta.originY + 200)

        // Botón Play
        val botonPlay = crearBoton( texto: "Nuevo / Continuar", skinBoton, etiqueta, juego,  xPosicion: 200,  yPosicion: 30,  salir: false, PantallaAccion(juego, Juego.ultimoMapaJugado))
        tabla.addActor(botonPlay)

        // Botón Mapas
        val botonMapas = crearBoton( texto: "Mapas", skinBoton, etiqueta, juego,  xPosicion: 200,  yPosicion: - 30,  salir: false, PantallaMapas(juego))
        tabla.addActor(botonMapas)

        // Botón Configuración
        val botonConfiguracion = crearBoton( texto: "Ajustes", skinBoton, etiqueta, juego,  xPosicion: 200,  yPosicion: - 90,  salir: false, PantallaAjustes(juego))
        tabla.addActor(botonConfiguracion)

        // Botón Salir
        val botonSalir = crearBoton( texto: "Salir", skinBoton, etiqueta, juego,  xPosicion: 200,  yPosicion: - 150,  salir: true,  pantallaSiguiente: null)
        tabla.addActor(botonSalir)

        Gdx.input.inputProcessor = escenario
    }
```

Función Siguiente Nivel

Para controlar el siguiente nivel, creo una función usando funciones de kotlin, comprobando si el ultimo mapa jugado es el mapa 1, mapa 2 o mapa 3 y le indico cual será el siguiente.

```
PantallaVictoria.kt
        // Botón Siguiente Nivel
        val botonSiguiente = crearBoton( texto: "Siguiente Nivel", skinBoton, etiquetaGameOver, juego, xPosicion: 200, yPosicion: 30, salir: false, PantallaAccion(juego, siguienteNivel()))
        tabla.addActor(botonSiguiente)

        // Botón Mapas
        val botonMapas = crearBoton( texto: "Mapas", skinBoton, etiquetaGameOver, juego, xPosicion: 200, yPosicion: - 30, salir: false, PantallaMapas(juego))
        tabla.addActor(botonMapas)

        // Botón Menú Principal
        val botonMenu = crearBoton( texto: "Menu Principal", skinBoton, etiquetaGameOver, juego, xPosicion: 200, yPosicion: - 90, salir: false, MenuPrincipal(juego))
        tabla.addActor(botonMenu)

        Gdx.input.inputProcessor = escenario
    }

    fun siguienteNivel(): IntArray {
        var siguiente = mapa1
        if (Juego.ultimoMapaJugado.contentEquals(mapa1)) siguiente = mapa2
        if (Juego.ultimoMapaJugado.contentEquals(mapa2)) siguiente = mapa3
        return siguiente
    }
```

Cajas

Las cajas son los elementos que conforman cada uno de los limites de los diferentes mapas. Todas las cajas trabajan de la misma forma.

```
class CajaMarron(x: Float, y: Float): Sprite(tipoCaja) {
    val mascara = Rectangle()

    companion object {
        private val tipoCaja = Texture( internalPath: "Mapas/CajaMarron.png")
    }

    init {
        setPosition(x, y)
        mascara.set(x, y, width, height)
    }

    fun actualizar(delta: Float) {
        mascara.x = x
        mascara.y = y
    }

    fun dibujar() {
        draw(sb)
    }
}
```

Monedas

En el caso de las monedas, la máscara que utiliza es de tipo circular.
Más adelante enseño como lo importante que es esto a la hora de colisionar con otros objetos.

Moneda.kt

```kotlin
package juego.pls.com.objetos

import ...

class Moneda(x: Float, y: Float): Sprite(monedaTextura) {
    private var sonidoMoneda: Sound = Gdx.audio.newSound(Gdx.files.getFileHandle( path: "Sonidos/juego.pls.com.objetos.Moneda.mp3", Files.FileType.Internal))
    val mascara = Circle()
    private val vector1 = Vector2( x: 0f,  y: 0f)
    private val vector2 = Vector2( x: 19f,  y: 18f)

    companion object {
        private val monedaTextura = Texture( internalPath: "Objetos/juego.pls.com.objetos.Moneda.png")
    }

    init {
        setPosition(x, y)
        mascara.set(vector1, vector2)
    }

    fun actualizar(delta: Float) {
        mascara.x = x + 16
        mascara.y = y + 16
    }
```

Sierra

En el caso de la sierra, su máscara también es circular, y a la hora de actualizar, además de activar su animación, controlamos el rebote y la dirección de dicho rebote.

Sierra.kt

```kotlin
package juego.pls.com.objetos

import ...

class Sierra(xPosicion: Float, yPosicion: Float, xPosicionTope: Float, yPosicionTope: Float, var velocidad: Float): Sprite(sierraTextura) {
    private val anchoFrame = 32
    private val altoFrame = 32
    var tiempo = 0f
    val animacion = defineAnimacion(sierraTextura)
    var posicionInicialX = xPosicion
    var posicionInicialY = yPosicion
    var posicionTopeX = xPosicionTope
    var posicionTopeY = yPosicionTope
    var topey = false
    var topex = false

    val mascara = Circle()
    private val vector1 = Vector2( x: 0f,  y: 0f)
    private val vector2 = Vector2( x: 24f,  y: 24f)

    companion object {
        val sierraTextura = Texture( internalPath: "Objetos/Sierras.png")
    }

    init{
        setBounds(xPosicion, yPosicion, anchoFrame.toFloat(), altoFrame.toFloat())
        mascara.set(vector1, vector2)
    }

    fun actualizar(delta: Float, lateral: Boolean) {
        tiempo += 0.1f
        setRegion(animacion.getKeyFrame(tiempo,  looping: true))
        if (!lateral) reboteVertical(delta)
        else reboteHorizontal(delta)
        mascara.x = x + 16
        mascara.y = y + 16
    }
```

Uso dos métodos diferentes dependiendo del movimiento de la sierra, usando siempre la misma animación.

```
    fun reboteVertical(delta: Float) {
        if (y >= posicionInicialY) {
            topey = false
        }
        if (y <= posicionTopeY) {
            topey = true
        }
        if (!topey) y -= velocidad * delta
        if (topey) y += velocidad * delta
    }

    fun reboteHorizontal(delta: Float) {
        if (x >= posicionInicialX) {
            topex = false
        }
        if (x <= posicionTopeX) {
            topex = true
        }
        if (!topex) x -= velocidad * delta
        if (topex) x += velocidad * delta
    }

    private fun defineAnimacion(textura: Texture): Animation<TextureRegion> {
        val duracionFrame = 0.1f
        val columnas = 4
        val filas = 1
        val provisional = TextureRegion.split(textura, anchoFrame, altoFrame)
        val fotogramas = Array<TextureRegion>()
        for (i in 0 ≤ until < filas) {
            for (j in 0 ≤ until < columnas) {
                fotogramas.add(provisional[i][j])
            }
        }
        return Animation<TextureRegion>(duracionFrame, fotogramas)
    }
}
```

Pinchos

Los pinchos tienen una máscara rectangular, y se son activados a través de la llave, que una vez cogida, hacen que los pinchos modifiquen su posición poco a poco abriéndose y dejando pasar al jugador.

PinchosAbajo.kt

```kotlin
package juego.pls.com.objetos

import com.badlogic.gdx.graphics.Texture
import com.badlogic.gdx.graphics.g2d.Sprite
import com.badlogic.gdx.math.Rectangle
import juego.pls.com.sb

class PinchosAbajo(x: Float, y: Float): Sprite(pinchosAbajoTextura) {
    val mascara = Rectangle()
    private val velocidad = 30f
    var esconder = false
    var posicionTope = y - 64

    companion object {
        private val pinchosAbajoTextura = Texture( internalPath: "Objetos/PinchosAbajo.png")
    }

    init {
        setPosition(x, y)
        mascara.set(x, y,  width: width - 30, height)
    }

    fun actualizar(delta: Float) {
        if (esconder && y >= posicionTope) y -= velocidad * delta
        mascara.x = x + 20
        mascara.y = y
    }

    fun dibujar() {
        draw(sb)
    }

    fun esconder(delta: Float) {
        esconder = true
    }
}
```

Mapas

Mediante la siguiente función conseguimos que se llene el array de “Sprite” con las diferentes cajas, partiendo del array de enteros que generé anteriormente con Tiled.

```
private fun rellenarCeldas(): Array<Sprite?> {
    var x = 0f
    var y = altoMapa - 64f
    val arrayBloques = arrayOfNulls<Sprite>(numeroBloques)
    var contador = 0
    for (celda in mapaEnteros) {
        if (celda == 7) {
            val caja = CajaMarron(x, y)
            arrayBloques[contador] = caja
        }
        if (celda == 8) {
            val caja = CajaRoja(x, y)
            arrayBloques[contador] = caja
        }
        if (celda == 9) {
            val caja = CajaAzul(x, y)
            arrayBloques[contador] = caja
        }
        if (celda == 10) {
            val caja = CajaVerde(x, y)
            arrayBloques[contador] = caja
        }
        if (celda == 11) {
            val caja = CajaGris(x, y)
            arrayBloques[contador] = caja
        }
        x += anchoFrame
        if (x >= anchoMapa) {
            x = 0f
            y -= altoFrame
        }
        if (y < 0f) {
            break
        }
        contador++
    }
    return arrayBloques
}
```

Colisiones Mapa

Aquí controlo las colisiones entre el jugador y los diferentes objetos dependiendo de la máscara que tengan definida.

```
if (jugador.mascara.overlaps(pinchosArriba.mascara) || jugador.mascara.overlaps(pinchosAbajo.mascara)) {
    sonidoEliminado.play(volumenMusica)
    Thread.sleep( millis: 4 * 1000)
    Juego.instancia.screen.dispose()
    Juego.instancia.screen = PantallaGameOver(game = Juego())
}
if (jugador.mascaraC.overlaps(sierra1.mascara) || jugador.mascaraC.overlaps(sierra2.mascara) || jugador.mascaraC.overlaps(sierra3.mascara)
    sonidoEliminado.play(volumenMusica)
    Thread.sleep( millis: 4 * 1000)
    Juego.instancia.screen.dispose()
    Juego.instancia.screen = PantallaGameOver(game = Juego())
}
if (jugador.mascara.overlaps(meta.mascara)) {
    sonidoVictoria.play(volumenMusica)
    Thread.sleep( millis: 6 * 1000)
    if (Juego.ultimoMapaJugado.contentEquals(mapa3)) {
        Juego.instancia.screen.dispose()
        Juego.instancia.screen = PantallaCreditos(game = Juego())
        Juego.ultimoMapaJugado = mapa1
    } else {
        Juego.instancia.screen.dispose()
        Juego.instancia.screen = PantallaVictoria(game = Juego(), contadorMonedas)
    }
}
```

Jugador

Animación del jugador. Dependiendo de la tecla que pulsemos para moverlo hará una animación u otra.

```
private fun defineAnimacion(textura: Texture): Animation<TextureRegion> {
    val duracionFrame = 0.1f
    val columnas = 3
    val filas = 1
    val provisional = TextureRegion.split(textura, anchoFrame, altoFrame)
    val fotogramas = Array<TextureRegion>()
    for (i in 0 ≤ until < filas) {
        for (j in 0 ≤ until < columnas) {
            fotogramas.add(provisional[i][j])
        }
    }
    return Animation<TextureRegion>(duracionFrame, fotogramas)
}
```

Para mover al jugador, modificamos la dirección.

```
fun leeTeclado(delta: Float) {
    tiempo += delta
    direccion.set(0f, 0f)
    if (!Gdx.input.isKeyPressed(Input.Keys.ANY_KEY)) {
        setRegion(ultimoPaso)
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.LEFT) || Gdx.input.isKeyPressed(Input.Keys.A)) && x > 0) {
        setRegion(detras.getKeyFrame(tiempo,  looping: true))
        ultimoPaso = detras.getKeyFrame( stateTime: 0f)
        direccion.x--
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.RIGHT) || Gdx.input.isKeyPressed(Input.Keys.D)) && x < Mapa.anchoMapa - width) {
        setRegion(delante.getKeyFrame(tiempo,  looping: true))
        ultimoPaso = delante.getKeyFrame( stateTime: 0f)
        direccion.x++
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.UP) || Gdx.input.isKeyPressed(Input.Keys.W)) && y < Mapa.altoMapa - height) {
        setRegion(subir.getKeyFrame(tiempo,  looping: true))
        ultimoPaso = subir.getKeyFrame( stateTime: 0f)
        direccion.y++
    }
    if ((Gdx.input.isKeyPressed(Input.Keys.DOWN) || Gdx.input.isKeyPressed(Input.Keys.S)) && y > 0) {
        setRegion(bajar.getKeyFrame(tiempo,  looping: true))
        ultimoPaso = bajar.getKeyFrame( stateTime: 0f)
        direccion.y--
    }
}

fun actualizar(delta: Float) {
    x += direccion.x * velocidad * delta
    y += direccion.y * velocidad * delta
    mascara.x = x + 12
    mascara.y = y + 5
    mascaraC.x = x + 32
    mascaraC.y = y + 32
}
```

Colisión Jugador

Para controlar el choque del jugador con los muros, lo haremos modificando su posición, guardando la anterior en todo momento.

Al igual con las cámaras.

```
private fun choqueJugador(jugador: Jugador) {
    var bloqueo = false
    if (jugador.x < 0f) {
        camaraAccion.position.x = jugador.posicionAnteriorCamara.x
        camaraAccion.position.y = jugador.posicionAnteriorCamara.y
    }
    for (celda in mapa) {
        if (celda != null) {
            if (celda is CajaMarron) {
                if (jugador.mascara.overlaps(celda.mascara)) {
                    jugador.x = jugador.posicionAnterior.x
                    jugador.y = jugador.posicionAnterior.y
                    camaraAccion.position.x = jugador.posicionAnteriorCamara.x
                    camaraAccion.position.y = jugador.posicionAnteriorCamara.y
                    bloqueo = true
                    break
                }
            }
```

Máscaras Jugador

Para la colisión del jugador con el entorno, he necesitado dos máscaras diferentes, ya que para controlar el choque con objetos de máscara rectangular y circular, es necesario que el jugador tenga la misma máscara o tratarlo con intersecciones, que a nivel de rendimiento sería algo peor.

Jugador.kt

```
15
16  class Jugador(x: Float, y: Float): Sprite(quieto) {
17      var tiempo = 0f
18      private val anchoFrame = 64
19      private val altoFrame = 64
20      private val subir = defineAnimacion(arriba)
21      private val bajar = defineAnimacion(abajo)
22      private val delante = defineAnimacion(derecha)
23      private val detras = defineAnimacion(izquierda)
24      var ultimoPaso: TextureRegion = bajar.getKeyFrame( stateTime: 0f)
25
26      var posicionAnterior = Vector2( x: 0f,  y: 0f)
27      var posicionAnteriorCamara = Vector2( x: 0f,  y: 0f)
28      private val direccion = Vector2( x: 0f, y: 0f)
29      var velocidad = 300
30
31      val mascara = Rectangle()
32      val mascaraC = Circle()
33      private val vector1 = Vector2( x: 0f,  y: 0f)
34      private val vector2 = Vector2( x: 0f,  y: 0f)
35
36      companion object {
37          val arriba = Texture( internalPath: "Jugador/Arriba.png")
38          val abajo = Texture( internalPath: "Jugador/Abajo.png")
39          val derecha = Texture( internalPath: "Jugador/Derecha.png")
40          val izquierda = Texture( internalPath: "Jugador/Izquierda.png")
41          val quieto = Texture( internalPath: "Jugador/Parado.png")
42      }
43
44      init {
45          setPosition(x,y)
46          mascara.set(x, y,  width: width - 22,  height: height - 12)
47          mascaraC.set(vector1, vector2)
48      }
```

BIBLIOGRAFÍA

Tiled
https://programmerclick.com/article/90881249473/
https://www.mapeditor.org/
https://www.genbeta.com/desarrollo/tiled-map-editor-el-editor-de-mapas-libre
http://www.pixnbgames.com/blog/libgdx/como-usar-libgdx-tiled-diseno-del-escenario/

Launch4j
https://sourceforge.net/projects/launch4j/files/launch4j-3/3.14/

TexturePackerGUI
https://www.codeandweb.com/texturepacker/documentation/user-interface-overview
https://phasergames.com/using-texturepackergui-make-sprite-sheets/

BitmapFont
https://www.jc-mouse.net/android/uso-de-bitmapfont-en-libgdx

Assets
https://www.kenney.nl/
https://iconos8.es/

Reproducción de Sonidos
https://www.jc-mouse.net/android/reproduccion-de-sonidos-con-libgdx

Cursor
https://libgdx.com/wiki/input/cursor-visibility-and-catching

Cámaras
https://libgdx.com/wiki/graphics/integrating-libgdx-and-the-device-camera