Object Oriented Programming – 6.4HD Task

Super Mario 29.11 - Design Report[1]

By Marella Morad – 103076428

[1] *All rights reserved to the original Super Mario Bros game. This game has not been developed to be published, only to demonstrate OOP and C# programming skills.*

Table of Contents

Overview: 3
Level Appearance: 3
Start Window: 3
Level One: 4
Level Two: 4
Level Three: 5
Exit/Finish Window: 5
Classes and Interfaces List: 6
UML Class Diagram 8
Main functions descriptions 8
Moving the Player Left and Right 8
Jumping 9
Collisions with Blocks 11
Player Resize (Enlarge and Shrink) 13
Door and Key Interactions 13
Superpowers 14
SuperJump in Level 1 14
SuperMagnet in Level 2 14
SuperVoice in Level 3 15
Video Demonstration Link 15
Full Game ZIP file Link (including image) 15

Overview:

Super Mario 29.11 is an adventure game, inspired by Super Mario Bros, initially released in 1983 (Super Mario Bros 2021), in combination with some features inspired by a game called, It Takes Two, which is also an adventure game, published in 2021 (Electronic Arts 2021).

Super Mario 29.11 features one main character, Mario, whose role is to avoid and get past all the obstacles in the level to get the key to the next level, which he can then use to open the door to the next level. This game consists of three levels, and as the game progresses, the levels get harders. However, the new feature is that Mario will be given a new superpower (powerup) for each of the levels that will help him finish the level successfully.

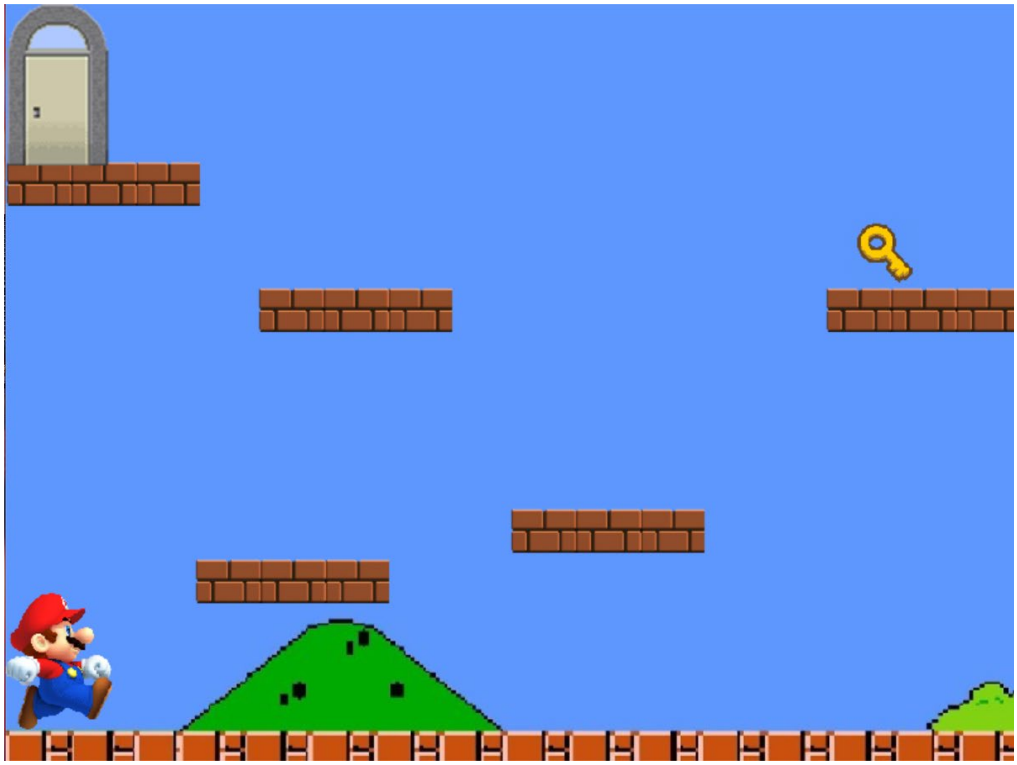
Obstacles in the game include, high normal blocks, lava blocks, spiked blocks and glass blocks. On the other hand, Mario will have three superpowers, corresponding to the three levels as follows:

1. SuperJump in Level 1
2. SuperMagnet in Level 2
3. SuperVoice in Level 3

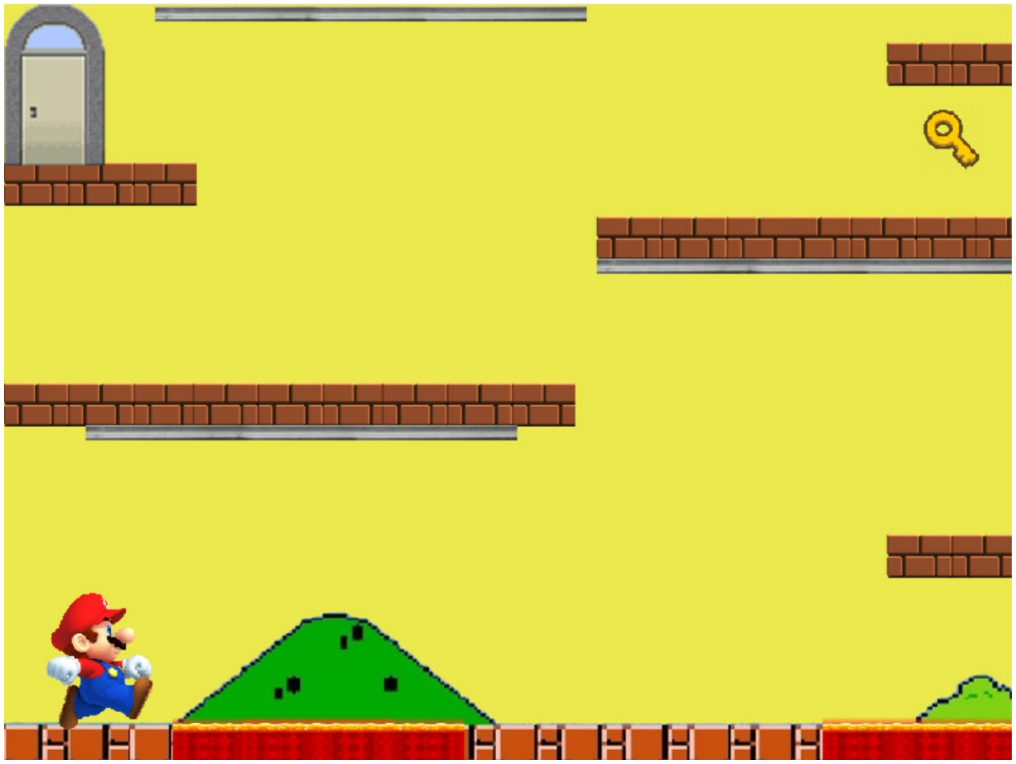
Level Appearance:

Start Window:

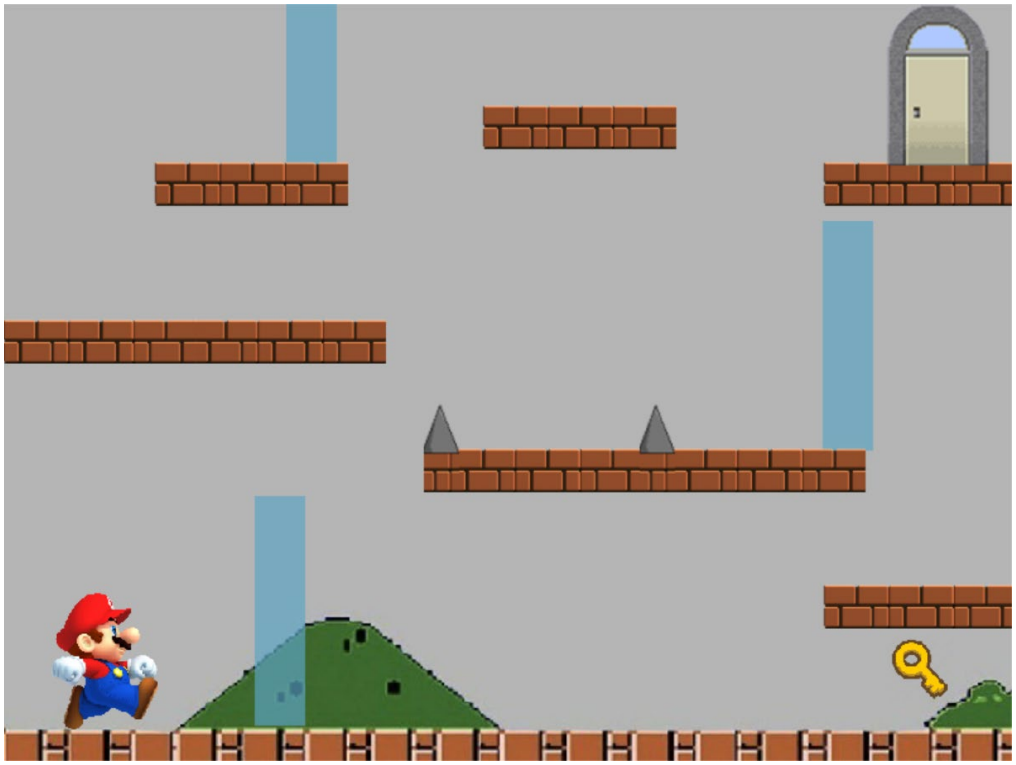
Level One:



Level Two:



Level Three:



Exit/Finish Window:

Classes and Interfaces List:

Table 1 Classes and Interfaces list, including responsibilities

Name	Type	Responsibility
Inheritance and Polymorphism		
Block	Abstract Class	Generalised form of the different types of blocks in the game (polymorphism)
Glass Block	Class	All these classes inherit from the Block class. Their responsibilities differ based on the collision they have with the player (explained later)
Lava Block	Class	
Magnetic Block	Class	
Normal Block	Class	
Spiked Block	Class	
Following the Strategy Design Pattern		
CollisionProcessor	Class	Responsible for running the two types of collisions on each of the blocks in a level
HorizontalCollision	Class	Checks for horizontal collision between the player and a block using their bounding rectangles, and an intersection rectangle for adjustments (left and right collisions only)
		Performs different functionality when the block is "glass", as it works with the superpower (SuperVoice) to break the glass block when the player collides with it having used his superpower.
Vertical Collision	Class	Checks for vertical collision between the player and a block using their bounding rectangles, and an intersection rectangle for adjustments (top and bottom collisions only)
		For bottom collisions (player lands on a block), the player is permitted to land on normal and glass blocks only. The player will fall if he tries to land on a magnetic block and will die (Reset) if he lands on a lava or spiked block. For top collisions (player's head collides with a block), the player can hang from a magnetic block, only in level 2, given that he is using his superpower (SuperMagnet)
ICollision	Interface	Provides the collision check method to be used in CollisionProcessor, Horizontal and Vertical Collision classes
Door	Class	The door of a level (three doors in total in the game). Is responsible for checking if the player has ArrivedAtDoor, by checking intersections between the player and the door's bounding rectangles and if the player HasKey. It is also responsible for setting the next level and next superpower (SetLevel), which are

		passed to it as parameters when the door is created (using its third constructor)
Following the Singleton Design Pattern		
Game	Class	Responsible for the construction of the levels, CreateGame() and running the functionalities based on the user's keyboards input
Inheritance and Polymorphism		
GameObject	Abstract Class	Most generalised object type in the game, child classes inherit properties (Name, X and Y) from it
IDrawable	Interface	Provides the ability for the object to be drawn when implemented by implementing its Bitmap property, and its Draw() void method
Key	Class	The key of a level (three in total), responsible for taking key functionality, which is achieved when the player collides with the key (turns the player's HasKey property to True), as well as the appearance of the key (once the key is taken by the player, it disappears from the level)
Level	Class	One of the major classes in the game, it is responsible for drawing the blocks (added to it from game class), drawing the key and the door (if not null, i.e. not the start and finish windows). It also runs the CollisionResponder functionality, which runs PlayerBlockCollision for each of the blocks in the level (including both vertical and horizontal collisions), along with key and door functionalities.
Direction	Class	Purely for display purposes, it checks the direction the player is facing, and displays the corresponding Bitmap
Player	Class	The biggest and most important class of the game, it handles the player's movement (left, right and jumping). It uses other classes to perform changes on the player (such as direction and resize classes). It is also responsible for displaying level specific instructions for the player in console window.
Resize	Class	Uses Direction class to identify the direction the player is facing, and also gets the Bitmaps from it to display the correct bitmap when the player requests to Enlarge and/or Shrink. It is also responsible for blocking enlarge if it will cause a collision between a block and the enlarged Mario Bitmap.
Inheritance and Polymorphism		
Superpower	Abstract Class	Provides the Name and Description properties for the three superpowers, and also the abstract Use method which is overridden in all of the three child classes
SuperJump	Class	Gives the player the ability to jump higher by changing the player's vertical speed (DY)

SuperVoice	Class	Give the player the ability to break glass using his voice (by setting the player's Sing field to true)
SuperMagnet	Class	Gives the player the ability to hang from a magnetic block (by setting the player's Attract field to true)
All three superpowers are used when Ctrl button is pressed (input handled in Game class)		

UML Class Diagram

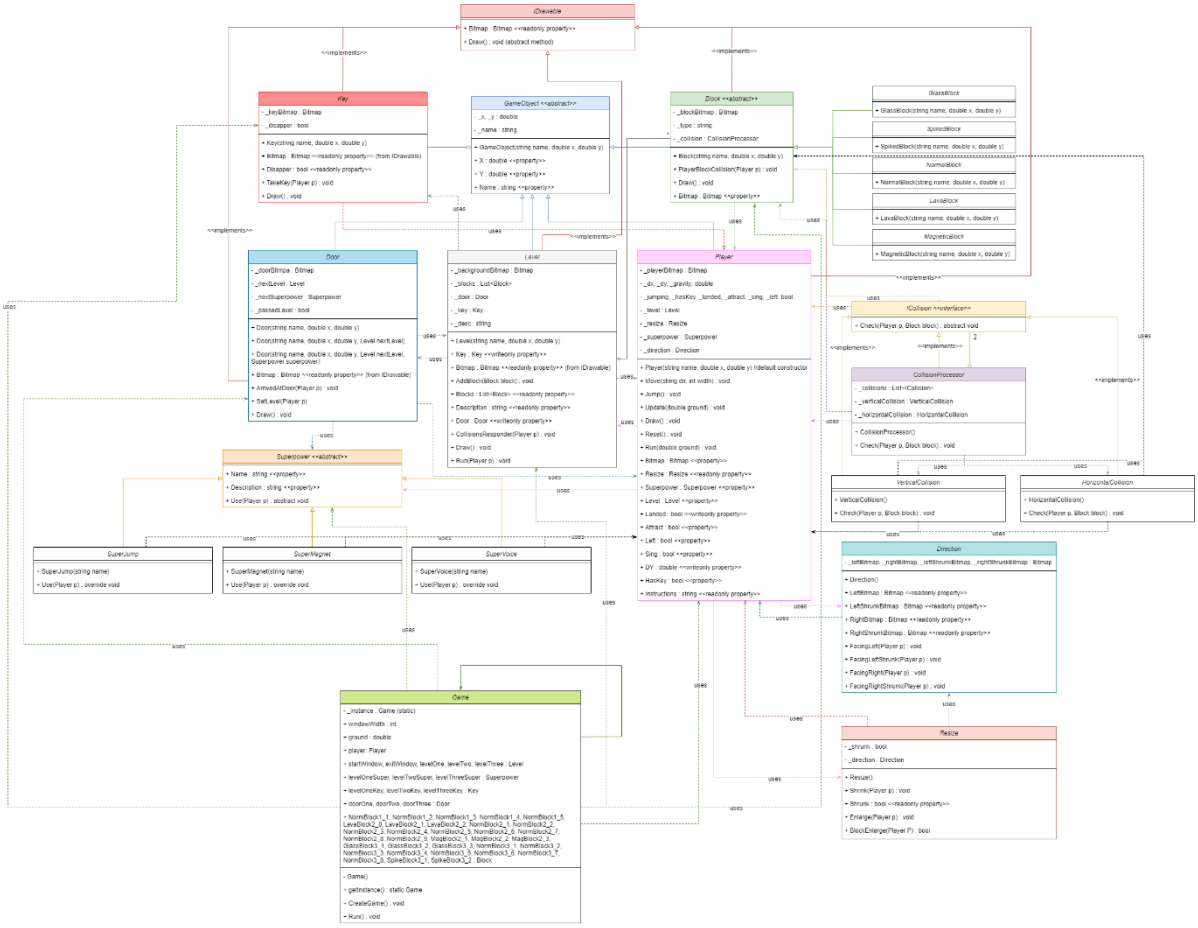


Figure 1 UML Class diagram with all associations marked and colour coded for clarity

Please note: The main UML Class diagram will be submitted separately, with only the main associations displayed (some uses associations will be removed for clarity)

Main functions descriptions

The functions will be explained in order of playing the game (Level 1 to Level 3)

Moving the Player Left and Right

The main method responsible for moving the player, is the Move(string dir, int windowWidth) method contained in the player class. Figure 2 shows a sequence diagram of how the player moves to the left, same sequence is followed when the player moves to the right

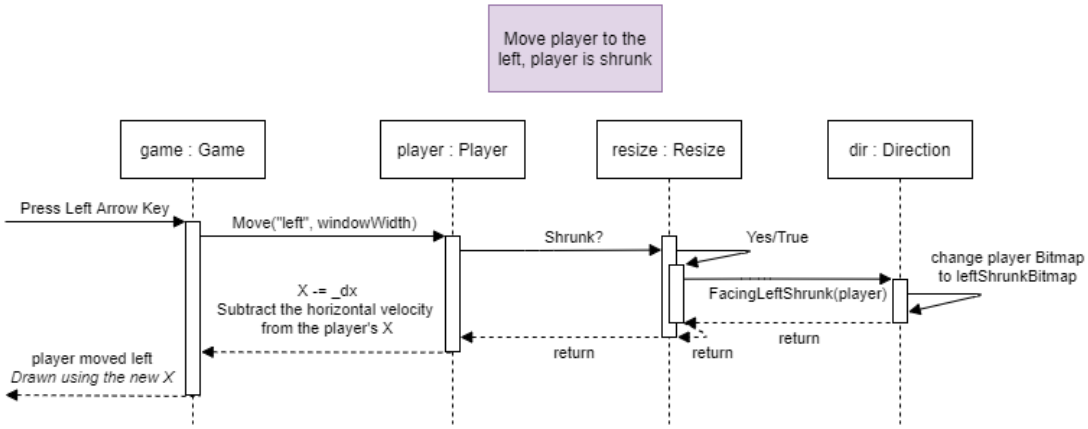


Figure 2 UML Sequence diagram of the move command (case: move player left, player is shrunk)

An additional check is in place, to check if the player is going beyond the game borders (which is why I pass in the window width as a parameter for the move method). Initially my design had both the jump and the move functionalities under one method, however I decided to break them down into two methods to make the methods more specific.

Jumping

This functionality required the use of physics to be achieved as the player undergoes projectile motion when jumping. Therefore, after some research, I was able to implement physics in my game. I used three fields that are responsible for creating a realistic jump:

- _jumping Boolean field: is responsible for only allowing one jump at a time
- _dy double field: is responsible for changing the player's Y coordinate
- _gravity double field: is responsible for changing the DY of the player, ultimately affecting the player's Y coordinate

These three fields work together inside two methods, as shown in Figure 3

- Jump() void method: called in Game class when the player presses space or up
- Update(double ground) void method: called continuously in Game Run() method

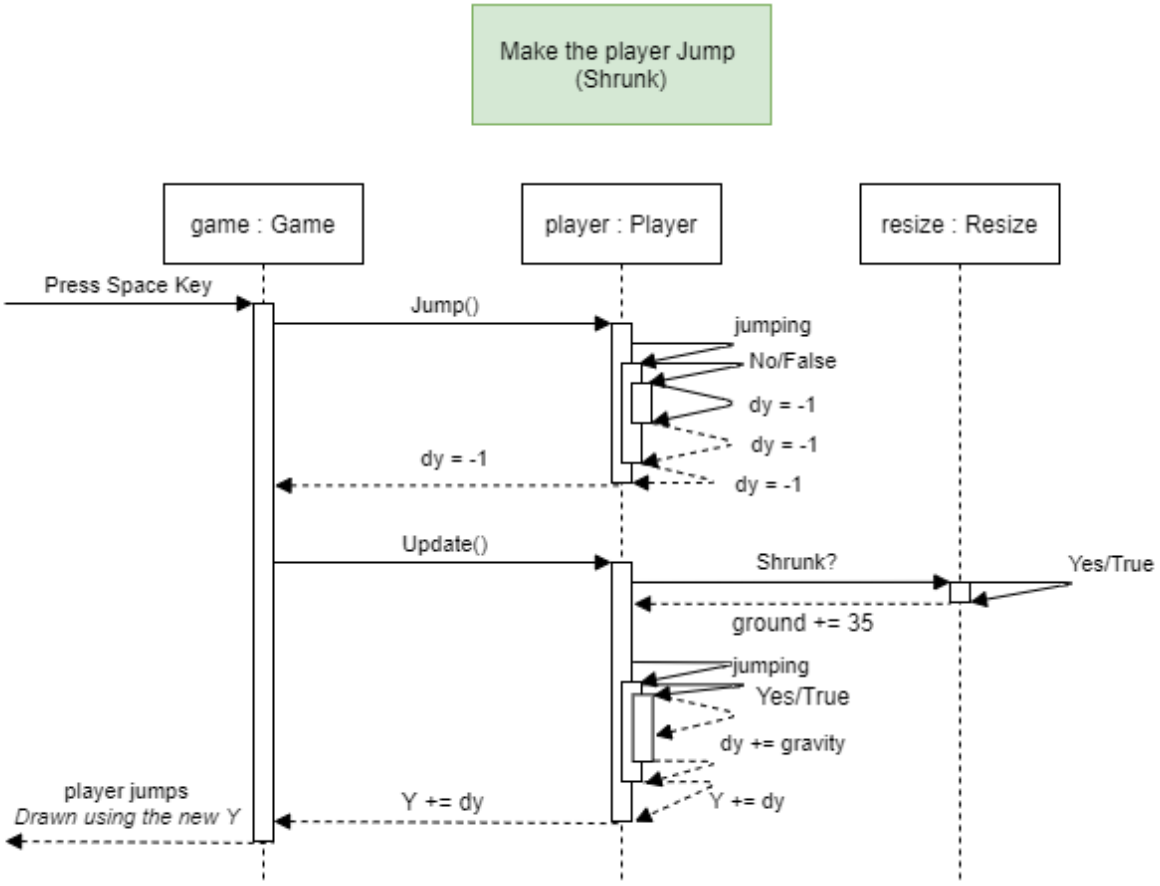


Figure 3 UML Sequence Diagram of the Jumping functionality (case: make the player jump, player is shrunk)

Collisions with Blocks

Collisions in this game are implemented using the strategy design pattern, since the two collisions only differ in their behaviour which is defined in the check method in each of the classes (implemented from the ICollision interface). The collisions are processed in the CollisionProcessor class which calls the Check methods for both the vertical and horizontal classes.

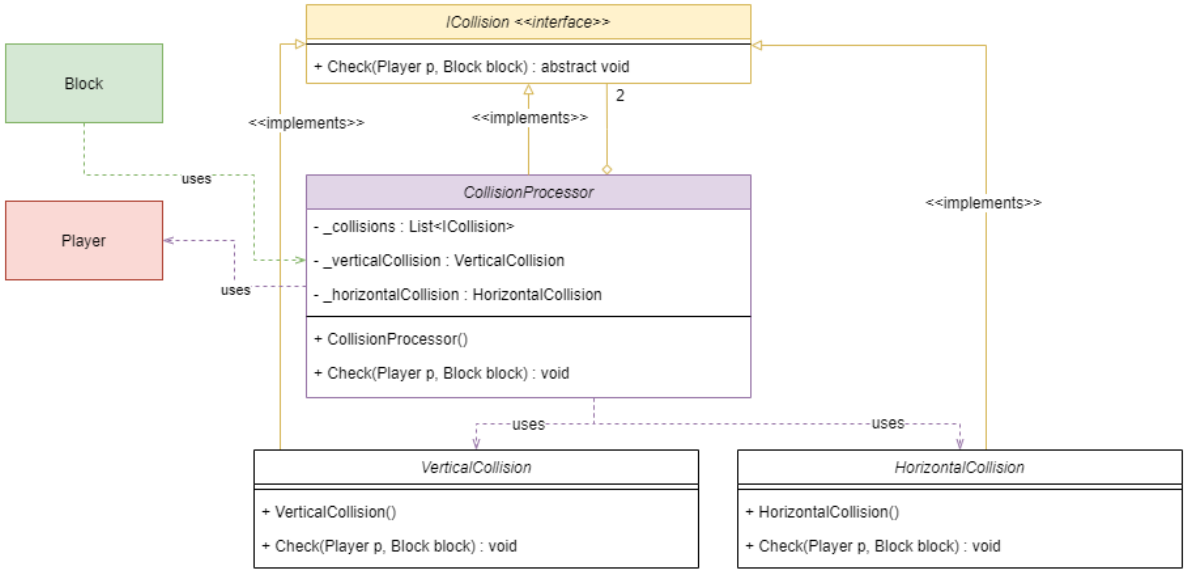


Figure 4 UML Class diagram describing collisions functionality

One of the player's basic collisions is which a normal block. If the plyer collides with a normal block either horizontally or vertically, it will be adjusted based on the intersection rectangle created using the Intersection function from Splashkit library, which returns a rectangle that represents the intersection of two rectangles.

Table 2 Snippet from the Check() method from HorizontalCollision class showing the logic of bounding rectangles intersection and player adjustment

```
public void Check(Player p, Block block)
{
    Rectangle playerRec = p.Bitmap.BoundingRectangle(p.X, p.Y); //getting the
bounding rectangle of the player
    Rectangle blockRec = block.Bitmap.BoundingRectangle(block.X, block.Y); //get
the bounding rectangle of the block
    Rectangle intersection = SplashKit.Intersection(playerRec, blockRec); //get
the intersection rectangle and a save it as a rectangle
    if (intersection.Height > intersection.Width) //horizontal collision
    {
        //Checking player intersection with the block from the LHS
        if (SplashKit.RectangleRight(playerRec) >
SplashKit.RectangleLeft(blockRec) && SplashKit.RectangleRight(playerRec) <
SplashKit.RectangleRight(blockRec))
        {
            p.X -= intersection.Width; //subtracting the intersection width from
the player's X coordinate
        }
        //Checking player intersection with the block from the RHS
        else
        {
            p.X += intersection.Width; //adding the intersection width to the
player's X coordinate
        }
    }
}
```

In a more simplified way, the adjustment of the player's coordinates is done by adding or subtracting the width and height of the intersection rectangle from the player's X and Y coordinates respectively as shown in Figure 5.

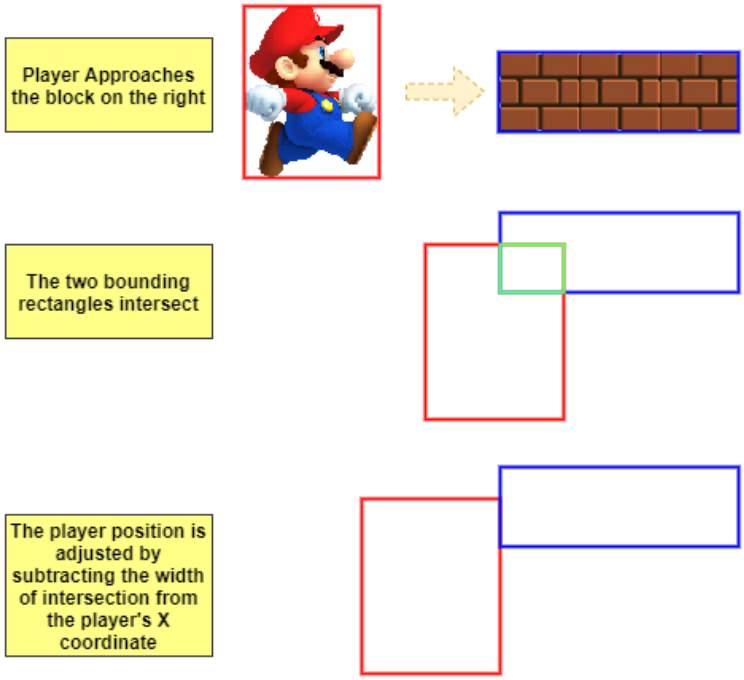


Figure 5 Collision Adjustment steps

Another main player collision is with a resetting block (i.e. Lava or Spiked Block). If the player collides with one of these blocks vertically from the bottom (player's feet collide with the block), the player is reset to its default position and appearance (using the Reset method in Player class), see Figure 6 for more details.

The same sequence is followed when the block type is "spiked"

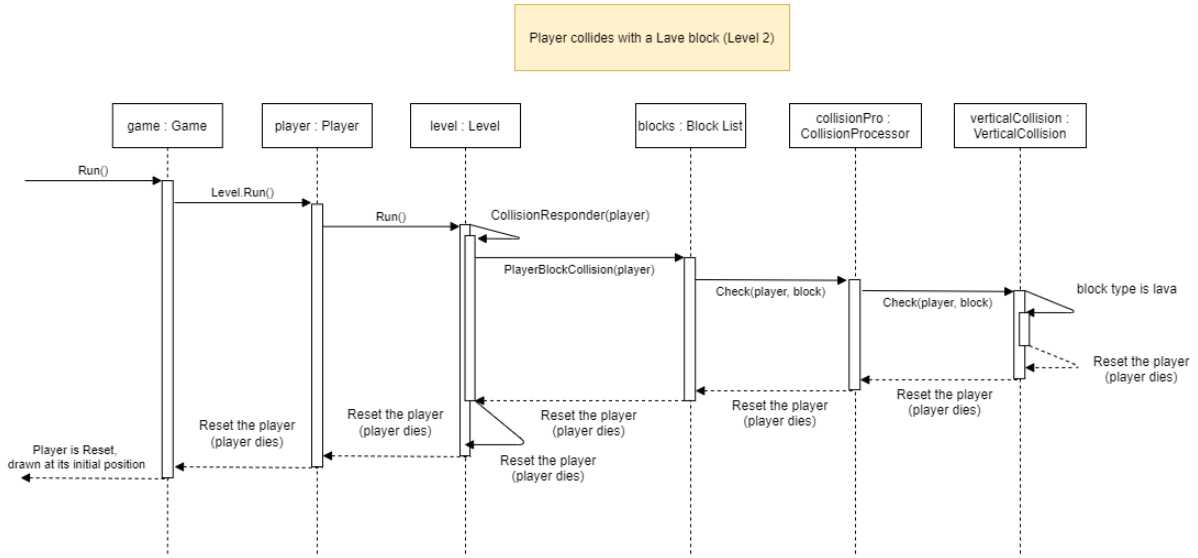


Figure 6 UML Sequence diagram of player collision (case: player collides vertically with a lava block)

Player Resize (Enlarge and Shrink)

I added the enlarge and shrink functionalities, to allow for added complexity of the game, as well as give the user a bigger control over its player. After adding the enlarge and shrink functionalities, I created some situations throughout the game that will require the player to be shrunk to achieve them. One of these situations is shown in Figure 7.

Figure 7 A situation that requires Mario to be shrunk

In some other cases, the player's enlarging will lead to a collision with a block above it, and so, the Resize class has a method to block enlarging in such cases. Figure 9 shows the sequence of calls to deny a player's enlarge request.

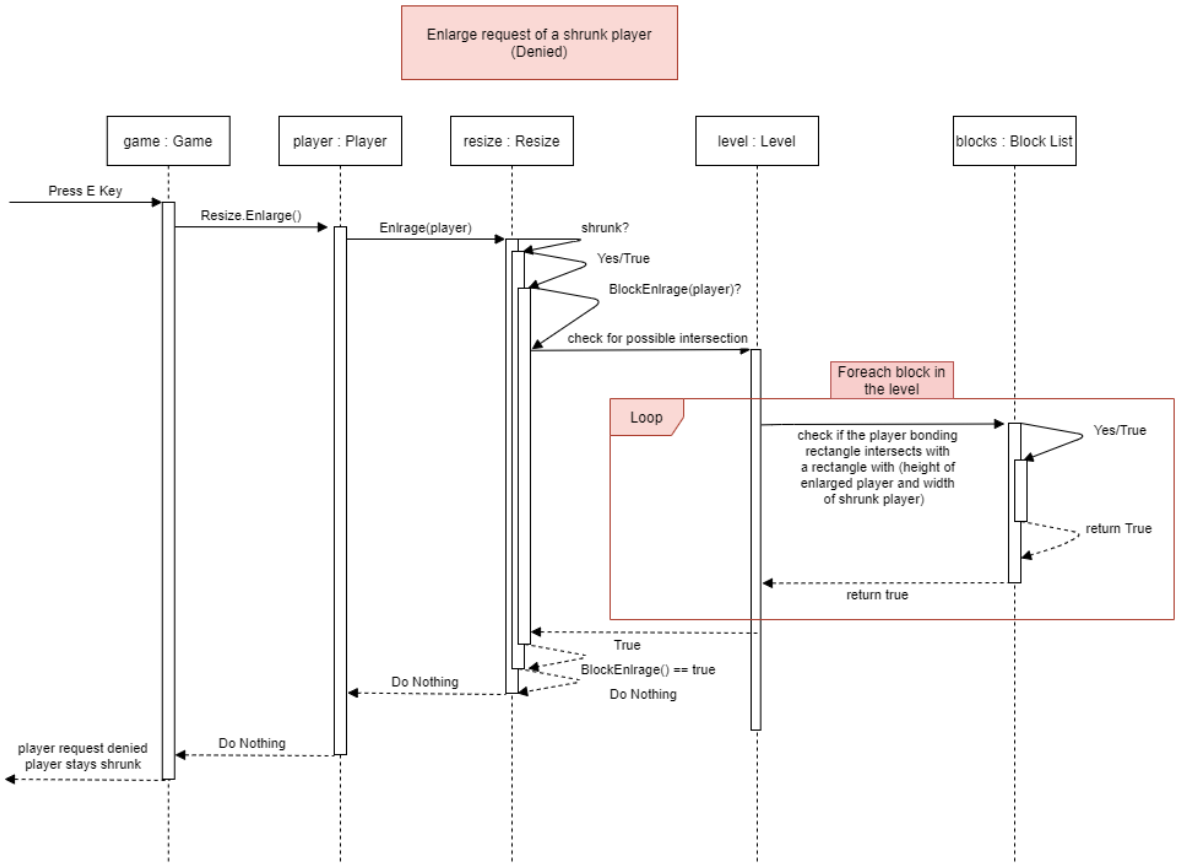


Figure 8 UML Sequence Diagram of Resizing request (case: player enlarge request is denied)

Door and Key Interactions

For the player to move to the next level, he needs to collect the key `key.TakeKey()` and get to the door to open it `door.ArrivedAtDoor()` and `door.SetLevel()` as shown in Figure 9.

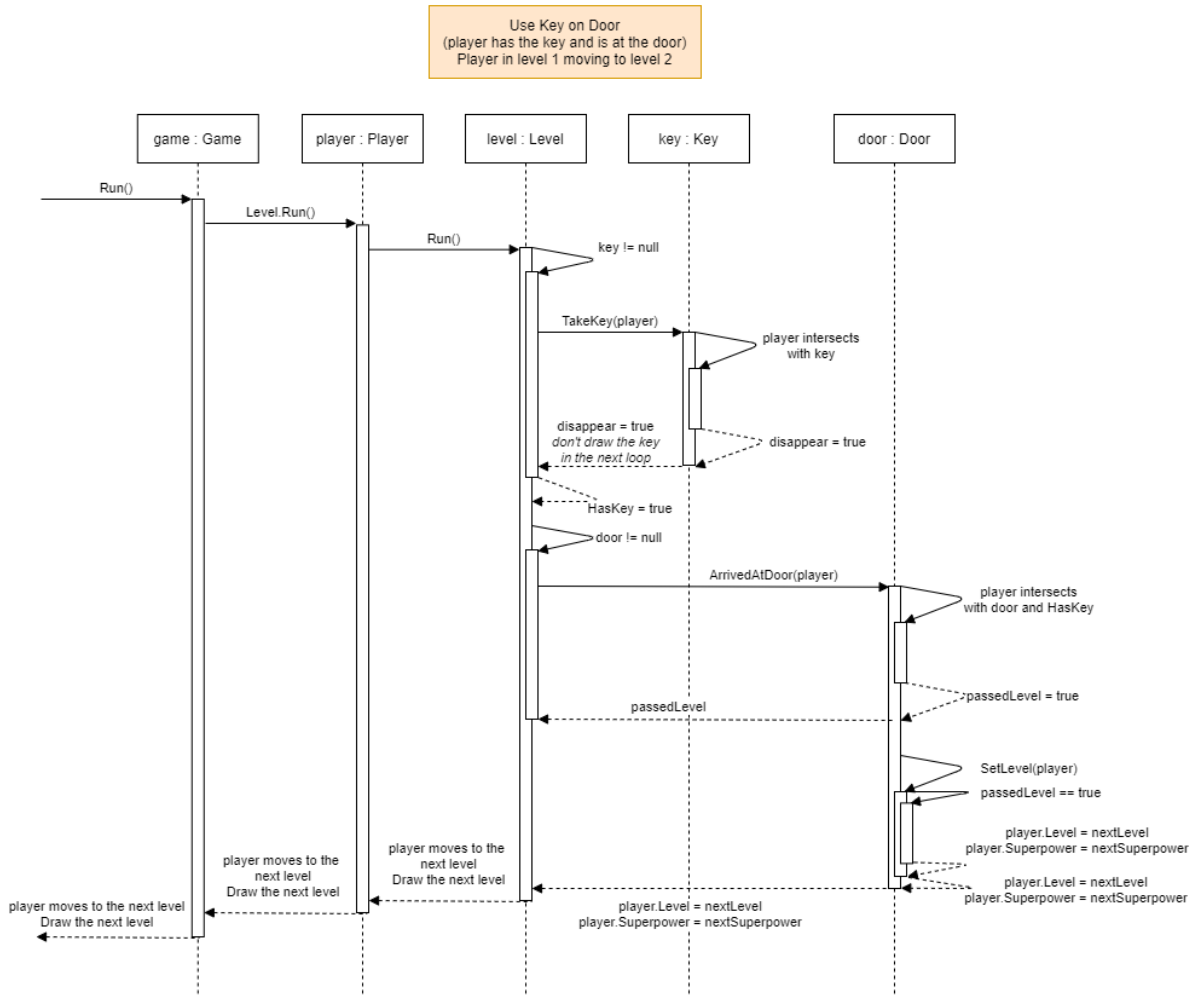


Figure 9 UML Sequence Diagram of using the key to open the door (case: player has the key and is at the door)

Superpowers

As mentioned earlier, the player is provided with a superpower as soon as he enters the level. I will show the sequence diagram for the using the SuperMagnet and the other two superpowers follow the same sequence.

SuperJump in Level 1

This superpower modifies the player's vertical speed, i.e., dy, by enlarging its magnitude (changing it from -1 to -1.2). This gives the player the ability to jump higher to reach blocks that using a normal jump is not sufficient.

SuperMagnet in Level 2

This superpower modifies the player's Attract property and sets it to true when called. The sequence of steps for using the SuperMagnet is show in Figure 10.

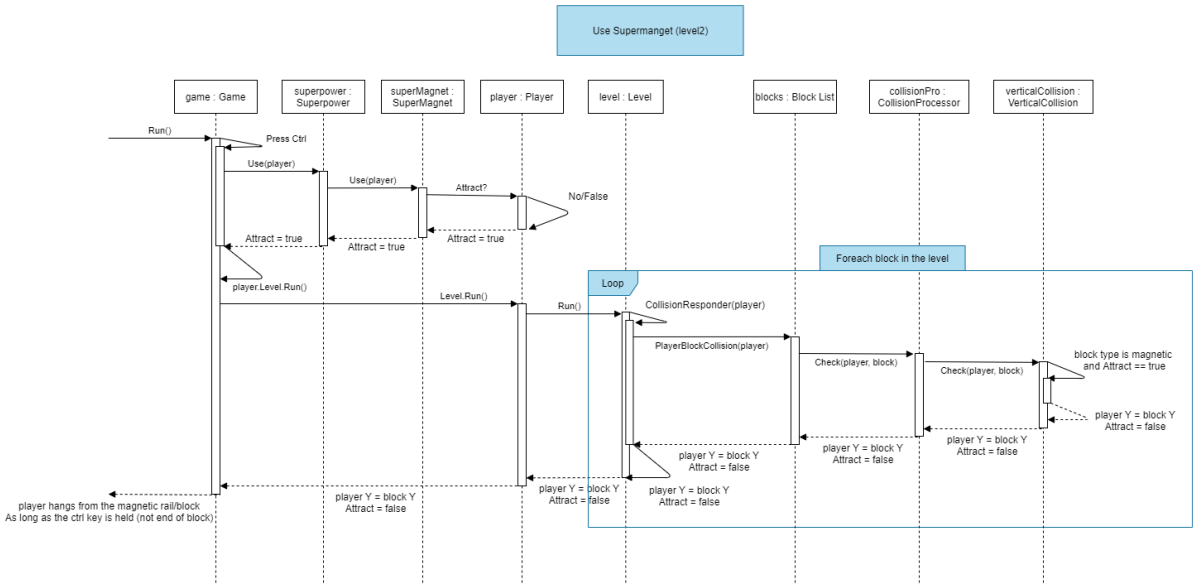


Figure 10 UML Sequence Diagram of using superpowers (case: player uses his super magnet power in Level 2)

SuperVoice in Level 3

This superpower modifies the player's Sing property and sets it to true when called. The sequence of steps for using the SuperVoice is very similar to the one shown in Figure 10.

Video Demonstration Link

[Game Demonstration YouTube Link]

Please note, Mario Bros music has only been added to the video as a background music and is not part of the game.

Full Game ZIP file Link (including images and diagrams link)

[OneDrive Game Link]