Programming Project report

1942

María Vázquez Goncalves & Sergio Cernuda Cueto

Group 89

Ingeniería Informática

Abstract

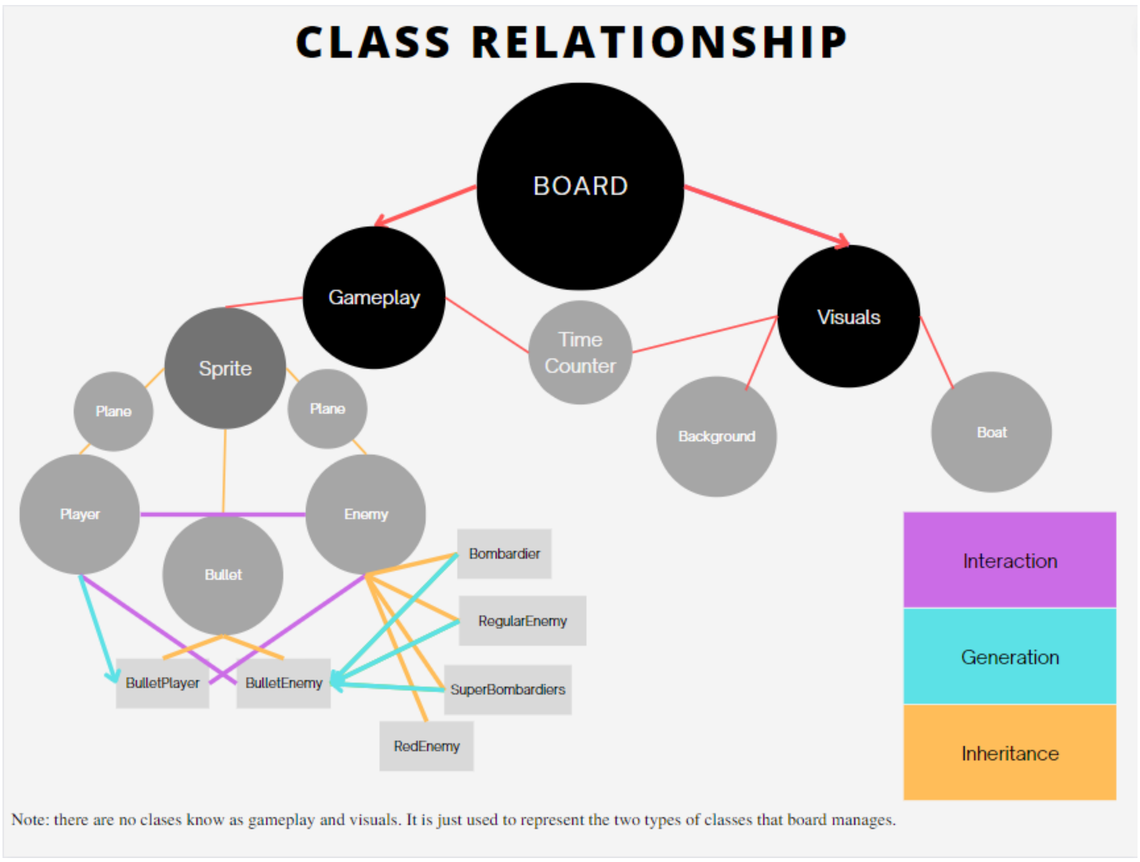
The aim of this project is to recreate the game 1942 by following the steps given in the project guide. Producing a version of the game functional and similar to the original. Code was made considering understandability, expandability and sustainability.

Table of contents

1. Class design
2. Most relevant attributes and methods
3. Most relevant algorithms
4. Work performed
5. Conclusions

1. Class Design

There are a total of 15 classes. The playability is shared between those derived from sprite (using inheritance) and board (the core class). The rest of the classes are used for visual purposes.



Board: the core class that interacts with every other class. Has functions containing enemy generation, collisions, display management, etc. Also is where pyxel update and draw are located as well as pyxel.run().

Visuals: interaction with the user in which the user is shown some visual information.

Background: used to generate the background of the game. It is a moving background made of tiles which movement speed can be altered. Originally it was intended to be a background generator that worked with different tyles.

Boat: generates a boat that moves at a random place.

Gameplay: interaction with the user in which the user can alter what is happening. (It contains some type of visuals)

Sprite: used as the "bones" of every class involved in gameplay. It contains several properties, as well as some methods, like move and draw. Which all sprites will have in common. It is the mother class for Enemy, Player, and Bullet.

Bullet (Mother class of BulletEnemy & BulletPlayer): "self-descripting name" it can be generated by enemies and plays an important role on gameplay experience. BulletEnemy is also able to perform more complex movements.

Plane (Mother class of Enemy and Player): contains two relevant methods which are common to every plane (except RedEnemy which doesn't shoot). Shoot and die. Shoot will consist of returning a bullet type object. Dying will change the state of self.dying as well as the lives that each individual plane has.

Enemy (Mother class of Bombardier, RegularEnemy, SuperBombardiers & RedEnemy): plays a heavy part on gameplay as enemies is what the player will be playing against. They all have a common dying animation. Movement is predefined inside of each child class. SuperBombardier will shoot in a different than the rest of the enemies. Where RedEnemy won't shoot at all.

Player: generates the object that the player will control. It is fully animated using the class TimeCounter. Has its own type of bullets as well as unique methods like loop. The shooting as well as the movement of this object is controlled by the player.

TimeCounter: "self-descripting". Also used for animations when given a tuple containing the images to be displayed. It is also used in enemy generation.

2. Most relevant attributes and methods

Board:

- randomize_enemies: it is used to determine which type of enemy is to be generated. It uses random.randint(0, n) to determine which type of enemy is chosen for generation. Also checks whether there are RedEnemies or SuperBombardiers so it doesn't call the function that generated them.

- add_bullets: adds the number of bullets a plane is shooting to the bullets list.

- clean: removes the elements of a list if they are not being used, in order to save memory space.

- update: invokes methods that involve some type of change in every frame.

- draw: invokes draw related methods in every frame so sprites and animations are shown on screen. An element whose method is invoked after another one is drawn on top of it.

Background:

- initiate: creates a list containing the initial amount of tiles required to fill a bit more than the screen.
- draw: generates new tiles when required as well as displays them.

Sprites:

- move: movement is automatically performed following a pattern determined in move when is a bullet or an enemy. If it is the player, it is gotten from the keyboard input. When called the x and y position of the sprite changes.
- draw: draws the plane and may also call the animate method.

Planes:

- die: deletes them and performs a specific death animation.

- shoot: returns a bullet but may be constricted by the available amounts of bullets. Also, it can return several bullets.

Player:

- loop: performs a loop-like animation and gives immortality to the plane.

3. Most relevant algorithms

While developing the game, we realized we had to find of way of dealing with time and cycles as comfortably as possible. Not only because we had to do animations, which consist on a collection of images that need to be shown at a certain rate, but also because we had to send random enemies at certain intervals. That is why we created the **TimeCounter** class, which uses the concept of frames per second in order to transform frames, something that the library has implemented, into time. We made it so that it only needs the parameters **initial_seconds** and **number_of_seconds**. If it initial_seconds is bigger than 0, the first time the cycle is executed, it ends early. And number_of_seconds is simply the number of seconds you want the cycle or the animation to last. It consists on 2 methods:

- **add_frame**: adds 1 to a variable in each frame and when it reaches the limit of the cycle, it goes back to 0.
- **animate**: given a certain collection of sprites, it draws them sequentially on the desired coordinates. First, it calculates how many frames each image has to be shown. Then, it uses the variable in add_frames to check if each image is on its corresponding interval of frames, and so it draws it.

We also had to create our own collisions for the objects of the game to interact. We created a **collisions** method on the Board that follows this logic. If we think of the definition of collision, we can say that it is the fact of sharing the same x and y in some area. But this is not as simple as saying that they share the same coordinates, because in the pyxel library the coordinates of the sprites are on the top left without taking into account how big the images are. So we had to take that into accont by determining the **x_size** and **y_size** of each sprite. This way, if the left side of an image is touching the right of another image or the upper part of an image is touching the lower part of another image, there is a collision.

For the trajectory of each enemy, we had to separate their movements into x and y components. Regular Enemies and Bombardiers follow a linear trajectory as their attributes **self.x** and **self.y** are modified by simply adding int numbers, but Red Enemies and Super Bombardiers follow sometimes an angular trajectory, so we used the sine and cosine functions. For the enemies to appear on screen, we created an **enemies_list** to stored them when generated with **generate_type_of_enemy** methods, which determined the x and y position in which they would be created

4. Work performed

When the program starts, it shows the user a start screen. We imitated the original game and put the 1942 logo, fake copyright claims and a text that says “press space to play” with a flickering animation. When the user does so, the level is created. The player plane is placed on the bottom middle part of the screen. The background ocean and boat start slowly moving, which create the illusion that the planes are flying high up in the air. The current score, the high score, the number of lives and the number of times the plane can loop are also shown. When the background boat exits the screen, it is relocated to a random position above the screen so it appears again after some time, as if there were actually multiple boats on the level.

Random enemies start to appear after 1 second and then every 2 seconds a new random enemy is created. It is actually not entirely random, because we set it so that Red Enemies and Super Bombardier cannot appear simultaneously as they have overlapping trajectories. They follow their corresponding trajectories and shoot randomly, except the Red Enemy that does not shoot. They are generated until the player loses.

The player can move horizontally and diagonally in every direction with the arrow keys, but with some limitations. It is not able to exit the screen and it also cannot move downwards after a certain y-coordinate. This is because the Super Bombardier appears on the bottom of the screen like in the original game and this gives the user some time to dodge it.

We implemented some animations on the game:

1) The player helix
2) The player loops
3) The player and enemy explosions when killed

The majority of the images of the game came from a sprite sheet of the original game which was available on the internet. Thanks to that resource, it was also able for us to put complex animations.

When the player plane collides with an enemy or is shot by an enemy bullet, it explodes and it loses one live. Also, the game is restarted so a new round begins, like in the original game. The player plane is placed again at the bottom middle part of the screen and the enemies that were being shown disappear. On the other hand, when the enemy is shot by a player bullet, the enemy loses a live. When the enemy runs out of lives, it also explodes. Every enemy and bullet that disappear because they exit the screen or collide with each other no longer remain stored in the program.

When the z key is pressed, the player is able to become invulnerable for a moment by doing a loop. This means, collisions with enemies and bullets are deactivated so that it cannot be killed, and the corresponding animation is shown.

When the s key is pressed, the player shoots. Obviously, this does not happen when the plane is looping or exploding.

When the player runs out of lives, a game over screen is shown. It informs the user of how many points he/she made and that he/she can quit the game with the q key or play again by returning to the start screen pressing space. If the user surpassed the high score, a text whose color changes every frame that says "NEW HIGHSCORE!" is shown. The q key can also be used in the other screens of the game to ensure there is always a way to finish the game.

We also tried making the game so it depended as little as possible on some concrete value for the width and height of the screen and also the FPS, although we couldn't do it perfectly.

5. Conclusions

To sum up, we created a game that was as similar as possible to the arcade game 1942. It was a very good way to apply the concepts of object-oriented programming and to challenge ourselves to develop a bigger program than the ones we were used to in the weekly exercises, but not excessively big. The fact that we were able to do it has been very rewarding, because it shows how much we have learned in this semester, but also how much there is still to learn. We have realized that creating bigger programs with team work require a certain set of skills that we will probably develop throughout our career.

We encountered several difficulties while programming the game. At first, it was really hard to understand how the pyxel library worked, because we had to analyze problems from a totally different point of view as the methods had to be invoked in every frame and we had to deal with a graphical interface, on top of the object-oriented perspective we had just learned in class. The examples of Aula Global were really helpful. We also had to create ourselves a lot of things we did not expect to create as we take them for granted in other softwares, like the time counter class, collisions and attributes for the sizes of the sprites. We had to program complex trajectories for the planes in order to mimic the original game, and we ended up simplifying some of them. It was also difficult for us to the work between 2 people because we had to share lots of files with modifications between us, although we improved as time went by. In addition, we realized we had to dedicate a lot of time to this project and we had to balance it with the numerous demands of the other courses of the semester.

Maybe this project could be done with other python libraries that are also made for doing videogames, like pygame. It would also had been ideal to learn about object-oriented programming earlier in the course.

Despite all of this, we learned a lot and it was a great experience.