SUPER MARIO REPORT

Mónica De Álvaro Mena
Iker Rosales Saiz
Group 96, 196
Data Science and Engineering

Contents

1. Outlook

This project has been developed in python; it is based on GitHub project "[Pyxel a retro game engine for Python]" by **Takashi Kitao.**

This library provides simple, intuitive, very valuable tools to develop games in python.

The first challenge of the project was to understand the functions and utilities that the library has, so we have followed a high-level description of the most relevant that have been used in our project:

pyxelditor -> It is a graphical tool where the user can define graphical and musical resources to use in the game as the building of sprites pixel by pixel using 16 different colors. This tool generates a file (.pyxres) with the resources that can be easily referenced and used at the game. (see *Figure 1*)

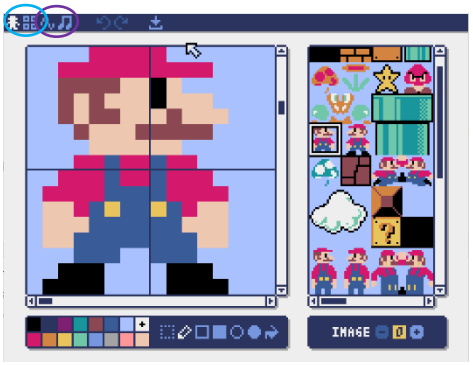


Figure 1

pyxel.init -> Initialization of the game

pyxel.load -> Loads the resource file created with "pyxeleditor"

pyxel.run -> Starts the pyxel application and calls the `update` function for frame update and the `draw` function for drawing. It will keep the game in an infinite loop. The update function is used for the game itself, and the draw function to update the screen output.

Pyxel also provides some other useful tools that can be called within the execution of the game to interact with the player, the most relevant ones that we have used are:

pyxel.cls, pyxel.blt, pyxel.bltm, pyxel.text -> for drawing purposes

pyxel.btn, pyxel.btnr, **pyxel.btnp** -> to scan the user's interaction through the key-board

pyxel.stop, pyxel.play, pysel.playm -> for sound handling

Detailed information about pyxel API can be found in the project place in GitHub: https://github.com/kitao/pyxel (Github, 2021) (GitHub User Examples, 2021)

1. Classes design

We have created classes for the different objects in the game, we can classify them according to the role played in the game:

MOTHER CLASS: First of all, we have created a **MotherClass** from which all the objects of the game that have common attributes (x, y sprite) inherit from, and where we will encapsule them.

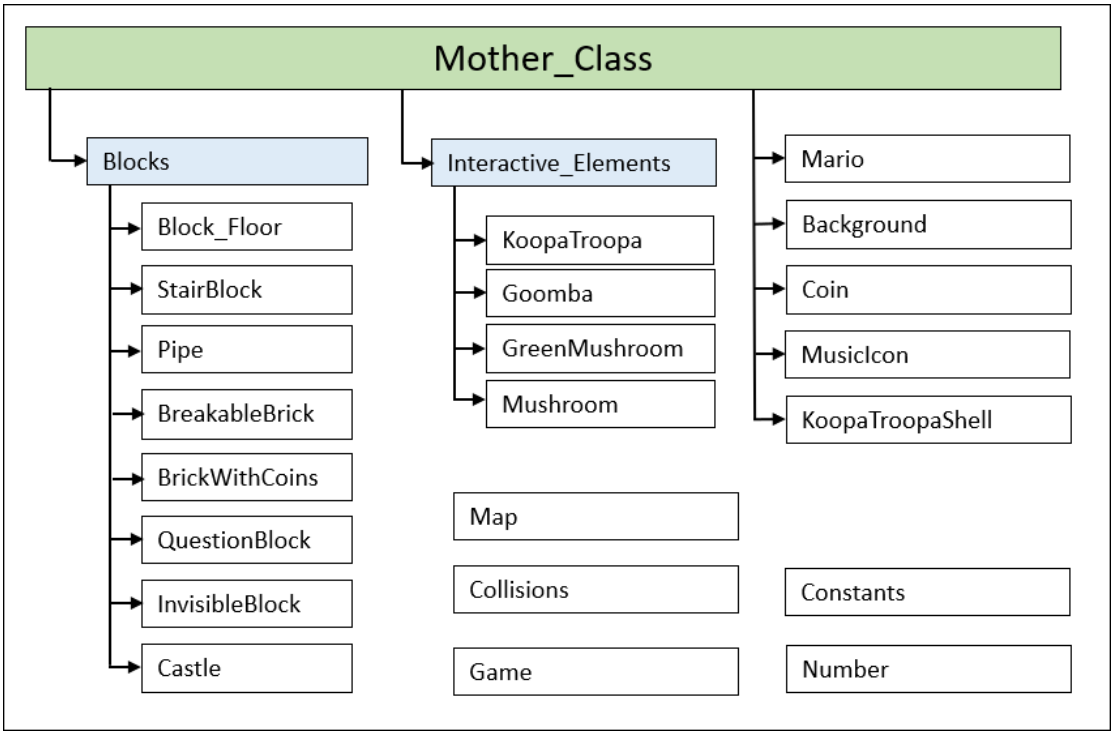
BACKGROUND ELEMENTS: Elements that are created to build the environment where the Mario will move: pipes, different type of blocks… These elements are **statically positioned** in the map and each one of them are defined in its own class or subclass.

INTERACTIVE ELEMENTS: Elements that move around the environment created along with Mario. These classes are: **Mario** itself, the enemies and the special objects that appear when Mario interacts with them, **Goomba**, **Koopa Troopa**, **red** and **green mushrooms**.

OTHER: Classes that define the game parameters and run the game (**Game**, **Constants**, **Collisions**)

Based on the previous functional classification we have the following map of classes.

We used inheritance as well as abstract classes, information hiding, encapsulation and other object-oriented techniques.



1.1 Game

This is the class that contains the **main thread** of the videogame. Inside it we find the pyxel functions **pyxel.init**, **pyxel.load** and **pyxel.run,** as stated before pyxel.run will keep the game in a loop forever, updating the game and refreshing the screen.

As Game contains the functions **update** and **draw,** it controls:

- The score, coin, and time countdown and all the game information that is represented for the player.
- Controls Mario's movement.
- Object creation, like enemies and special objects.
- Screen movement.
- Music to be played.

1.2 Constants

This class is used for data and predefined variables, important data values that are used by all other classes in the game.

Due to the use of **pyxel**, mapping graphical and music resources created with **pyxeleditor** are accessed by pointing them in the file via 5 "coordinates", having those resources mapped in the **constants** class makes it very easy to change graphical and audio resources without touching the phyton code, since we only have to modify it in one place.

1.3 Collisions

This is where we have created methods that we will call during our program which will check the interactions (or collisions) between the different objects of the game.

This class is called in every frame to check the collisions between the different elements of the game, Mario with the elements of the background like the blocks, as well as the enemies or other animated objects (like the mushrooms) and their own collisions since they also interact with the background.

1.4 Mario

This is the class for Mario where we have created methods for instance: move, orientation, jump, rebound, gravity…

1.5 Interactive Elements

This is the main class for the enemies and the mushrooms, it will be the mother class with the methods that they will share, like the movement (which includes moving with background when Mario is in the middle of the screen), and gravity. We have two subclasses of enemies **Goomba** and **KoopaTroopa**, corresponding to the two different enemies we have in our game, and two different mushrooms, **Mushroom** (which is the red one that transforms Mario into SuperMario) and **GreenMushroom** (that adds a life).

1.6 Map

In this class we will create all the elements that will belong to the map: pipes, floor, blocks, stairs... Initializing them in the corresponding positions, which will later move to the left as Mario advances in the game. The whole map is an object, which will be created in the Game class, that contains lists with all of these other objects that are mostly subclasses of the Block class.

It also has the methods that move it (with the background) and that draw all of its elements.

1.7 Blocks

This is the mother class we will use for all type of blocks that form the map.

All the block objects had similar characteristics, so the most appropriate solution was to create an abstract mother class called blocks, inside of which we created the following subclasses.

- **BlookFloor**
- **StairBlock** (with a method that creates the stairs with the individual blocks)
- **Pipe**
- **BreakableBrick**
- **BrickWithCoins** (which will have a random number of coins)
- **QuestionBlock** (which will contain either a red mushroom or coins)
- **InvisibleBlock** (with the green mushroom)
- **Castle** (that appears at the end of the game, and once Mario has reached it, the user will have won the game)

2. Most relevant fields and methods

All objects contain the following attributes:

```
self.x       -> Contains the x coordinate where it will appear on the screen.
self.y       -> Contains the y coordinate where it will appear on the screen.
self.sprite -> Contains the image of the object in the resources file.
self.width -> Contains the width of the object image.
self.height -> Contains the height of the object image
```

Regarding their methods, all objects (except for Mario) have a `move_with_background` method that we call upon to update the elements that appear on the screen, when Mario is in the middle of the screen and advancing.

Moreover, the interactive elements, like the enemies and the mushrooms also have their own `move` method, with their according speed and a `gravity` method.

Furthermore, the main methods called upon during our game are those which correspond to the **collisions**, that check those with the static elements that form the map, with the animated objects (Mario, enemies, koopa troop shell and the mushrooms), which have called `nothing_up`, `nothing_down`, `nothing_right` and `nothing_left`. As well as the collisions between said animated objects, which have consequences in the game like when Mario defeats or is defeated by an enemy, or when he takes a mushroom.

3. Most relevant algorithms

3.1 Update

The Update Method is the "Super Mario Bros" game itself. It controls the game, running every frame.

A high-level description of the algorithm is:

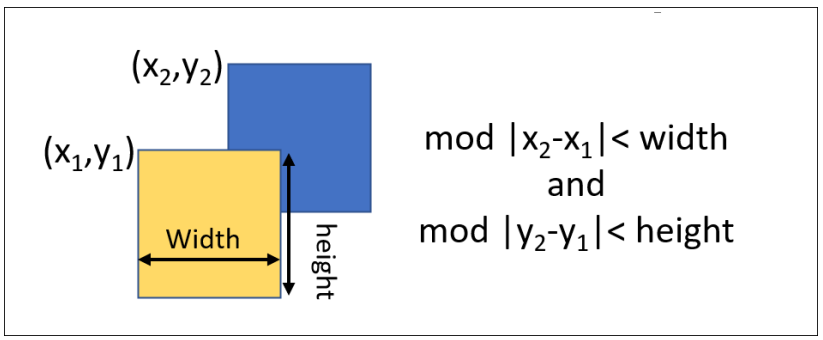
1. It controls the user's interaction with the game by scanning the keyboard every frame.
2. It controls Mario's position to update either Mario's position or the other element's position, to avoid Mario passing to the right side of the screen. Only when Mario reaches the castle, does he move to the right to finish the game.
3. It checks if Mario can move right/left/up/down, if so, Mario moves.
4. In case Mario cannot move it analyses why, if there is a collision, checks the collision element and acts accordingly.
    a. **BlookFloor / StairBlock / Pipe,** Mario can not move in that direction
    b. **BreakableBrick**, the block breaks if it is hit by SuperMario.
    c. **BrickWithCoins**, creates a coin if there are coins left in the block.
    d. **QuestionBlock,** it creates either a red mushroom or coins.
    e. **InvisibleBlock**, it creates the green mushroom
    f. **Castle** (that appears at the end of the game, and once Mario has reached it, the user will have won the game)
5. It controls the jumps.
6. It controls enemy creation and movements.
7. It calls the `clean_up` method to remove the objects that are not needed any longer. For example, enemies that have fallen into a hole.

3.2 Collisions

We based this algorithm on the following principle:

There are two objects in the game space, where the position, coordinates X and Y and the size (height and width) of each object is known.

There is a collision when one object invades the space of the other, and this can be detected when (**mod $|x_2-x_1|<$ width** and **mod $|y_2-y_1|<$ height**) as we can see in the following figure.



In our game development every object has an attribute that indicates the height and width, those parameters are used to detect the collision.

As an example of use of this algorithm. During the game before each movement of an animated object we determine if there is free space in the direction of the movement by calling the methods; “nothing_rigth”, nothing_left”, nothing_up”, nothing_down”, where we use the next expected position of the animated object to determine if the movement can be done.

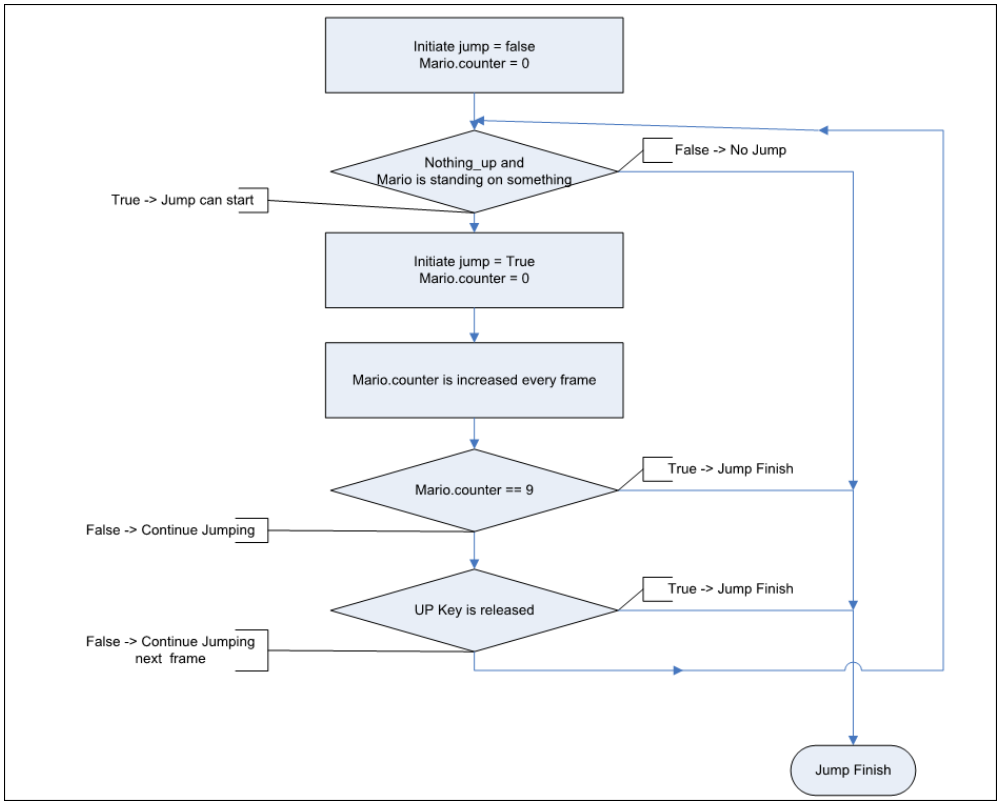
3.3 Jump

Making Mario’s jump has been something in which we have invested some effort to accomplish a “natural jump”.

High level requisites where:

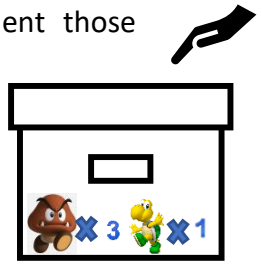
1. Jump has to be done in several frames in order to give continuity to the player’s perception.
2. Jump can only start when Mario is on top of something (floor, pipes, blocks…).
3. Jump has to be limited; a new jump cannot start when Mario is already jumping.
4. Jump must finish when hitting blocks on top of Mario.
5. Mario rebound’s when he has jumped on top of an enemy, this is a jump that is not started by the player.

In order to accomplish those requirements there is a variable that controls the jump along several frames in game “initiate_jump”, and a counter to limit the jump which is an attribute of Mario (mario.counter). Additionally, when Mario is jumping gravity function does not apply.



3.4 Enemies

The enemies' creation was one of the main duties to work in Sprint 3. We needed to create them at a specific position and height so they were not always on the floor and could come from upper blocks. What is more, they needed to be produced in a random space of time. With the use of the *pyxel.frame_count()* and the *random* library, we easily implement those functions.



Furthermore, using the "random" library we were able to create those enemies randomly with the probabilities asked (75% Goomba, 25% Koopa Troopa).

Finally, we added an element of control that manage the production of enemies meant for not having two enemies produced simultaneously.

4. Work performed

According to the final project requirements our Mario-Bros game has accomplish the following sprints.

Sprint 1: Objects and Graphical Interface

We have created a class for each element type, see chapter two for class definition details. There are super-classes to group the elements that have the same behaviour in the game, for example blocks.

Map class creates the Level, the size of the level is 10 screens with around 50 elements (pipes stairs and different block types). Element's creation is flexible, there are lists that contain the positions to create dynamically the level.

Score, Time countdown counter and coins counter has been developed

Sprint 2: Basic movement of Mario

Mario can move right, left, jump, and fall until landing on top of another object or falling into floor holes.

Sprint 3: Enemies

Mario interacts with enemies according to the game specifications, see chapter three for details regarding enemy creation.

Sprint 4: Game

Mario interacts with all the blocks according to Mario's State (normal Mario or Supermario), coins appear on top of the hit block that contains them, the blocks that contain coins and mushrooms change their appearance when they are empty. The game is restarted when Mario is defeated, losing a life. After three lives, the game is over.

Sprint 5: Extra elements

1. An invisible block has been introduced with a green mushroom, that adds a life.
2. Start, restart and end screen.
3. Sound effects that can be switched ON and OFF during the game, with an icon that appears on the top-left corner to show the status, following is the list of sounds created.
    - game start
    - game end
    - jumping
    - collisions with blocks
    - win coins
    - kill enemy
    - discover and capture of mushrooms
4. Koopa Troopa shell, When Mario jumps on it, Koopa Troopa hides in its shell, if Mario jumps on it again it is used as a projectile.
5. When Mario defeats an enemy the score points appear.
6. If Super Mario is defeated and turns into normal Mario he flickers, in which state he cannot be killed again during that brief time.
7. We have added some utilities to interact with the program at game level.
    - ***Key-R*** -> Restart the game
    - ***Key-Q*** -> Forces the exit of the game.
    - ***Key-M*** -> Activates and deactivates the sound. This can be seen by an Icon that appears on the top left part of the screen.
    - ***Key T***-> Sets the game to test mode, which allows the user to continue the game even though Mario has been killed by enemies. This can be seen on the screen with the text *TEST MODE* in red. Once the test mode is activated it cannot be deactivated.

5. Conclusions and Personal comments

Creating the Mario-Bros game from scratch has been a challenging project, we found it very pleasant to see it working at the end. At the beginning of this, we could not even imagine that we were able to create something that works as it does. We are proud of our work when we see our Mario defeating enemies and reaching the castle.

It has been a bumpy ride with plenty of obstacles along the way, like managing all the collisions properly, with the background movement, adding the music, a proper class design, which has been modified several times before our final version… and many other obstacles we have had to face up to, but all in all it has been quite a learning experience which hopefully has helped us become better programmers.

Even though we have introduced many extra features in our game, there are others that we have thought about and would have liked to include, but due to time constraints we were not able to develop them, those are kept in the backlog for the future.

In terms of saving time, something that could have helped us, would have been to have the graphical resources ready in a "*.pyxres*" file, even though it has been interesting to know how to develop our own graphical and sound resources, it has been a time consuming activity that was not the aim of the assignment.

6. References

- **CaffeinatedTech**. (2020, November 27). *Python Retro Game Tutorial*. Retrieved from https://www.youtube.com/watch?v=Qg16VhEo2Qs

- ***Github***. (2021, December 12). Retrieved from https://github.com/kitao/pyxel

- ***GitHub User Examples***. (2021, December 13). Retrieved from https://github.com/kitao/pyxel/wiki/User-Examples

- **NIntendo**. (1993, July 14). *Archive.org*. *Mario Sounds Effects*. Retrieved from https://archive.org/details/mario_nes_snes_sounds/Mario+1+-+Die.wav