Multitasking Game

Report

Pietro Olivi
Leonardo Tassinari
Lorenzo Dalmonte

March 26, 2023

Contents

Chapter 1

Analysis

The software commissioned by the professors of the Object-Oriented Programming course aims to create an application designed to enhance psychomotor skills through a gaming experience based on multitasking.
The term *Multitasking* refers to the ability of a person or a product to do more than one thing at a time.

1.1 Requirements

Functional

- Upon starting, the software will display a simple minigame.[1]
- After a short amount of time a new minigame will appear and so on until all four minigames are shown.
- Generally, the difficulty of each minigame shall increase over time.
- The player's goal is to survive as long as possible. After failing a minigame the application will display the final result, therefore the software has to keep track of how long the player has lasted in the match in order to calculate the score.
- It will be possible to appreciate the improvement on a *Statistics* page that will show the record of all the past runs.

[1] A minigame is a potentially-never-ending challenge that requires simple actions from the player in order to keep the game going.

Non functional

- The application shall sustain acceptable frame rate (around 60 fps) in all the sections of the gameplay, even on older hardware[2].
- It shall be possible to easily develop and swap the minigames among the ones that best fit the training purposes of the user on top of those already provided.
- It shall be possible to play in full screen mode.
- The window shall be resizable to fit in any kind of screen[3].

1.2 Domain analysis

MTSK-Game must exhibit some *minigames*, the ones supplied by us are:

- *WhacAMole*: where the goal is to crush all the moles that emerge from the dug holes, before they re-enter them, avoiding detonating bombs that will also pop up from the pipes.
- *DodgeATriangle*: in which the player has to slide a *rectangle* up and down in a column switching lanes, aiming to avoid hitting moving *triangles*.
- *CatchTheSquare*: where the user should destroy *squares* running over them with a *circle* before their timers runs out. With time, multiple squares will spawn at the same time with an increasing rate.
- *FlappyBirdAlike*: where the user needs to control a *cursor* leading it to avoid *obstacles* that will come towards it.

For the game to end, and the *score* to appear, it'll be sufficient losing in only one of the currently displayed minigames.

[2] e.g. Intel Core i3 (fourth generation), 4Gb of RAM.
[3] Provided a minimum resolution.

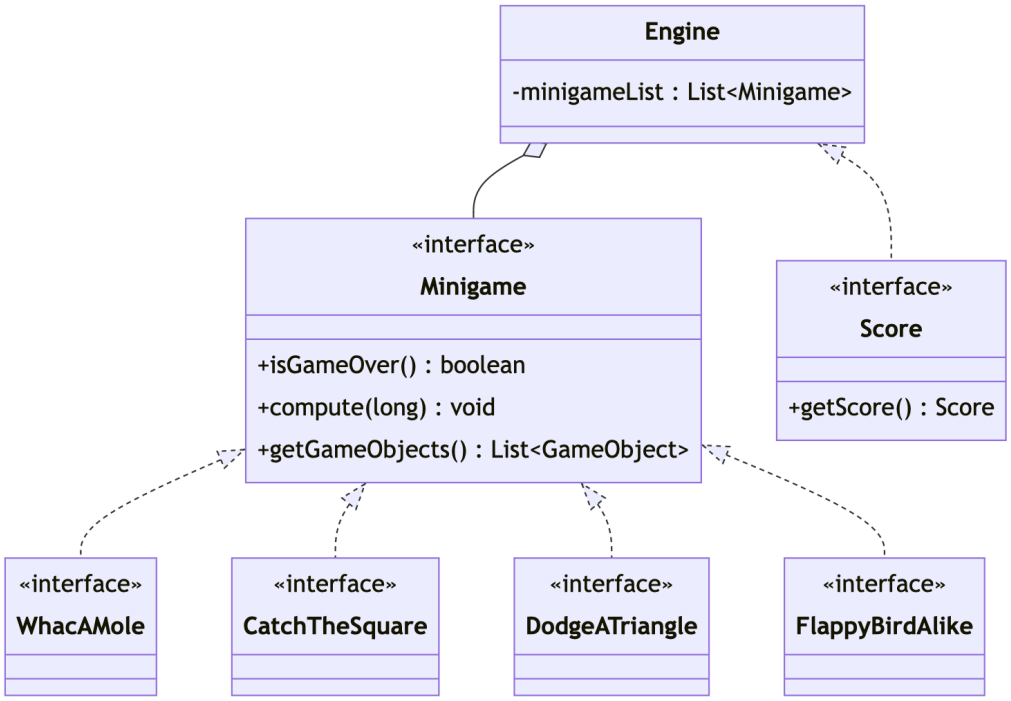

Figure 1.1: UML diagram of the domain analysis

Chapter 2

Design

2.1 Architecture

The project is built on the "model-view-controller" architecture: the **View** will be the entry point of the program. At its creation it will instantiate the game window and the **Controller** class. When the user decides to start the run it will be asked to the controller to create an **Engine**, this class will manage each **Minigame**.
The Model part is represented by the implementation of **Minigame**s these will contain **GameObject**s that will evolve over time with a specific logic. The **Controller** directs the run with the game loop pattern, updating each frame the view with the **GameObject**s fetched from the **Engine**. The **Controller** takes also care of intercepting the user's input from the **View**, then it forwards it to the Engine. Once the input has been received, the Engine is responsible for communicating it to each minigame: these in turn will update the state of all the GameObjects they contain, i.e. every single entity in the various playing fields.

2.2 Detailed design

Each minigame is composed of **GameObject**s: those items use a component pattern, thanks to which we get a full separation of concerns based on domains (allow a single entity to span multiple domains each other[1]).

[1]From GPP, CH 14

- *PhysicsModel*: Interface that deals with the physical state of a Game Object, moving it according to its speed, considering the environment in which it is located (edges of the field) and the other objects it interacts with (collision with obstacles).
- *AspectModel*: It is the interface that orders the updating of the graphics of the single object to the View, specifying which of the instructions already contained in SwingView need to be used.
- *InputModel*: Interface related to a single GameObject that reads the input stored in the engine and applies it, if the object recognizes it as its own command, changing its specifications (e.g. coordinates, speed).
- *HitBoxModel*: Defines the shape and sizes of the hit box that the object shall interpret when colliding with other hit boxes.

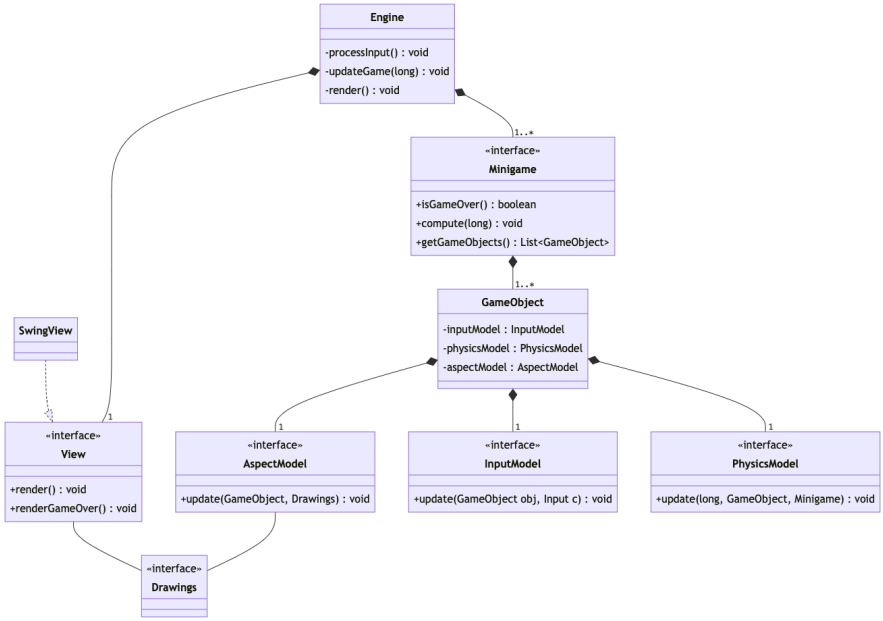

Figure 2.1: UML diagram of the architecture TODO MUST BE UPDATED

Leonardo Tassinari

Difficulty in CTS
Problem: The spawn rate of the squares shall increase over time.
Solution: The CatchTheSquare's constructor adopts the strategy pattern

taking a *Function* interface. That function will return to the minigame the number of bombs that are expected to be spawn so far given the total amount of milliseconds elapsed from the beginning of the minigame. My implementation of the function uses an exponential curve to increment the spawn rate progressively until a certain rate. The limit rate is represented by the inclination of the line tangent to the exponential curve on the point where the derivate of the exponential reaches that steepness.

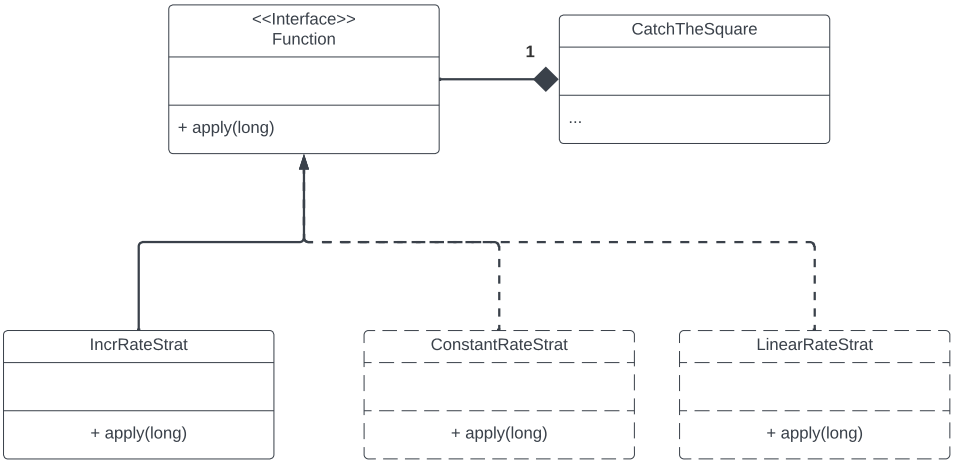

Figure 2.2: The corresponding UML diagram

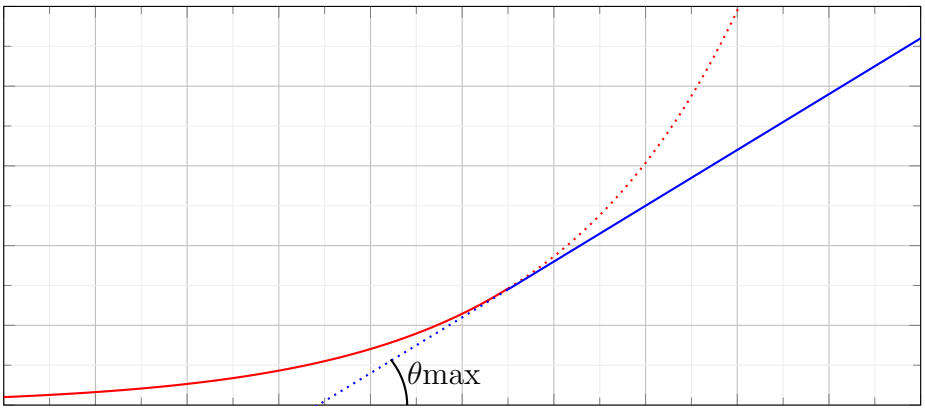

Figure 2.3: An example of the trend of the implemented function