Imagen que contiene Interfaz de usuario gráfica

Descripción generada automáticamente

Group 97

**Sergio Rodríguez & Alejandro Sánchez**

Programming

FINAL PROJECT:

SUPER MARIO BROS

ÍNDEX

1.Classes design, methods and algorithms…………(2-6)

- Hierarchy and Introduction…………(2)

- Main (Class and simple functions) …………(2-3)

- Board (Class, Methods, Algortihms) …………(3)

- Blocks (Class, Methods, Algorithms) …………(3-5)

- Accountant (Class and methods) …………(5)

- Enemies (Class and methods) …………(5-6)

- Sprites (Class and methods) …………(6)

- Mario (Class, Method) …………(6)

- Back (Class and methods) …………(6)

2.Performed Work…………(7-8)

3.Conclusion…………(8)

4.Personal Comments…………(9-10)

ABSTRACT

- This project tries to emulate a functional simplified version of the first level of the original Super Mario Bros game for the NES. We have added some not original functions and through the next pages we will explain the different classes and their hierarchy, the methods inside with the correspondent algorithms, what we tried and worked, what we tried and don´t, a general conclusion of the development of it and a final review with our thoughts on the project.

**1.CLASSES DESIGN, METHODS AND ALGORTHIMS**

Diagrama

Descripción generada automáticamente- We have created 8 different classes, considering the main one which runs the game, two mother classes which store most of the raw code and five complementary classes which provide support and different functionalities to the previous two.

**Figure 1: Classes mind map**

**MAIN**

**- Class and simple functions**

This is the principal class as it starts the game. It initializes the class **Board** by giving it the screen values (width and length), loading the pyrex file from where all the sprites are coming and starting two of the most important methods: draw and update.

**BOARD**

**- Class**

This class, accompanied by **Blocks**, constitute the brute code of our game. In this case, it serves as central node, evoking the rest of the classes, and computing their methods, kind of shortening them into no print-methods (update method) and print-need methods (draw method). Because of that **Main** just need to directly work whit it.

**- Methods**

The two most important methods are in this class which are:

**Update method**: a self-method in charge of the functions which needs to be active all the playthrough, it refreshes them every frame (1 sec = 30 frames). It oversees some basic functionalities as overall all the movement, the collisions, gravity, the key input translation, and the enemy’s randomizer.

**Draw method:** a self-method which groups up all the graphical interface stuff, it prints the interactive items on the **Blocks** list, the background and the timer, scorer, coins, and live counter, also keeping the last one’s algorithms since they change what it is on screen.

The **\_\_init\_\_ method** the class already have also starts some other features as the timer, the Mario position and its range and the background coordinates so the update method can move it.

**- Algorithms**

This class does not have some complex abstract code but some honorable mentions can be: the enemies randomizer which every frame generates a **random.randint** and with and if-loop checks the probability, choosing the enemy type to generate with another randomizer and loop; and the mathematical computation for time which if reach 0 teleport the player into the game over screen.

**BLOCKS**

- This class oversees all the interactions between the different elements of the game. As the Board class is more of a classifier and runner, this keeps the larger quantity of code. It has a first part with the lists of interactive items, which will go into the class **Board** **Draw method** and a second one full of movement and collisions code for the **Update method.**

**- Methods and algorithms**

This class methods are the brain of the game:

**Right Left Up and Down collisions method:** the responsible of the all the relations between Mario and all the items, the fourth of them follow a principal algorithm based on the coordinates of the items compared to Mario’s.

An if statement evaluates the corners of the sprites (the collision spot) of Mario and the item, and a third spot in Mario, which can be the opposite corner in the parallel line to the spots, but also a closer one for more precision. For example, right collision:

M

The yellow one’s are the corners that collide and the blue one is the one that gives the accuracy.

So, if the item’s spot is between those points, and keeps true another statement which compare the other coordinate in a similar way but understanding that the opposite coordinate can enter in both ways, for example, you collide right, but it does not compare the same y coordinates for entering right up than right down (so you have an or statement).

M

M

Then if the middle point in each case is between the range of the others, the statements became true (you can minimize it, so you get the maximum accuracy), it subtracts the length of that coordinate in actual Mario´s sprite, acting like an invisible wall.

Returning to the methods, you only must adjust the coordinates to the 4 different types of collisions make sure the don´t overcome each other, and design the specifies interactions with each one, for example up and down had to change the Mario state of falling or jumping. Talking about the items, the pipes sprites don´t follow the default size of 16x16 so we had to make and if statement on down, left, right collisions with the **element.type** on the list, invoking the **Sprites class**, the same happened on those ones with the **big.mario** boolean from the **Mario class**. Moreover, we put in them a different outcome for the **element.type** they were colliding with, for enemies, right and left will subtract Mario one life or **not big.mario** (on the other hand, hitting a mushroom goes to **big.mario** if **not big.mario**) while down, .remove them from the list, similar to the up with normal blocks or the **.type** transformation with the question ones concatenating a randomized spawn above them of a mushroom or a coin.

**Gravity method:** a simple but necessary one, depending on Mario’s size, does again a coordinate comparation with **mario.y** related to the block one, and a **mario.x** or statement, remembering if Mario is not completely outside it shouldn’t fall. Depending on that change the Mario ‘s over and falling booleans parameters.

**Enemy movement method:** minimalist, set left as the facing direction for all items in enemies list, and set a parameter that put it in **Update method** change the x advance of the element.

**ACCOUNTANT**

**- Class and methods**

A necessary class destinated at first to regulate all the graphic interface values as time, coins, score, and lives, but ending up just setting as an auxiliar class for time, where an **\_\_init\_\_ method** set the start time of the game by invoking the time default class. It also stores the sprites used in the interface and their base coordinates.

**ENEMIES**

**- Class and methods**

Like the next **Sprites class**, **Enemies Class** is a storage of sprites, in this case for the enemies, why this separation? It was orientated to the fact of the movement and collisions of its items. It has an \_\_init\_\_ method whit essential qualities as the x and y coordinates, also a small randomizer for the bullet range of appear from the floor to the sky, saving it on a variable because if not each frame the y coordinate of the bullet would change. Then a **show method** stores all the data for each element sorted by their names, coordinates, bank, coordinates in the bank, size.

**SPRITES**

**- Class and methods**

As said before, a storage for the information of each non-enemy element the same characteristic as the last one.

**MARIO**

**- Class**

A class which storages all the information related to Mario. In a first sight, an **\_\_init\_\_ method** storages a lot of variables as coordinates, states of movement, size, lives, and Mario’s jumping gap and ground. You just need to insert the initial location and direction he is facing and that´s all.

**- Method**

The **move method** is basic but compulsory, have four sections for each direction, and changes the states of movement (falling, over, jumping), the size if **big.mario** condition is modified and the detail of turn over the sprite by adding a minus on the **x.bank** coordinate depending on the direction.

**BACK**

**- Class and methods**

Finally, the background class, responsible of the level following the Mario’s path. It’s compound by an **\_\_init\_\_ method** with coordinates and a parameter used on a **move method** for invoking the hole level from the pixel bank where it is designed.

**2.PERFORMED WORK**

At first, we follow a straight path from the pdf points as we didn´t had the best class hierarchy design. We start the basic functions, knowing the pyxeleditor environment, designing sprites…

The Main, Mario and Board classes were already started, but then we began modifying them with little aspects as position, the different counters where the Accountant class was born, which at first only could set the time but we predict the future and did the completely graphical interface.

Then we got to the first big struggle, collisions, the theory wasn’t difficult to understand even though we didn´t saw it at first sight but implement it in a class required knowing more about the code we already have and class behavior. Simultaneously, the Sprites appeared as we need of it for trying the collisions because at a first attempt, we also draw all the interactive blocks in the background.

Once we got it, the interaction with the blocks begins, a little bit frustrating but we got it when we realized it was a list task. Separating big steps into little ones helped a lot.

The time was running, and we keep trying somethings that we were almost done, half-worked or in most of the cases, we knew the theory but putting into work was more difficult. We let the code of those attempts in our code as:

- Applying the properties, we were more focused on the performance of the game, and telling ourselves once it´s perfect we applied them, but time caught us.

- The coin and score counter, we already created classes with \_\_init\_\_ methods, initial values, even set the subtracts and sums with the collisions and the graphical interface, but it kept showing the 0 value in the draw function.

- The koopa tropa, we set the koopa movement in a fixed area with an original x, and if statements it surpasses by a defined value that point in any direction turn into the other. Also made the shell sprite and collision so when hit it turns into a shell, and the shell just goes away in a direction based on the right or left collisions until it goes out of screen where we deleted from the list or collides with something and changes directions.

A similar code used the goomba, but when we randomized their spawn, as we couldn´t find on time a way to count the screen enemy’s element, the shell was on other storages so did it count or not, etc. Enemy’s collisions with the environment were almost done.

- Big Mario broke the entire collisions original system, with big efforts we managed to fix it on time but it isn´t as accurate as we would have wished.

Besides all of that, we managed to make a fully range of movement and size Mario, with general collisions and interactions with every item and a few extras like the bullet enemy, the lives counter and reset, a game over screen, an end with a win screen, randomized question blocks items generator, randomized enemy generator with different types, stairs interaction, gravity, time counter with a game over at the end, full ground with collisions so if there is a gape, Mario would fall and the halfway done ones.

**3.Conclusion**

- During this work we surely learnt a lot of methods, class and OO programming in general, although we have done so by working very hard and trying so many times.

The biggest help came from talking between companions about the development of the code, because most of the time we knew what we wanted to do, how to do it, but the code wasn´t working and Internet webs like Github didn´t came handy because pyxel seems to not be a frequent tool used to design games, pygame was just everywhere.

Unfortunately, the deadline comes when we are at our best level, because we have improved significantly during this time and now upgrading the game it its finest is more a matter of time than of knowledge or skill, at least for the objectives we had on mind.

Summing up once we dealt with collisions and the list of objects the rest was more of putting the final touches to our ideas. In general, a positive outcome for this project for being the first big touch with programming.

**4.Personal comments**

- The general first impression when it was announced was a very big gap of difficult between the class exercises level and the final project, even between people. It has required a lot of time trying, talking, and thinking. In OO programming you need to practice for getting skill, the theory is easy, but you need to put that on work more that in other sub-areas seen this course.

We feel we didn´t feel prepared also to apply properties at this level, at least as a first impression.

Maybe some guided weekly progress at first, especially for the hierarchy and pixel command would leas into a better performance and higher quality results.