PyGame Project Report

Group 10: Viggo Hoffsten, Olof Hedenus, Linus Hermansson and Albin Edvärn

Introduction

This project aimed to develop a game using pygame inspired by the mechanics and style of the well-known and classic “Super Mario” series. The objective was to create a playable and functional game that captures the requirements of a main character, two different enemies, a final boss and a game environment. Through our game idea we have been able to implant the requirements in a natural way and been able to create a simple but entertaining game. The game idea was born by us playing around in pygame. The project started with a circle that could jump around in the window which gave us the idea that we should develop a “Super Mario” inspired game with different levels. The game idea is that you control a main character who has to get through three different levels to complete the game. Each level has different kinds of challenges which the player must go through in order to get to the next level. The levels increase in difficulty, introducing various types of enemies and obstacles that the player need to get through to complete the level

Project rolls

The different levels facilitate our work enormously as we could naturally divide the work evenly between the group members and each level could be coded on individually. The different project roles in the group followed that structure, overall saw the project roles looked like that Viggo was the project leader as he had previous experience which made him suited to manage the project. Beyond being project leader, Viggo was also responsible for the development of the first level of the game. The second level was managed and developed by Linus who focused on creating a new level by mainly using the game environment in level 1 to create a new and completely different game environment in level 2. The third and final boss level was created by Olof with the main challenge to tie it all together. Albin was responsible for documenting the meeting protocols, schedule and writing the report. All of us have together played and bug tested each other's code and given each other feedback for further improvements.

Conceptual Design

We created our system sketch after our agreement to make a “Super Mario”-like platformer game. For a platforming game, we need a character, which has already been created, as well as enemies, obstacles, platforms obviously and an end goal. Also, as stated earlier in the introduction, we decided on going for a level structure that requires the player to reach an end goal in every level to proceed to the next one. We also don’t want the player to have to start over from the first level after every mistake, so the character should have a set limit of lives, that way the player doesn’t have to go back to a level they’ve already beaten, they instead get to make a few more attempts on the current level. However, if all lives are lost, they should be met with a “Game Over” screen and then returned to the first level with three new lives.

In broad strokes, that is the concept of the game, but in a more detailed sketch the game should work as following:
First we create a Pygame window and load in all eventual textures and colors needed for the game. Thereafter we want to define our character and its attributes, like its size, hitbox, lives and such. Before we start our main game loop, we also need to define enemies' attributes and a level counter to keep track of what level we are on, so the game knows where to put our character, enemies, obstacles, platforms and the end goal.
Now in the main loop we should start with creating our level 1, so we render the background, create and render obstacles, place the first enemy and the goal. The player has to be able to move the character, so the code should check for inputs on the arrow keys, if a key is pressed the character should move in that direction. If the character collides with the enemy or a damaging obstacle, we teleport the character back to its starting point and deduct 1 life. With this the player should be able to maneuver to the goal and flip the level counter to 2, meaning the code should teleport the player to a new starting point and then create and render the objects, platforms and the enemy for level 2. Same goes for the transition between level 2 to level 3 that should include a boss enemy.

Our time plan for the project is to have most of the main functionality of the game done by 5/11, that day we will have a meeting to see how far we've come and decide if there is more that can be done before the deadline.

Implementation

Again, as mentioned earlier in the introduction, we started with a circle that could jump around in the 800 by 600 pixels big pygame window. In order to do so, we started off giving the circle its attributes, like a starting position in the window, a radius and some other default values for its movement abilities like a default speed for x and y axis and gravity, so it falls down again after jumping. A variable "on_ground" was also created to make sure the circle also stops and does not get affected by gravity while on the ground, as it would otherwise fall right through. These attributes are in lines 86-93 in the final code. The "on_ground" variable is originally set as false, as our character, the circle, does not necessarily start from ground level. The variable is then set to true when the circle touches the bottom of the window, see lines 306-313.

main character

For the circle to be able to move we used Pygames' built in "keys" function to check if any arrow keys are pressed down. For movement on the x-axis the code checks if the left arrow or right arrow is pressed, if one is, the circle will move that direction in the default speed (beeing pixel per tick) we already assigned to the circle. For jumping however, the code also needs to check if the player is grounded, otherwise we could continuesly fly by holding the up arrow key. If the player is grounded and presses the up key, the player's y-speed is set to the "jump_strength", being -10, meaning the player will move upwards 10 pixels per tick. The

player's y-speed is though recalculated for every tick by adding the value of "gravity" to "circle_speed_y", which makes the players upwards movement slower and slower until they eventually start accelerating downwards instead till they either hit the ground or a platform (Lines 301-304). We had not yet made any obstacles or platforms the player could jump on and off, so we started with that next.

In the final code at lines 691-719, you can see the collision detection code we created. We gave our circle its hitbox and named it "circle_rect". Thereafter we use a for-loop that will loop for every obstacle in a list named obstacles, and use Pygames' "colliderect" function. What it does is check whether the circle is overlapping with an obstacle (platform/wall) or not. Also, depending on where the circle is overlapping with an object, we need it to do different things. That is why we made "if"-operations for four different cases, being when the circle is above the obstacle, to the left of the obstacle, right of the obstacle and finally under the obstacle. This is to make sure that when the code realizes that the circle and an obstacle are overlapping, the circle is instantly pushed out of the wall in the right direction. Special for when the circle is above the platform is that we also need to make the variables "onplatform" and "on_ground" true, this makes gravity not push our circle down while on a platform and it also makes the player able to jump again.

obstacles

With this collision function created, we could now start building levels by creating obstacles using "pygame.rect" and putting them all in the obstacles list and then, for every obstacle in the list we use "pygame.draw.rect" to make them visible in the window. The lines in the code file for obstacles in level 1 are 384-391. The same list was also used later on while making obstacles for the other levels, only the elements in the list are different.
Next up was creating an enemy. We defined its attributes before the main game loop being things like a starting position and default speeds for the y and x-axis (Lines 100-106). After testing different ideas, we ended up having the first enemy only go up and down, making the player have to time a jump across a gap according to the enemy's position. The enemy's movements in level 1 are done in lines 336-346. Before working on the collision between the enemy and player, we first added a variable for the amount of lives left, being "hearts" which is originally set to 3. In order to check for collisions between the player and enemy, we created the two variables "player_rect" (player's hitbox) and "fiende_rect" (enemy's hitbox) together with a third variable "fiende_kollision", being either true or false depending on if the two hitboxes are overlapping or not. If "fiende_kollision" were to be true, we would normally remove 1 heart from the player, but we made an exception for if the player is landing on top of the enemy. If that is the case we keep our hearts and make the enemy disappear. (This was later changed to have the enemy change direction and be stuck at the bottom of the level instead of disappearing). As a side note, we also made a "ground pound" mechanic meant to be used for this enemy, but found landing on top of the enemy just enough. The ground pound feature stayed inside the game however to be used simply as a movement function, see line 293-299.

The variable “circle_speed_y” is also changed to “jump_strength” making our character bounce off the enemy after landing on it. A later change to level 1 was the addition of lava on the ground, see line 397.

enemies

If the player collides with the enemy from any other direction, that is when the player loses one heart and we set the player's position back to starting position, meaning they get to try the level again. We were experiencing a bug making the player lose all 3 lives instantly when colliding with the enemy, so in order to prevent that, we added another variable named “tagen_skada”. Every time the player loses a heart, the code also checks if the player has any hearts left, if not, add the “gameover” variable and set it to True. We then use the function “show_gameover()” defined at line 156, to display a gameover screen. The actual check for whether it is game over or not is done early in the main loop at line 168 and will call the function to display the “Game Over” screen. Just a few lines above you can see another check for the “gameover” variable, where the code also checks if the “enter/return”button on the keyboard is pressed. If both are pressed the game will leave the “Game Over” screen and restart the game. This is done by moving the player back to the starting position, setting a level counter to the value 1 (the player should restart at level 1 if all lives are lost), setting the value of “hearts” back to 3 and the “gameover” variable should then be false, so we’re no longer sent to the “Game Over” screen. A start up screen was also implemented in a similar fashion later on in the project. Two new functions were also made to display the amount of lives left and what level the player is on. These are defined at line 164 and 176.

All that is left for the first level now is a way to transition to the next level which we implemented by setting positional requirements for the player. If the player meets the requirements they get teleported to the starting point for the next level, and 1 is added to the level counter. In the lines 315-334 u can see the positional requirements (goal) for each level. The main loop is actively looking at what the value of “level” is, and then afterwards creating the objects and background for the level. That way, when the value of “level” changes, on the next tick, the objects and background from the previous level will not be loaded this time, only objects, background and the enemy for the current level.

The code for level 2 is built in a similar structure to the first level. The “obstacles” list gets defined with the new walls and platforms for this level just like in level 1, but the collision for the new enemy and some new damaging spikes works a bit differently. Firstly, locations for each spike are given. Then, instead of using the “colliderect” function from the pygame library, different collision variables were made that each manually check whether the player is inside the location of the corresponding spike or not, if so, the variable gets the value “True”. Thereafter we check if the variable “kollision” is false. That way the code only finds it necessary to look for new collisions while not actively colliding with something else already. If any collision is detected, one heart is removed, the player is put back at starting position and “kollision” is

true. Code for the enemy in level 2 is a bit spread out, the collision checks are done together with the spikes but its movements are calculated at the lines 311 - 318. The enemy starts out on the left side of the screen and moves towards the right side, where the intended goal is. When the enemy is far enough out on the right side, it is then teleported back to the left side and repeats its movement. “Fiende2” was planned to be the only enemy for level 2, but we thought it was a bit easy gameplay-wise, so we then borrowed variables from the first enemy and made it move the same way the second enemy does if we are in level 2 (These are the two “bullet-bill”s in the final product).

Level 3 was meant to be the final boss level for the game, but we decided to add two more, making the boss encounter feel like a longer and bigger challenge, more suited for the “final boss”. Each of these levels' objects and goal requirements were created the same way as in the previous levels, but a separate code file was made for the boss’s attributes. That file contains different functions for the boss’ features, like how it moves, its hitbox, how it attacks and generally how it works. If you look inside the bosscode file you can see that we gave the boss some randomized values for its starting position, as well as different possible values on its y-axis speed. At line 31 we have the first important function being “spawn_boss” which simply uses Pygames “blit” function to display the boss on the screen. The function under starting at line 37 defines how the boss moves. It uses a variable “boss_exist” to check if the boss is active, if true then the boss moves its set x-axis and y-axis speeds at the same time. Just like the DVD-logo (the theme we went with for the boss) it is supposed to bounce off the edges of the screen. Therefore, the code on lines 54 to 57 checks whether the boss is along an edge, if it is, the speed for the correct axis is inverted. The “boss_collaterate” function is a collision function with the inputs of the circle’s coordinates. Last function in the bosscode is the “boss_attack” at line 68. This attack consists of having the boss shoot out smaller DVDs in random directions. Here we again use functions from python's “random” library to first determine how often the attack should occur. If the condition for doing an attack is met, we calculate a random angle in radians in the values 0 to 2pi. Once the angle is given, the code needs to calculate how fast this DVD should move in the x-axis as well as the y-axis. With python's math library we can calculate “speed_x” by multiplying the initial speed, set to 5, with the cosine of the angle we randomized. Sine is used for “speed_y”
These values are appended to a list named “DvD_projectiles”. For every DVD in the list we change their location with the help of their calculated x and y speeds, and display them on the screen. To check whether these DVD projectiles collide with the player or not, the code looks at the distance between them, which is done at line 90 using Pythagoras. On line 92 the code checks if the distance is small enough, if it is, the whole function “boss_attack” is returned as true. The last couple of lines in the function also removes all of the DVD projectiles that haven’t hit the player, but instead drifted off outside of the screen.

These 4 functions are imported into the main code file which we call upon in level 3, 4 and 5.

Other than that, those levels are similar to the first two. The obstacles list is updated with new walls and platforms, and new goal requirements are made.

At our second meeting we realized that we will have time to create animations and different textures for the obstacles, enemies and the player. We committed to the Mario theme we first were inspired by and made the player look like Mario himself.
We went back to our movement functions and added a “movment” variable to keep track of how the player is currently moving, which makes it possible to use a different picture of Mario depending on what movement is done by the player (running and jumping animations, lines 663-688). You can also see there is a sound effect for when the player jumps.
The different enemies also got unique textures from the “Super Mario”-universe. Our first enemy has the texture of a “Thwomp” which we found suitable as it only moves up and down. The second enemy is now a “Bullet bill” being shot out of a cannon. The green pipes were also added as textures over the goal requirements in level 1 and 2. For the boss levels however we used the blue “P-button”. Textures for the obstacles in levels were also added, which we implemented via a class named “ObstaclePainter”. Inside the class is a “render function” that uses the obstacles list as an input in order to display the texture onto every obstacle.

Sound

Lastly we imported music to the different levels, as well as the “Game Over”, the start screen, and the victory screen. The soundtracks and sound effects are on lines 48-63, and under is a function using the name of a soundtrack as input and then plays it. While adding music to the game, we also ended up throwing in a monologue for the boss. While that monologue is playing, the character should be stopped in place and listen to the boss before continuing. Therefore, we introduced a movement lock variable along with a timer as soon as the boss spawns in. Once 7 seconds have passed, both the player and boss can now move. That was our last addition to our now completed game.

Conclusion and discussion

In conclusion, the group has tested the waters in Pygame and constructed a conceptual design for a game. We started off with making a system sketch to plan out how the game should be implemented and different parts of the work was dedicated to all of the group members. Aside from our frequent updates in the groups GitHub, we had 2 collective meetings in person to have everyone fully grasp how along in the project we had come. In these meetings we also planned out assignments to be finished by either the next meeting or the deadline. Together we have been able to implement code with the help of the conceptual game design and its system sketch, changes and new ideas have been implemented along the way but we are happy and satisfied with how the final product has turned out.

Our work process could however have been a bit smoother. Because most of us in the group are new to pygame, we have used a lot of trial and error while writing the code. Basically, we have written code, tested if it works, and kept everything that does, but changed what doesn't. This has led to a bit of an unstable structure for the code because we don't have a lot of generally applicable functions or classes, but instead a bunch of variables needed at a lot of different places in the code. It also makes the code a bit harder to read and it makes a tougher challenge to add or remove things from the code. Our main takeaway is therefore a less codependent code structure.

We didn't make specific time estimations for a function itself, but instead used our second meeting as a "soft deadline" to see how long it took to implement the actual functionalities in the game. We were basically on schedule by the second meeting as the game was fully functional besides the two extra boss levels that were to be added. From what had been done so far, we judged that we will have time for those two levels as well as music and animations.