Tux

Luísa Amaral, 93001, Maria Cunha, 93089

January 4, 2023

1 Game overview

This project is a partial recreation of Super Tux, a widely known open source 2D platform game. Similarly to the original game, this recreation also renders a side scroller where the user can control the penguin Tux, the hero of the game. The game features two types of enemies, several platforms, a power-up that makes Tux grow, coins and different levels and the goal in each level is to reach Tux's home. Tux can step on Snowball enemies to kill them, but he cannot step on Spiky enemies. Every time Tux catches a coin or kills and enemy, the scoreboard increases the number of coins or the number of enemies, respectively. If Tux happens to fall out of the platforms to the abyss or if an enemy hits him while he is small, he dies. To grow, Tux has to catch an egg, which can be obtained by breaking the correct bonus boxes. There are three types of platforms that assist Tux in reaching his home: snow, wood and flying platforms.

Figure 1: Screenshot of the game

2 Project Architecture

In the root folder, two folders can be found: the *assets* folder and the *src* folder.

The *assets* folder holds all the sprites needed for the game, divided by folders corresponding to the object or objects that those sprites represent, as well as the text font and the background of the game. Its structure is as follows:

- blocks folder - Sprites of the snow and wood platforms' blocks
- bonus_block folder - Sprites of the bonus blocks

- coin folder - Sprites of a coin
- creatures folder - Spritesheets with images of Tux and the enemies
- drafts folder - *xcf* files with drafts of the platform's tiles
- flying_platform folder - Sprites of the flying platforms
- goal folder - Sprites of the goal object and Tux's home
- levels folder - Images with byte maps that define the components of each level
- scoreboard folder - Sprites of the symbols used in the scoreboard
- arctis.jpg - Background used in every level
- SuperTux-Medium.ttf - game font

The *src* folder contains the code necessary for the game to run. Its structure is as follows:

- agent.py - Contains the class that defines the Agent object, and this class will be inherited by the Tux and Monster objects
- bonus_block.py - Contains the class that defines a bonus block, which is a block that holds a coin or a power up and can be broken by Tux when it is hit from below
- coin.py - Contains the coin class
- common.py - Contains the Directions class and the Subject class that is used in the observer pattern, as well as some global variable declarations that are common to several files
- fsm.py - Contains the class that defines a finite state machine
- goal.py - Contains the class that defines the goal object that Tux needs to reach each level
- level.py - Contains the Class that defines a Level. This class has several attributes that store the level's component states and positions
- main.py - Main class where the game loop is present
- monster.py - Contains the Class that defines the Monster object and the classes that define each enemy
- platforms.py - Contains the Class that defines a platform and two classes that inherit it and that define the wood and snow platforms. It also contains the definition of the class of the flying platform
- scoreboard.py - Class that defines a scoreboard that will present the number of coins caught and the number of enemies killed
- spritesheet.py - Auxiliary class that allows the parsing of spritesheets
- tiles.py - Enum class that defines what object each colour represents in the level map images
- tux.py - Contains the Class that defines the Tux object

The root folder also contains the file *requirements.txt* that contains the packages necessary to run the game.

3 Game Patterns

This section describes each game pattern that was implemented in the project

3.1 Game loop

The Game Loop pattern was implemented using a loop that keeps the game running continuously until the user wants to exit. At each iteration of the loop, the game processes user input without blocking, updates the game state, and renders the game. The number of frames per second is also defined in the loop.

3.2 Double buffer

In each loop iteration, the graphics are updated and displayed on the screen. While they are being displayed, a new update is occurring and the next sprites are ready to be displayed. This method allows the sprites' animations to appear smoother and more realistic. A swap operation swaps the next and current images instantly so that the new images are now visible.

3.3 Update

Each sprite implements an update method and they are divided into groups which handle their update [1] [2] [3] [4].

3.4 Byte code

With the various objects that can appear on a game level, which includes enemies, platforms and bonus blocks, among others, it is necessary to define a way of establishing where each object of the level will be rendered when the level starts. Considering this, each level is described in a *png* file, where each coloured pixel represents an object that will be rendered in that position. For example, red pixels represent snowball enemies, with a lighter red meaning that the enemy will start walking facing left and a darker red meaning that it will face right. A green pixel represents a spiky enemy and a yellow pixel represents a coin, whereas a purple pixel represents a bonus box that when broken will increase the coin count. A class that inherits the Enum class was used to attribute each colour to its meaning. In Figure 2 is possible to see an example of how a level is described. When each level starts, the respective map file is parsed and the level components are rendered to the screen.

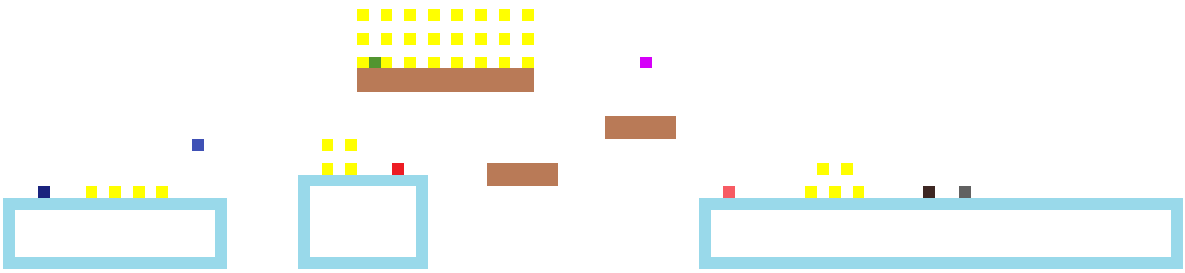


Figure 2: Level 1

3.5 Prototype

The Prototype pattern's main objective is to be able to spawn other objects similar to itself which means that if we have an instance of the object Snowball, it is possible to make more objects Snowball from it. This functionality is really helpful for instances like this, where we repeat a lot of the same objects, in this case enemies and objects, and it makes it possible to simplify the code.

For spawning enemies in the levels it is used a prototype of Snowball, a prototype of Spiky and a spawner, used to spawn enemies from prototypes [1]. For this, it was implemented a Spawner class and it was added the *clone* method, which is called by the spawner, to every enemy class [1] [2].

3.6 Observer

The Observer method allows the definition of a subscription mechanism where notifications are sent to one or multiple objects regarding any event that happens to the object that the observer is observing. Considering this, a scoreboard was created where the number of coins caught and of enemies killed is shown, and it behaves as an observer that receives a notification every time one of these events happens.

To implement this, a Subject class was created. This class stores the events that will be received, as well as the observer's methods that should be executed when a specific event happens.

The Tux class inherits this Subject class and when it is detected that Tux caught a coin or killed an enemy a notification is sent with the respective event [1] [2], which will lead to the execution of the scoreboard methods that increment the coin count and the enemies kill count, respectively. When the scoreboard is rendered in the next game loop, the scores will be updated.

3.7 State

The State pattern permits an object to change its behaviour according to its internal state.

The State pattern is generally associated with the usage of a Finite State Machine (FSM), which is a machine defined by a set of states and a transition function. Its current state then changes based on the input and the transition function.

With this pattern, we were able to control the movement of the agents, that is: making it jump, walk in one direction or another, or stay still. Additionally, it was also used to obtain the coordinates of the respective state's sprite on the agent's spritesheet, and also kill the agent.

For this pattern, the base structure used was the one used in class, and provided in the slides [1].

From the *State* class, the states used by Tux, the main player, and by the enemies, the secondary characters were created.

3.7.1 Tux

Tux's behaviour is similar to any 2D platform side-scroller's main character, it can move to the right, to the left and jump. Apart from that, Tux is able to interact with other characters in the game, enemies, whose in contact with can kill Tux. The only way that Tux can kill the enemies is by jumping on top of them, but this action may vary depending on the enemy.

Moreover, Tux is able to change its size, when catches an egg hidden in a bonus box.

Tux has minor differences on its behaviour depending on its size. When Tux is hit by an enemy, the state transition depends on its size. That is, if Tux is 'small', Tux dies, if Tux is 'big', its size changes to 'small'.

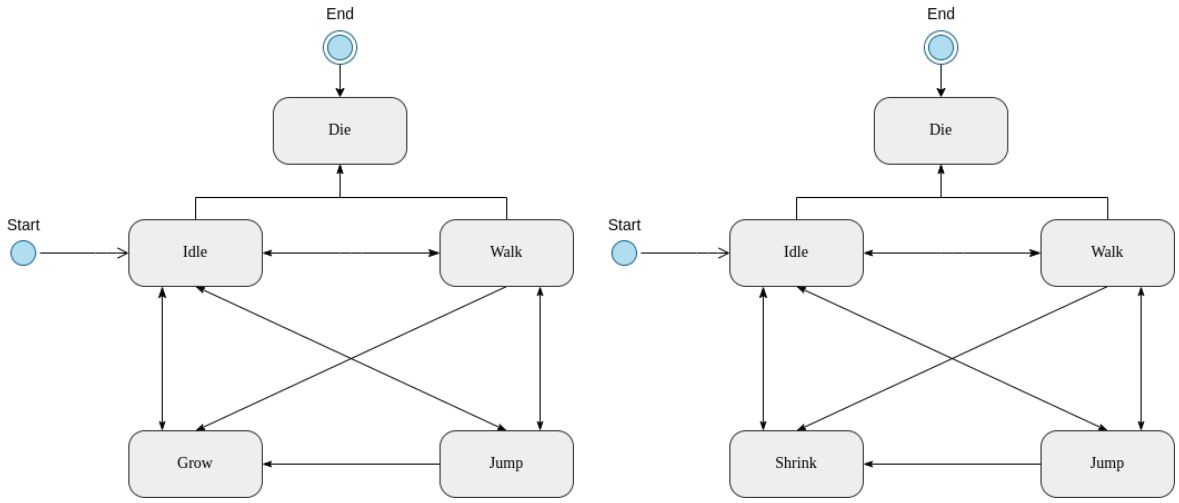
To manage this particular condition, the usage of 2 State Machines, one for each size, was implemented. When Tux is 'small', the State Machine used is *fsm_mini* (Figure 3a), when it is 'big', the State Machine used is *fsm_max* (Figure 3b). The only difference between them is one state, 'GROW' and 'SHRINK', where Tux gets 'bigger' or 'smaller', and changes the currently used state machine [1] [2].

3.7.2 Enemies

The enemies primary function is to walk around, while not falling from the platforms, and damage Tux when in contact with them. They move automatically and can be killed if Tux jumps on top of them. However, not every enemy can be killed by Tux.

The two types of enemies implemented are: Snowball and Spiky, and they are both subclasses of the class Enemy. Snowball walks around, damages Tux, when touched, and can be smashed/killed if Tux jumps on top of it. Spiky has a similar behaviour to Snowball, less the fact that it cannot be killed if jumped on top of it.

With this being said, their State Machine is particularly basic (Figure 4), since its states are only: 'IDLE', 'WALK' and 'DIE' [1] [2].



(a) FSM used by Tux when 'small'

(b) FSM used by Tux when 'big'

Figure 3: Finite State Machines used by Tux

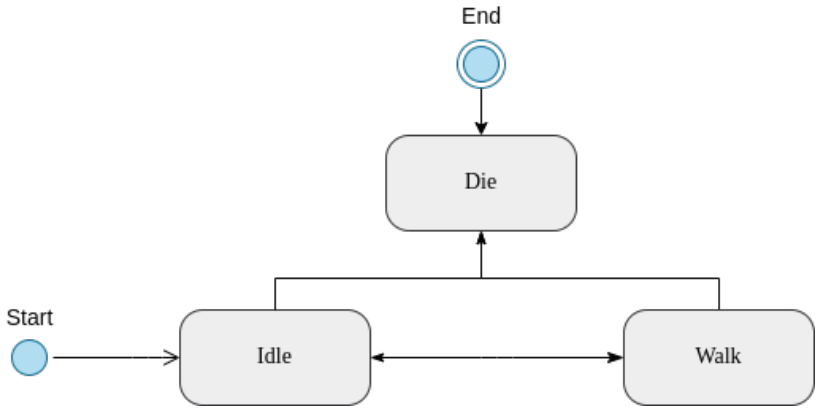


Figure 4: FSM used by the enemies

3.8 Command

The Command pattern main objective is to separate the input of the user from the implementation of the corresponding action. That is, when the user presses the arrow right key, this action is mapped to the related *Command*, in this case the "Right" command. This command will then express, to the state machine, that the user intends to move Tux to the right.

As told previously, in 3.7.1, Tux only moves to the right, to the left and jumps, so the only commands needed are: 'Up', 'Right' and 'Left' [1].

Tux's agent receives the input event key from the main game loop and maps it to the Command object. This object is then used to decide which action to make, in the *commands* function of the Tux's agent [2] [3].

4 Innovative features

Overall, the features that the project presents are similar to part of the features that the original Super Tux game presents. However, the definition of a level and its components in the original game follows a different structure from the one we used. Being a more complex game, the original Super Tux uses a higher level of detail to define each component of a level and its characteristics, and it does it in an extensive way. Since this project's level of complexity was lower, due to the goal of recreating only part of the game, each level is defined using a byte map with coloured pixels. This allows for an easier

way of building and visualizing levels before rendering them, and it translates in a faster level parsing process, since the information to acquire regarding the level is much more simpler.

5 Repository link

The link for the repository is https://github.com/LuisaTheAmaral/super-tux.

6 References

The original game's website is https://www.supertux.org/. The sprites contained in the folders *blocks*, *bonus_block*, *coin*, *flying_platform*, *goal* and *scoreboard* and the text font were obtained from the official Super Tux repository while the sprites contained in the folder *creatures* and the *artics.jpg* background were obtained in this website.