Assignment 2

Platformer: The Game

By: Carl-Johan Johansson
Course: Game Development For Android, 5SD812 54851 HT2023
Teacher: Ulf Benjaminsson

Emulation setup

The application has been developed and tested in Android Studio in a Windows 11 environment. The AVD is emulating a Pixel 6 phone with API version 34.

Implementations

The following features have been implemented and are being presented in the order they were added to the game:

5. Replacing the hardcoded values level in Test Level by loading the tile layout from a file instead

I have chosen to keep the TestLevel class (which has been renamed to LoadLevel instead) since the data must be added in some way either way. The game is in an unfinished state so it still makes sense to have a test environment to plug into the LevelManager. But instead of having the tile layout hardcoded into the class itself the level structure is now being read from a file in the assets folder of the project.

In the finished game I imagine that the LevelManager itself would be responsible for loading and unloading various levels by manipulating LoadLevel, based on some kind of choice made by the player (I'm very fond of the world maps in Super Mario Bros. 3 and Super Mario World for instance!).

2. New enemy

I've created an enemy – but it isn't static (because, well, I misread the assignment)! The enemy class, very cleverly called Enemy, is a subclass of DynamicEntity instead which provides it with move semantics. The enemy sprite simply moves up and down between two tiles. When it hits either one it changes direction. This behaviour is made possible by overriding the onCollision() function of the super class:

```
override fun onCollision(that: Entity)
{
    val overlap = PointF()
    if (getOverlap( a: this, that, overlap))
    {

        if (overlap.y != 0f)
        {
            movingDown = !movingDown
            y += overlap.y
        }
    }
}
```

When the enemy collides with another entity the value for movingDown switches around, which inverts the speed in the update():

```
override fun update(dt: Float)
{
    if (movingDown) { y += speed }
    else { y -= speed }
}
```

The player has also been given a life bar represented by three little heart, similar to a Zelda game. He has also been given a few recovery frames where he becomes invulnerable after getting hit by the enemy. To communicate this visually the player sprite starts blinking.

This behaviour is implemented by using something called Kotlin Coroutines, because the book claimed that doing this asynchronous often makes for better performance than lots of context switching (I have no idea if this is true, but it sounded appealing).

In order for Coroutines to work I had to add an implementation to my dependencies in the build.gradle file in the Gradle Scripts folder:

```
dependencies { this: DependencyHandlerScope

    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.4")
    implementation("androidx.core:core-ktx:1.9.0")
    implementation("androidx.appcompat:appcompat:1.6.1")
    implementation("com.google.android.material:material:1.9.0")
    implementation("androidx.constraintlayout:constraintlayout:2.1.4")
    testImplementation("junit:junit:4.13.2")
    androidTestImplementation("androidx.test.ext:junit:1.1.5")
    androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
}
```

Since the LevelManager is responsible for the player object it felt apt that this class should also be responsible for how long the player is invulnerable (this way it could also use the same coroutine implementation in the future for other things in the world that might need a delay).

I've placed the coroutine scope in checkCollisions() since it has no further usage to me, but it could also easily be refactored into its own method which might take the time delay as an argument:

```
if(isColliding(player, enemy) )
{
    if (!player.isInvulnerable)
    {
        player.takesDamage(damageAmount)
        player.isInvulnerable = true
        player.isBlinking = true
        CoroutineScope(Dispatchers.Default).launch{ this: CoroutineScope
            delay(invulnerabilityTime)
            player.isInvulnerable = false
            player.isBlinking = false
        }
    }
    enemy.onCollision(player)
    player.onCollision(enemy)
}
```

The blinking logic exists in the Player class and is essentially a simple system that decrements a blinking interval in each update() call, and turns on and off a boolean. While the boolean is false the render() method will render the player sprite. While the value is true it will skip the rendering. Once the coroutine delay is finished the code will permanently set the isBlinking boolean to false again, and it will remain so until another hit is detected.

4. HUD to show health bar and messages on screen

Just like in the previous task we have a HUDRenderer class that is responsible for drawing information on the screen.

The health bar is represented by hearts in the upper-left corner of the screen. The bitmaps for the hearts are loaded on initialization, ensuring that this only happens once:

```
    init
    {
        loadAssets(context)
    }

    @SuppressLint("UseCompatLoadingForDrawables")
    private fun loadAssets(context: Context)
    {
        val res = context.resources
        val fullHeartDrawable = res.getDrawable(R.drawable.hearth_full, theme: null)
        val halfHeartDrawable = res.getDrawable(R.drawable.hearth_half, theme: null)
        fullHeartBitmap = (fullHeartDrawable as BitmapDrawable).bitmap
        halfHeartBitmap = (halfHeartDrawable as BitmapDrawable).bitmap
    }
```

Depending on how much health the player has left the HUD then draws a corresponding number of hearts onto the screen, along with a counter showing how many coins the player has picked up. Whenever the player's health is decremented he loses half a heart. The code to achieve this was more complicated to figure out than I imagined it would be. There might a simpler way to do it, but this solution works, and the golden rule of programming (which I just made up) states: never fix anything unless it's broken… and even then, only fix it if it's really noticeable:

```
if (!isGameOver)
{
    val maxHearts = 3
    val playerHealth = levelManager.player.health
    val fullHearts = playerHealth / 2
    val displayHalfHeart: Boolean = playerHealth % 2 == 1

    for (i in 0 ≤ until < maxHearts)
    {
        val heartX = x + i * (heartSize + 10)

        if (i < fullHearts)
        {
            canvas.drawBitmap(fullHeartBitmap, heartX, y, paint)
        }
        else if (i == fullHearts && displayHalfHeart)
        {
            canvas.drawBitmap(halfHeartBitmap, heartX, y, paint)
        }
    }
    canvas.drawText(coinCountText, textX, textY, paint)
}
```

3. Dynamic collectible entity

I've made a new class called Coin, which is a subclass of StaticEntity. The player object contains a variable that keeps track of how many coins that have been collected (that is to say: how many coin entities the player has collided with).

In the checkCollisions() method of the LevelManager class we then call collectCoin() from the player object whenever the player collides with an entity that is a coin, and we also make sure to remove the coin from the level:

```
for(e in entities)
{
    if(e == player) { continue }
    if(e == enemy) { continue }

    if (e is Coin && isColliding(player, e))
    {
        player.collectCoin()
        removeEntity(e)
        return
    }
```

This way we also manage to avoid undesirable behaviour, such as the player stopping dead in his tracks every time he collides with a coin.

6. Sound

For the background music I experimented a little bit with old tunes. I kinda wanted music that was swinging because the chubby player sprite reminds me a little bit of the character in Mr. Gimmick, which has a beautiful soundtrack that was heavily inspired by jazz.

I took Benny Goodman's old swing classic Stompin' at the Savoy (which was recorded in 1932 and no longer is under copyright protection) and converted it to MIDI format with the use of an online converter, and then I ran the resulting file through a free program called GXSCC, which (among other things) has a Famicom preset that transforms the music into a facsimile of a NES tune.

The result ended up much better than I thought it would so I decided to keep it! By spending a little more time on the editing I'm sure the music could have been improved further – there are some parts where the sound drops a little bit because the conversion to MIDI meant that some instruments were lost – but for the purpose of this short project it was a quick and dirty fix that gave the game a unique touch. I dunno, this might be a good tip for students who want to customise their background music and feel that the free selection on the web doesn't have the right feeling they are going for.

To actually get the music to play I've implemented the better sound engine described in the article under the course modules. The media player is used to play the background music while the sound effects are loaded directly by the Jukebox and mapped to Game Events such as jumping, collision and collecting.

Per the recommendation in the article I've created a dynamic list, soundEvents, where every sound effect is added once onGameEvent() is being called. The list is then looped through and each event sound in it is played at the end of each frame:

```
fun onGameEvent(event: GameEvent, e: Entity?)
{
    soundEvents.add(event)
}

private fun playSoundEvents()
{
    for(event in soundEvents)
    {
        jukebox.playEventSound(event)
    }
    soundEvents.clear()
}
```

We're not using the entity in this case so it could be omitted as an argument, but I figured I'd keep it for now, just in case I want to do something more expansive with it later on as I keep experimenting.

On game over

Nothing really happens when you collect all coins. Since the game is just a tiny demo there's no real win state, which I think is fine. The alternative would be to just put some text on the screen once all the coins have been collected, but it feels a bit pointless. I did however think it was prudent to add a reset feature on Game Over, with an associated restart sound.

The checkGameOver() function activates a listener if isGameOver is true, and once the player touches the screen outside of the controls the listener runs the reset() function:

```
private fun reset()
{
    jukebox.pauseBgMusic()
    jukebox.unloadMusic()
    levelManager = LevelManager(LoadLevel())
    isGameOver = false
    jukebox.loadMusic()
    jukebox.resumeBgMusic()
    jukebox.playEventSound(GameEvent.Restart)
}
```