FINAL PROJECT

Super Mario Bros

Pablo Simón Martín & Álvaro Martín
Group 196

Index

1.Classes

To “build” our game we have created a class for every “object” that is part of the game. Besides, some of them act as mother classes for other objects, and we have also created a super class called “locatable”, since all the objects have two common attributes, the coordinates x and y.

Main:

This is the class from where we run the game. Here we create the board of the game (with size 255x255 pixels), load the image bank, and with pyxel.run() we are continuously calling the methods *update* and *draw* from the board class (until the game is quit).

Board:

This is the most important class for the behaviour of the game because it is where we store the update and draw methods that are necessary to run a pyxel program. In addition, inside the board class we also create all the different objects that are shown in the game. Most of these objects are stored in lists so that we can control its behaviour automatically.

Mario:

Among all of the different objects created in the board class, the most important one is mario, because everything revolves around him. Its information is stored inside the Mario class. The Mario class has as parameters the coordinates x and y, along with the lifes.
The Mario class has several attributes of boolean type that tell us information about Mario’s collisions or about what he is doing on the board (is_jumpling or is_blinking). Apart from that, there also several methods that are necessary to control Mario’s movement (jump, falling, move) and another one, blinking, that is used for the transition from Super Mario to regular Mario

Block:

Most of the objects of our game could be qualified as blocks, so we created a mother class named Block in which we stored methods that were common to all these different blocks, mainly related to collisions with Mario and the enemies. Among all the blocks, there are two different subgroups: regular blocks and boosts.
For boosts, we added an extra mother class because they had several characteristics different to regular blocks.

Enemies:

This was also a very necessary mother class, because all enemies have several common characteristics: they are either alive or dead, they die when Mario jumps over them and they kill Mario if they collide in any other way. Apart from that, all enemies have a direction when moving and have to fall when they are not over something

2. Methods and attributes

2.1. Methods

move

Different variations of this method were used in three different classes: Mario, Goomba and Koopa_troopa. The essence of the method is to change the x coordinate of the object every time the method is called, but there are slight differences for every type of object. However, all of the methods depended on the direction of the object, adding or subtracting the x coordinates depending on the direction. The methods have different ways to be implemented. For example, Mario moves when the right arrow is pressed, while the enemies methods are implemented automatically in every iteration of the update.

jump / mini_jump

As with move, there are several times we implemented these methods. In general, the way our jump method works is the following. When an event that ignites a jump occurs, we store in an attribute of the object that is about to jump the frame when that occurred and we change an attribute of type boolean (usually called is_jumping or similar) that is False by default to True. Then, inside the update method of the class Board, we use a conditional to check if that attribute is True and call the jump method when it is, using the frame count at that moment as a parameter. Inside the method, we change the y coordinate of the object while the frame count is smaller than a bound we set using the initial frame count and then reduce the y coordinate in another interval. Finally, when the frame count exceeds this value we set the boolean attribute back to False.

falling

Falling is a very simple method that enemies and Mario have. It is called when they are not over something (blocked_below) and it consists on decreasing the y coordinate of the corresponding object until it is back over something or it is no longer on the screen. In the case of Mario, falling is used for the second part of the jump.

Is_blocking_below / is_blocking_above / is_blocking_front / is_blocking_back

These methods belong to the block class. They have as a parameter an object of the type Mario, Goomba or Koopa_troopa and return a boolean. They return either True or False depending on a conditional that compares the coordinates of the block and the object entered as a parameter

2.2. Attributes

coordinates(x and y)

This is maybe the most important attribute of all, mainly because every single object needed to have them. That's why, as we mentioned before, we created a mother class called Locatable that is used in every class(except from board).

sprite

The sprites stored the information of the aspect of the objects. Every class, except from the mother ones and the board, had sprite as an attribute. We decided to create the sprites as read only attributes so that they could be read but not changed outside the class. An advantage of doing this was that we could control and change the sprite very easily depending on other attributes of the class, without having to do anything on the draw or update method. We used this to change the aspect of Mario and the koopa troopas when the direction changed, the aspect of Mario when he was Super, the aspect of the question blocks when they were hit and the aspect of the breakable blocks when they were doing their mini_jump to display a coin over them

visible

This is an attribute of type boolean that we used in Boost and in enemies, with the same objective (to control when the objects were displayed on the screen), but for different reasons. In the case of enemies, the aim of the attribute was to wait for the boost to spawn. All the enemies' objects were created in the init function of the board, but we didn't want them to do anything until Mario was close enough to them. This means that when Mario was close to those elements, the attribute visible became True and the enemies started doing things. With boost, the visible attribute had a different reason. We created the boost so that there was one boost assigned to every question block displayed on screen(they are assigned by having the same index). The attribute visible of a boost object would change when the question block assigned to that boost was hit from below

blocked_below / blocked_above / blocked_front / blocked_back

The only class that has every single one of these attributes was Mario, while enemies only have blocked_below. They are attributes of type boolean that are strictly related to the collisions of Mario and the enemies. Using the methods explained previously, we checked if a collision of Mario or an enemy with a block happened and when it happened, we changed the corresponding boolean to store that information. Finally, inside the update method, we used the information given by the boolean to do different things. For example, if a certain Goomba object had blocked_below as False, the method fall would be called

hit

This is an attribute of type boolean that every block has. As with the attribute visible, the aim of the attribute was similar (to know when Mario hits a block) but the reasons and the ways it was used was different depending on the type of the object. For example, with boosts, it was used to know whether a boost had been obtained or not and with breakable blocks to initiate the mini_jump they do.

3. Algorithms

The main algorithm that we created in our project was to check the collisions. We created that algorithm as a method inside the Board class. Inside it, we checked every single block in our board using for and while loops to see where Mario and the enemies were interacting at every frame. One of the most important things of the algorithm was to assume that Mario and the enemies were not interacting with anything at the beginning of every check/frame. We did this by setting the blocked attributes of each object to False at the first part of the method. Apart from checking the collisions of Mario and the enemies, our algorithm was also useful to know which breakable or question blocks were hit, so that we could increase the score or unlock the corresponding boosts. Moreover, we also used this algorithm to check when the boosts were obtained.

Another algorithm we implemented was to control the spawning of enemies and its interaction with Mario. In this algorithm, we check every enemy created previously in the init method to see if it is in range. When this happens and there are less than four enemies on the screen, we turn the attribute visible of that enemy to True. In any other case, we set visible as False. This way, when enemies leave the screen through the left side, they can not become visible any more. Once we have finished working with the attribute visible, we use two methods created in the mother class Enemy on enemies with visible as True. These methods , which have Mario as a parameter, are called mario_kills and kill_mario and they both return a boolean. They are very similar to the collision methods in the Block class, but adapted to work in another context

4. Our work

Even though we had to go through some rough patches, we always tried to keep moving forward, slowly and steadily.
Once we knew our way in pyxel, we started implementing the different sprints of the project. Firstly, we created all the different classes with their main attributes. Obviously, we didn't create all the attributes necessary in the first try.
Having the classes created, that worked as the skeleton of our project, we started working with pyxel itself, placing Mario, the floor and other objects in the board. At first, they were only drawings and Mario could not interact with them. We also added the score and the time, but just as fixed values. This first sprint was easy for us.

In the second sprint, we had to implement Mario's movement, which we had done previously in a reduced class. We also had to force Mario to stay stuck in the middle of the screen and make other objects "move". We were able to accomplish this rather easily. Changing Mario's sprite along with the directions was a bit more challenging, but we sorted it easily. However, we got stuck with making Mario jump, because, at first, we didn't understand the information we were obtaining with pyxel.frame_count. After that, we moved on with the collisions. They were probably the most time consuming part of the project and we were not able to implement them perfectly (as in the real game) in the end.

When we got to the third sprint, we were already working at a steady speed. Implementing the movement of the enemies was very simple, because it consisted of automating the same movement that Mario did. Spawning the enemies correctly was a little bit harder. We decided to create the enemies in the init method in a fixed area and do things with them depending on the pyxel frame counter because of what we read on the script. However, with the way we had thought to implement the enemies, it made more sense to control the spawning of the enemies with the so mentioned visible attribute.

Finally, we started the fourth sprint with the coins, which looked like the easiest thing, which indeed was. After that, we moved on to try and implement the time correctly. It seemed easier than it was at the end, but we were able to handle it correctly. Then we implemented the hitting of blocks and the appearance of boosts over question blocks. It was a bit time consuming because we had to take into account several things that we didn't think of at first, but we sorted it out. After that, we implemented Super Mario. Apart from changing the sprite, which was the easiest and hardest part at the same time (we had to draw the sprite), we had to change the boosts and to implement the blinking of Mario. Implementing the blinking was similar to the jumping, but instead of changing the y coordinate of Mario we kept changing its sprite until the blinking was over. With the boosts, we decided that while Mario is Super, every mushroom turns into a fire flower and that if mario is not super, every fire flower turns into a mushroom.

5.Conclusions

At the beginning it seemed impossible to carry out a project like this and we were a bit overwhelmed the first few days. We have to deal with the OOP methodology and also with Pyxel, two things that were new for us, but later (algouth it sounds like a cliché) we see that going step by step we could make it work.

Within this process we have learnt that if we try to take on many tasks at the same time, everything will fail. When programming, it is much better to organize the work in little tasks, starting with the easiests and the ones that you feel more comfortable with, and then (after checking that everything 'fits') move on to the next task. If we start thinking about how difficult it will be to create the enemies, even before Mario moves, everything will look a lot harder, but if on the contrary we start making Mario move to the right, then to the left, then jump… and continue following a logical path, when we will reach to the part of the enemies everything will make much more sense and we will see things in a different way. This project has definitely changed the way we approach problems in programming.

Another positive aspect we learnt was not getting stuck on something, and learning how to move on. Sometimes we found ourselves crashing into the same problem again and again, and there are times that the best you can do is to move on, and later come back to the problem and see the problem from a different angle.

Finally, the most important thing that we have learnt is that it is always better to have less code but of better quality. With this what I mean is that especially at the beginning, as we wanted to move quickly on to the next task, we did the code, ran it, and if it worked, we continued. We then realize that this was a horrible decision, and that the fact a code works doesn't imply that is good. It may work, but to be good it needs to be done in an efficient and clear way, with comments and making it easy for your partner to understand it, otherwise you will spend a lot of time trying to understand what you did in the past.