Programming Project report

1942

Salvador Ayala & Antonio Vázquez

Group 89

Computer Science (2022)

Abstract: This project tries to emulate the first stage of 1942. After following the steps indicated on the project requisites, a final version has been reached. The code has been designed in order for its future maintainability, understandability and possible implementation of new game functions and features.

Index.

1. Class design

We have created 11 classes to achieve a proper code organisation. Inheritance was used along with abstract classes and other object oriented techniques.

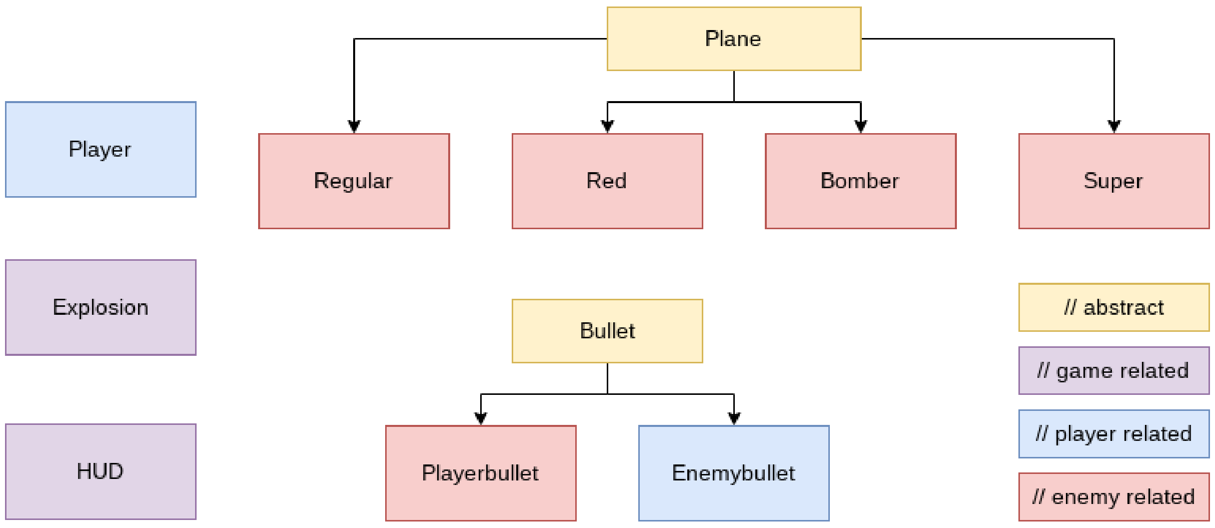

Fig. 1 : Class Design Scheme

As it can be observed from figure 1, we have 4 types of classes:

Abstract classes are the ones that only work as mother classes for the ones below. Objects of these classes are never created, only from the child classes.

Game related classes are classes that only serve for game related topics, such as creating explosions or showing the score.

Player related classes create objects that can collide with enemy related objects. Player bullet objects do not collide with enemy bullet objects.

Enemy related classes create objects that can collide with player related objects. Enemy bullet objects do not collide with player bullet objects.

Player

This class contains all the player related information, such as the sprite coordinates in the sprite file, player position and the player functions such as *move* and *fire*. This class contains some common attributes such as positions and size, but in order to avoid a circular import error, the player class was declared apart. Also, since the player behaviour is determined by the input, it was considered a better practice to create a fully new class.

Explosion

This class contains all the explosion related information such as position and size. Given a position and a size, determines which type of explosion object to create on the specified position. It’s most relevant function is *draw*

HUD

This is the most important class since it contains all the game related information such as *lifes*, *realframecount* or *deathframe*. It will be further explained in point 3.

Plane

This is an abstract mother class that sets all the common parameters for the enemy planes, such as positions, *hp* or *framecount*. Each specific child class has its own *move* function, but they are all different. The general functions are *damage* and *fire*, the first one damages the plane by 1hp when casted and the second one creates an enemybullet object.

Regular

This class contains all the information about regular enemies such as position and their movement functions. The most relevant function is *createregular(x,y)*, which creates a list of x waves of y enemies on each. Waves are divided by 500 (pyxels) from each other.

Red

This class contains all the information about red enemies. The most important function is move, since it defines a circular trajectory using the function sin and cos. It also uses a parameter called framecount that allows to register properly the start of the loop.

Bomber

This class contains all the information about bombardier enemies. The most relevant remark it’s that it has 15hp, higher than regular and red enemies that only have 1.

Super

This class contains all the information about superbombardier enemies. The most relevant remark is that it has even higher hp than the bombardier and that it appears from behind, being its initial position below the screen.

Bullet

This class is a mother class for player bullets and enemy bullets. Its most relevant function is *damaged*, casted when a collision is detected. The bullet is taken outside the screen and deleted by another function called *oos_delet*.

Playerbullet

This class contains all the information about player bullet objects. The most relevant function is *move*, which is different from the enemybullet’s move function. This function just takes the player bullet further when casted, making it travel further.

Enemybullet

This class contains all the information about enemy bullet objects. The most relevant function is move, which calculates the direction vector taking the player’s position.

2. Most relevant fields and methods.

A large number of methods have been created in order to regulate the game’s behaviour. The most relevant ones are those related with collision implementation. They are all stored inside the collision file and are called *oos_delet*, *distance_calculator*, *collision_player* and *collision_playerbullet*.

oos_deleter checks a given list and checks that the elements are being displayed on the screen. If not, it deletes them. This function is called every frame for playerbulletlist and enemybulletlist in order to avoid bullets to keep travelling further and possibly destroy enemies outside of the screen.

distance_calculator is a simple function created for calculating the magnitude of the distance between two objects, distance that will be used to calculate every collision. The function takes their positions, corrects them to the centre of the sprite and then uses those positions to calculate the real distance between them. The magnitude of the distance is radial, so the collisions are spheric around the objects.

collision_player checks the distance between the player and all the enemy objects, such as enemy planes or bullets. If a collision is detected, both objects cast their damage function, resulting in enemy planes and the player receiving 1 damage and for the bullets to go out of the screen for being deleted in the next frame by oos_deleter.

collision_playerbullet checks the distance between each player bullet on the playerbulletlist (remark that those outside of the screen are previously deleted, so they are not checked) and each enemy plane. If a collision is detected, both objects cast their damage function as stated before.

It is important to also mention the *constants* file, which allows the modification of some parameters such as the screen size, the scale and the sprite's position in pyxel's sprite file. This allows later improvements and editions since these parameters are imported for all the other classes and can be changed if needed.

We also have to remark the realframecount property of the HUD class, which allows to calculate the real frame count for each frame, since pyxel's counter cannot be reseted, and it's essential for some game implementations like resetting the level when the player is killed or the game restarts after the game over.

3. Most relevant algorithms and game behaviour.

The most relevant algorithms are the function that pyxel uses to run the game and that we defined as *update* and *draw*. These two functions work in a similar way: they both run one time for each frame and keep the program on a constant loop between them. However, the draw function is the only one that can execute drawing subroutines such as pyxel.blt or pyxel.cls. Nothing outside those functions is executed once the pyxel.run function has started running those two algorithms, therefore, everything that has to be executed should be called from them.

Since they are so similar, we used them in a similar way:

Update

Update function is essentially used for controlling the inputs of the player and updating the objects' positions every frame by casting each object's move method. The function determines when each of the objects start to move, regulating them to appear gradually and, therefore, to create the level design. Also, it controls whenever the game is in game or in the title screen, since objects should not start moving until the player has pressed enter and loaded the game.

This function has two algorithms defined inside of it: the player_death algorithm, which is executed once the player's hp value is lower 1, and the player_roll algorithm.

The player_death algorithm defines the process that has to be followed for the player to disappear properly, creating emphasis on the player's mistake. Therefore, it takes exactly 5 seconds for the player to respawn and the level to be reseted, from which 3 of them are taken for the player's explosion to load. Also, all the level is stopped as the loop goes out of the main object-updating part of the function, highlighting even more the death of the player.

The roll_algorithm is also defined in the update function. It basically toggles off the *not_rolling* boolean parameter from the player object and creates the animation of the player rolling by changing the sprite gradually and then setting it back to its original appearance. Player collisions are only active when *not_rolling* = True, so all the player collisions are toggled off for all the duration of the roll (a total of 0.5 seconds) while the player's bullet collisions are still active. At the end of the algorithm, the boolean's value is set to True again and the *rolls* parameter of the player is decreased by one.

Draw

This is the most complicated algorithm of all the program, since it checks the total lifes and, therefore, it has to determine whenever to switch between the regular game, the losing screen or the winning screen.

The first part of the function is simple, it draws the player and the enemies on screen. However, for a better code understanding, some processes that could have been defined on the update function are actually defined here. After drawing all the objects, the function deletes the bullets outside the screen in order to avoid calculating wrong collisions. Collisions, in fact, are calculated right after, and the proper explosion objects that have been created as a result are later drawn here. Finally, the deletion algorithm is run, which basically checks the dead enemies and takes them out of the screen or removes them from their lists.

The last step is to draw the HUD: the score, the “number of remaining lifes” icons and the “player’s rolls left” icons.

4. Performed work

We manage to achieve almost all of the requirements with a lot of effort, trying to keep the code organised and understandable for anyone that checks it.

When the program it’s executed, it will appear on the title screen and will not go further until the user presses Enter. To highlight that, we made the text appear and disappear slightly so the user notices the message easier.

After pressing Enter, the level is loaded. Enemies will start to move at their respective times, being their order and appearance time predetermined by the program. After 5 waves of regular enemies, a bombardier appears, then the row of 5 red enemies, then another bombardier and finally a superbomber.

If the player manages to kill the superbomber, the game will end there. If not, the game will end when the superbomber manages to escape. The winning screen will show “YOU WIN” and the player’s score.

As we explained before, when the player gets hit, all the enemies freeze on screen and the death animation triggers. When the player reappears on screen, the level is reset. All the values but the lifes are reset when the level restarts. The amount of lifes is only restored when the losing screen is shown and the game goes back to the title screen.

The only field that we did not cover properly was the graphic one. The player's helix does not turn and the background does not contain any isles or boats. However, we thought that a simpler background design would be better for this shooting-like game. Since pyxel's colour variety is very reduced, enemies and background elements appear to be at the same level, so simplicity avoids this confusion. This will be improved in future versions.

As extra details, we added background music and some sound effects like explosions or shooting. This helps the responsive part of the game, making it more fun and intuitive to play. The music also stops when killed, emphasising even more the player's death.

As a final final remark, we must highlight that the enemies behave almost exactly as the original game's ones.

5. Conclusion and personal comments

During this project we have learned a lot about python and object oriented programming, as well as learning to appreciate the modern graphic motors such as Unreal Engine or Unity, which make the task of graphic designing video games much easier than the old way of painting pixels and clearing the screen for each frame.

During the journey a lot of bugs appeared, even at the very last day. We managed to successfully solve all of them and make a functional and simplified version of the first level of the game 1942.

The game still has a lot of room for improvement, but we are proud of the final result. The most difficult, while interesting, part of the development were the collision systems since we had to apply vector concepts. Designing the loop was also complicated, since we could reach a backwards loop drawing sprites for the player's plane. However, the sideways loop also looks fine.

As a final comment, we are very proud of the circular trajectories of the red enemies, since we had to use trigonometric functions to calculate them and the final result looks just like the game.

We enjoyed the journey and we are proud of the final result.