PROGRAMMING FINAL PROJECT

Universidad Carlos III de Madrid
Degree in Data Science and Engineering
Subject: Programming

Denis Iliyanov Antonov

Alejandro Leonardo García Navarro

Contents

INTRODUCTION

For the subject of Programming, we had to put into practice all that was learned so far during the whole course. To do so, a final project had to be performed. We were required to implement a simple version of the first level of the original game Super Mario Bros for the NES. It is vital to notice that blocks, special objects, and enemies must be created.

Hence, several steps must be done:

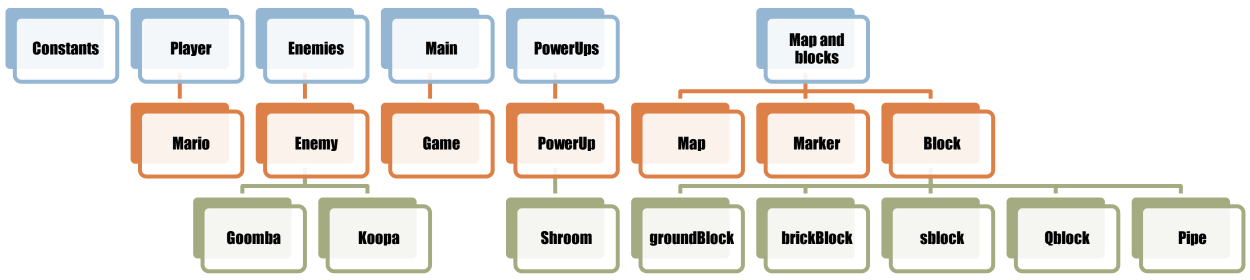
1. Sprites and graphical interface. In this very first step, sprites must be created. A sprite is a two-dimensional image that is part of an even larger graphic scene, and it is typically a certain type of object that interacts.
2. Good class design. In this final project, we would use a module to put all the variables and functions related to Mario, another for enemies… each module is called a Class. Once the class hierarchy is defined, the game will be developed.
3. Create a class for every element in the game, with the appropriate attributes. These are the variables characterizing each object. Each class may be different depending on the attributes that we want to add to them. This will be specified later in the report.
4. Basic movement of Mario. For this part, it is important to notice that Mario needs to be able to move forward and backward, as well as going up and down. If Mario advances to the right, the map will do so too, but he cannot either go back or move past the center of the screen, so those are important cases to contemplate.
5. Enemies. Note that these will behave almost like Mario. On the one hand, they are going to move by themselves and if they collide with an object, they will change their direction. On the other side, we need to spawn them considering the chances of them appearing in the game. May these chances be: 75% of it being a Goomba and 25% of being a Koopa Troopa.

Other requirements must be fulfilled, like getting a special object or coin when hitting a block and, hence, changing the appearance of the block; when coins are obtained or Mario defeats enemies, the coin counter increases; Mario transforms when touching special objects; consider time limit and having no more lives…

After following the steps indicated on the project requisites, a final functional version has been reached.

CLASSES' DESIGN

In our case, we used 15 classes to perform this project. To create them, several object orientation techniques were carried out. Moreover, the most used one was inheritance. In the figure from below, we can have a clear vision of the organization of our codes:



(Figure 1)

May we observe in *Figure 1* that the blue color represents the modules where the classes are and, in orange and green, the classes. The ones in orange refer to the mother classes, while the ones in green refer to the ones that were created using inheritance.

Game

This class will be the most important one of the codes. This is basically where the game resides, as well as the part where the map is created. Inside it we can find a method where several things are done:

1. Pyxel is initialized.
2. The assets (drawings) are charged.
3. The initial position of the player is set.
4. Details like the coins' counter, the time and number of lives are specified.
5. The blocks are created.
6. The arguments of 'run' function and 'update' function to update each frame screen when necessary are established.

We can even find the update method, whose functionality was specified before. In here, we included all the things that must change during the whole game. Among these, we can find some like:

1. Lives, score, time.
2. Every possible interaction of the player with the enemies, blocks, and special object (and vice versa).
3. Movement of Mario, enemies, and special objects.
4. Positioning of objects.
5. Quitting the game.

The draw method is the last that can be found in this class. Its functionality resides on its name, it draws all the sprites.

Player

Contains the code related to the sprite controlled by the user. To do so, we created 7 methods with different objectives. We can find some along the lines of:

1. def __init__(self, x, y): This magic method initializes the attributes related to dimensions, appearance, floor, lives, score, and physics.
2. def draw(self): A method that draws the player depending on the key we press.
3. def moveMario(self, map: MapAndBlocks.Map): this method controls the movement of the map when Mario moves.
4. def marioCollisionBlock(self, blocks: list, publocks: list, shroom: PowerUps.Shroom, marker, level): in charge of controlling the collision of Mario with blocks.
5. def marioCollisionEnemy(self, enemies: list, level: MapAndBlocks.Map): its functionality resides in managing the interaction of the player with the enemies.
6. def marioCollisionPU(self, shroom): a method that controls the interaction of the player when it touches a special object. Note that whenever this happens, we must consider that Mario needs to change in the case that, for example, touches a mushroom.
7. def changeSprite(self): this is not a vital function of the game, but it adds the feeling that the player really moves. In here, whenever Mario jumps its sprite changes, as well as when it walks.

Enemy

This constitutes the mother class of Goomba and Koopa, the two types of enemies that will appear during the entire game. It contains the common characteristics of all of them, such as the self-movement. Contrary to Mario, these are not controlled by the player, and they need to move by themselves.

Goomba, Koopa

The behavior of these enemies is created in two different classes. They constitute the child classes of Enemy and, inside each of them, we used an extra method that draws them.

Map

This class contains the information related to the position of the image bank where the map is.

Marker

This class is mainly used in the Game class described above. It is used to create the texts that shows the number of lives, the points, the level of the game and the time.

Block

It represents the mother class of groundBlock (blocks that constitute the ground), brickBlock (specific type of block for decoration), sblock(specific type of block for decoration), Pipe and Qblock(referring to questions blocks that contain the special objects), the 4 types of blocks and the pipes that will appear in the game. It contains the common characteristics of all of them. We considered the fact that, as Mario advances, the map and the blocks advance too, so a function was used to tell the program that when this happens, the position of the block changes.

groundBlock, brickBlock, sblock, Qblock, Pipe

The behavior of these blocks is developed in several classes. They constitute the child classes of Block and, inside each of them, we used an extra method that draws them.

PowerUp

This constitutes the mother class of Shroom, the special effect that changes the aspect of Mario when this is touched. It contains vital characteristics like the positions, dimensions, and velocity.

Shroom

This is the child class of PowerUp. It contains some extra methods necessary for its behavior and creation. Apart from the one that draws it, there is another that describes its movement.

Constants

Even though this is not a class, it is a worth-mentioning part of the project. It contains predefined variables. Some important constants like the screen width or height are declared here. This is useful if we want to change a game constant because we only must modify it in one place.

GENERAL DESCRIPTION OF THE MOST RELEVANT FIELDS AND METHODS

It was during the whole process of developing this project that we noticed the most important methods of it. Obviously, every single part of the code is important because if we were to take out a single thing of it, several parts will stop working and won't make sense. Yet there are some methods that are the pilar of the game. There are some along the lines of:

marioMove: this method controls the movement of Mario. It is controlled by the user and depending on the buttons we press; it behaves in one way or another. In this part resides the physics, the gravity and how Mario should behave in specific conditions and locations of the map

marioCollisionBlock: this is another vital method, as it controls the way in which Mario behaves when interacting with blocks and pipes. It also contains the sound that needs to be played when the player breaks a block. If Mario breaks a brick, he gets points.

marioCollisionEnemy: this one explains the interactions of Mario with the enemies. There is a for loop that controls everything. For all the enemies, the same behavior must be performed depending on the type of Mario. If it is Big Mario and touches an enemy, it becomes normal Mario. Moreover, if it is already normal Mario, a life is taken and goes back to the initial screen, where the game begins. Even a sound is played when an enemy is killed. When this happens, Mario receives points.

marioCollisionPU: this describes what happens when Mario touches a special effect. In our case, we only included the mushroom that transforms normal Mario into Big Mario.

changeSprite: this is not as vital as the methods from above, but it is mainly used to add the sensation of movement to the sprite of Mario. Specific sprites are shown when Mario is running and jumping.

On the one hand, as observed, the most important methods are described in the Player module. On the other hand, there are others that make the game more entertaining. Some examples are:

moveEnemy: this method, found in *Enemies*, describes the whole movement of the enemies. Its weightiness resides in the fact that it belongs to the mother class *Enemy*, which means that it will be used for the different types of enemies found in the game.

movePU: found in the module *PowerUps*, controls the physics of the special objects such as mushrooms.

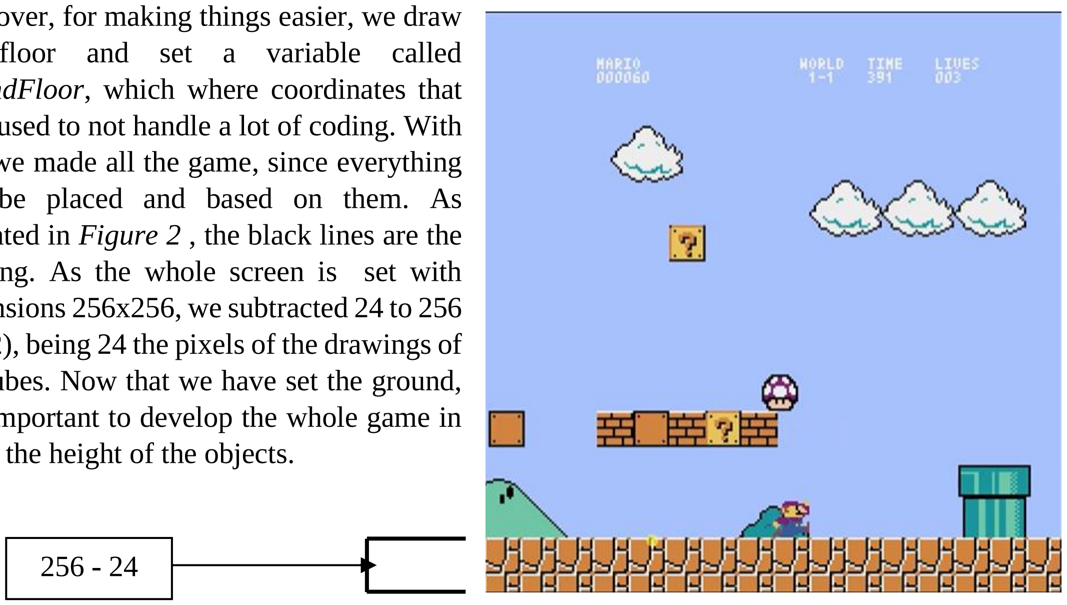
update: finally, this function is the heart of the game, without this, nothing will work. It is used to update each frame screen when necessary.

PERFORMED WORK

To begin with, as specified, some basic sprints must be in the project. From these 4 sprints, we were able to achieve all of them.

According to the first sprint, the one related to the objects and the graphical interface, it was very time-consuming. Once we were clear about the attributes or fields of every element in the game, it was time to spawn all these things and create the game. First, we created the map in the *pyxeleditor*. Note that, in these steps, we could only draw the clouds, the mountains and the bushes. Nothing else could be done, as if we draw a block, it would only be a drawing, not an object. Once the map was designed, it was time to get down to business. With this, we mean that we had to play several times the game to know the coordinates where we had to place the objects.

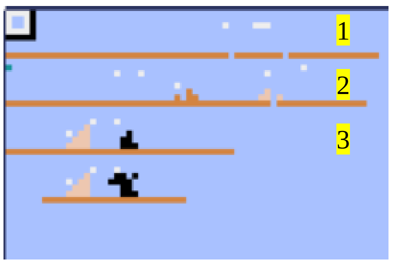
Moreover, for making things easier, we draw the floor and set a variable called *groundFloor*, which where coordinates that were used to not handle a lot of coding. With this, we made all the game, since everything will be placed and based on them. As indicated in *Figure 2* , the black lines are the drawing. As the whole screen is set with dimensions 256x256, we subtracted 24 to 256 (=232), being 24 the pixels of the drawings of the cubes. Now that we have set the ground, it is important to develop the whole game in 232 – the height of the objects.



(Figure 2)

Once that part was performed, we could go on and place Mario, the blocks, the pipes, and the enemies. All of these were established in the *Game* class. Placing Mario was easy, as we just had to specify the coordinates where we wanted him to be. Moreover, the rest wasn't that easy. We initialized the objects and grouped them in lists. Then, in the update part, it is where magic happens. We placed the objects and the enemies depending on the coordinates of the image bank where the player was. In this update part, other behaviors were specified, as what should happen if the player is out of lives, or if we reach the castle or we run out of time.

Moving on to the second sprint, this part was also successfully performed. In this exact sprint, we basically had to create the physics of Mario, as well as some details like the fact that Mario cannot go past the center of the screen and ones the map advances, he cannot go back. We had to take into account that the map is not infinite, so we divided the stage into 3 important rows. We can observe these rows highlighted in *Figure 3*. If Mario reached a certain position, it would go to the next row.



(Figure 3)

Once the second sprint has been explained, it is time for the third sprint. The main objective of this part was to make enemies appear randomly and implement their basic movements. This was also performed in the *Game* class and was done exactly in the way in which we placed the objects. We did it in the update part and depending on the coordinates of the image bank where the player was. We basically placed the enemies in specific positions and depending on their chances of appearing we placed more or less number of enemies.

Finally, in the fourth and last compulsory sprint of the game, we did the rest, which constitutes the basic game. This refers to the behavior of Mario with blocks and special objects*;* the increasement of points when Mario kills an enemy or hits a brick; the transformation of Mario when he touches a special object; and what the program should do when Mario either has no more lives or we ran out of time. All of this was described in the most important methods explained in *page 6.*

Moreover, in our case, we did some extra details. This corresponds to fifth sprint. We thought that this game would not be the same without sound, that it would be dead and not catching enough. Eventually, after having asked permission to our professor, we were told that we could use sounds from the internet to perform this part. Once we had the most important sounds, we searched for a long time the easiest and shortest way in which we could implement the sounds. For this, we used *pygame* and, to be more precise, the function *mixer*. This allows us to play and stop music whenever we wanted, as well as controlling the volume of the sounds.

CONCLUSION

As a conclusion, we consider that, with the knowledge we could have about programming in Python, it was a bit rough to perform this project. Moreover, it is for that exact purpose that we enjoyed it. We feel that this project tested us every time and it helped us improve our skills in Python, get some basic stuff clear in our minds, and reach a level of imagination that we thought we could never reach.

The journey has been full of obstacles, such as the placement of objects or making Mario transform when touching special objects. Besides, we were able to overcome those difficulties. There were some extra struggles, like dealing with code-sharing, as every time we changed something from the codes, we had to overwrite everything.

We would have liked to do a code with a better quality, but we are proud of what we have done. A lot of hard-working was done for developing this project but doing it in pairs really helped, as we could solve doubts between us and propose things that could be improved from the code.

For finishing this project, we would like to say that we found helpful the page: *kitao/pyxel: A retro game engine for Python (github.com)*. In this website we learnt a lot about *pyxel* and how to deal with important parts of the project such as making images look as though they were transparent.