Introduction to the project

This project consists in the development of a game, an arkanoid clone, using the 2d-engine previously created. For this purpose, some new entities that inherit from drawable entity have been created. These new entities are: wall, brick, ball and player. Most of these entities, except wall, have their own update function as background. The update function is used during the game loop to control the entity accordingly to its behavior on during the game.

The basic entities have been improved through virtualization, making entity and drawable entity abstract classes with virtualized methods. This virtualization has made possible to improve and reduce the code used to store the entities of the scene.

Another improvement made over the old design has been the outsource of the user interface. Having now its own class to iterate over most of the ImGui functionalities. The UI has also been updated to give support, edition and creation, to the new game entities.

The external libraries used by the application haven't changed. Even so we think it's important to remember which of them are being used and take the chance to thank it's creators. The graphical entities are still being integrated using the library SFML (Simple and Fast Multimedia Library) created by Laurent Gomila. To operate through all these entities the project continues using the library IMGUI (Immediate Mode Graphical User Interface) created by Omar Curnat. These libraries need to work together. Therefore, we use the binding for SFML-IMGUI created by Elias Daler. It's also important to notice that the application still uses the library tiny file dialogs created by Guilleme Vareille to create native Windows windows. To save and load the scenes the library JSON for Modern C++ created by Niels Lohmann is also still being used.

Data structure

The data structures can be found at the doc folder, it has been created with Doxygen.

Strategies and Solutions to the Development Problems

First, let's talk about the virtualization. A new variable Type has been added to entity that is enum. This has been done to be able to know from the drawable entity with which of entity we are operating. This way we can cast at any time the drawable entity to the specific entity it is. This Type is protected as it's the responsibility of every entity in its constructor to correctly assign it to its type.

```
enum Type {
  kRect = 0,
  kBackground = 1,
  kLabel = 2,
  kSprite = 3,
  kWall = 4,
  kBrick = 5,
  kBall = 6,
  kPlayer = 7
};
```

Let's talk about the functions that have been virtualized. The unuse function that the pools use to reset the entities has been declared as a virtual function at entity. The draw function has also been virtualized, being this one of the key points for making a unique drawable entity container at scene. The update function has also been virtualized, so the main loop of the game can easily update the entities every frame without needing a specific type variable for every type. Another functionality that has been improved using the virtualization is the check of collisions. A new function called getBoundaries has been introduced and made virtual at drawable entity. The purpose of this function is getting the sf::FloatRect that contains the entity. This way making possible to create the check collisions functions just one time at drawable entity as they will only need to check if the boundary has collided with a point or another Rect. The virtualization of these functions makes it possible for all these operations to be accessible through drawable entity.

To create the game new graphical entities have been added. We have the wall class that inherits from Rect, and it's mostly a wrapper of it to separate the game object from a normal rect. We also have the brick class that inherits too from Rect, in this case it's not a simple Rect as it also has lives, score and an update override. This method is in charge of changing the set active flag when he is dead, or setting it to active if we want to resurrect it in edit mode, as well as adding it's score to the total score once dead. The next we have is the ball class that inherits from Sprite and has a speed vector it also has an override of the method update that is in charge of changing its position. At last we have the player class that also has a speed vector and process the input of the player in its update method.

To benefit from the fact of using virtual entities the data structures containers of the scene have changed. Previously there was a mapping for every kind of entity while now there's only one of drawable entities. This way is easier to maintain and operate with it making the code clearer. Nonetheless, this hasn't been done for the edit mode of the user interface that still uses a pointer for every specific type of entity. This has been done this way to avoid a casting from drawable to the specific type at every frame while the object is selected, as ImGui is an immediate User Interface meaning that it is rebuilt at every frame.

At last, a tag system recognition has been implemented to implement the behavior between different objects. For example, the label with tag 14 knows that must update its own value with the score of the game manager. Another example is for the collisions, as the ball will only bounce with the elements with an specific tag.

Personal Tasks and Workgroup Plan

To divide the work between the two members of the team we have implemented agile methodology, SCRUM to be more accurate. The kanbas used to monitorize the workflow can be found at: Scrum Kanbas.

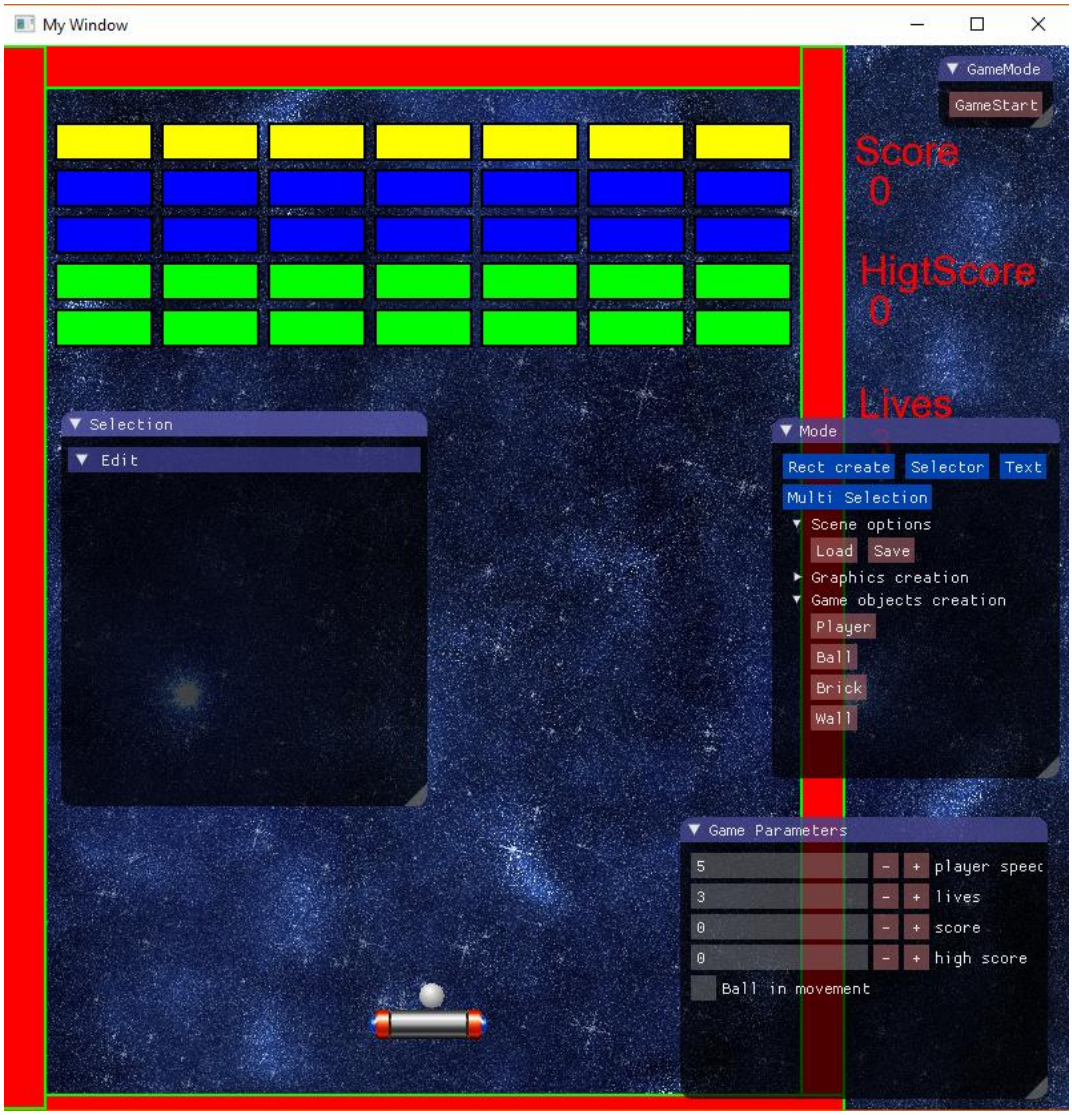
To summarize the distribution of the multiple tasks has been:

Sebastián Adrover Pedrosa: Task 2, Task 4, Task 5, M1(Especial efficiency through mapping for selecting entities and draw with z-Order using json library) and M2 (implementation of native windows through tinyfiledialogs)
José María Martínez Carvajal: Task 1, Task 3, Task 6, M1 (efficient creation of graphical entities through Factories and Pooling) and M2 (alternative grahpic library use with SFML).

Short User Manual

First, to create the executable file two bat files can be found at the build file. compilacion_d.bat in case you wish to compile in debug mode and compilacion_release.bat for release mode. In case they are not present the following dll must accompany the exe file: sfml-graphics-2.dll, sfml-system-2.dll and sfml-window-2.dll. These dll can be found at the 2d_engine\deps\SFML\bin path.

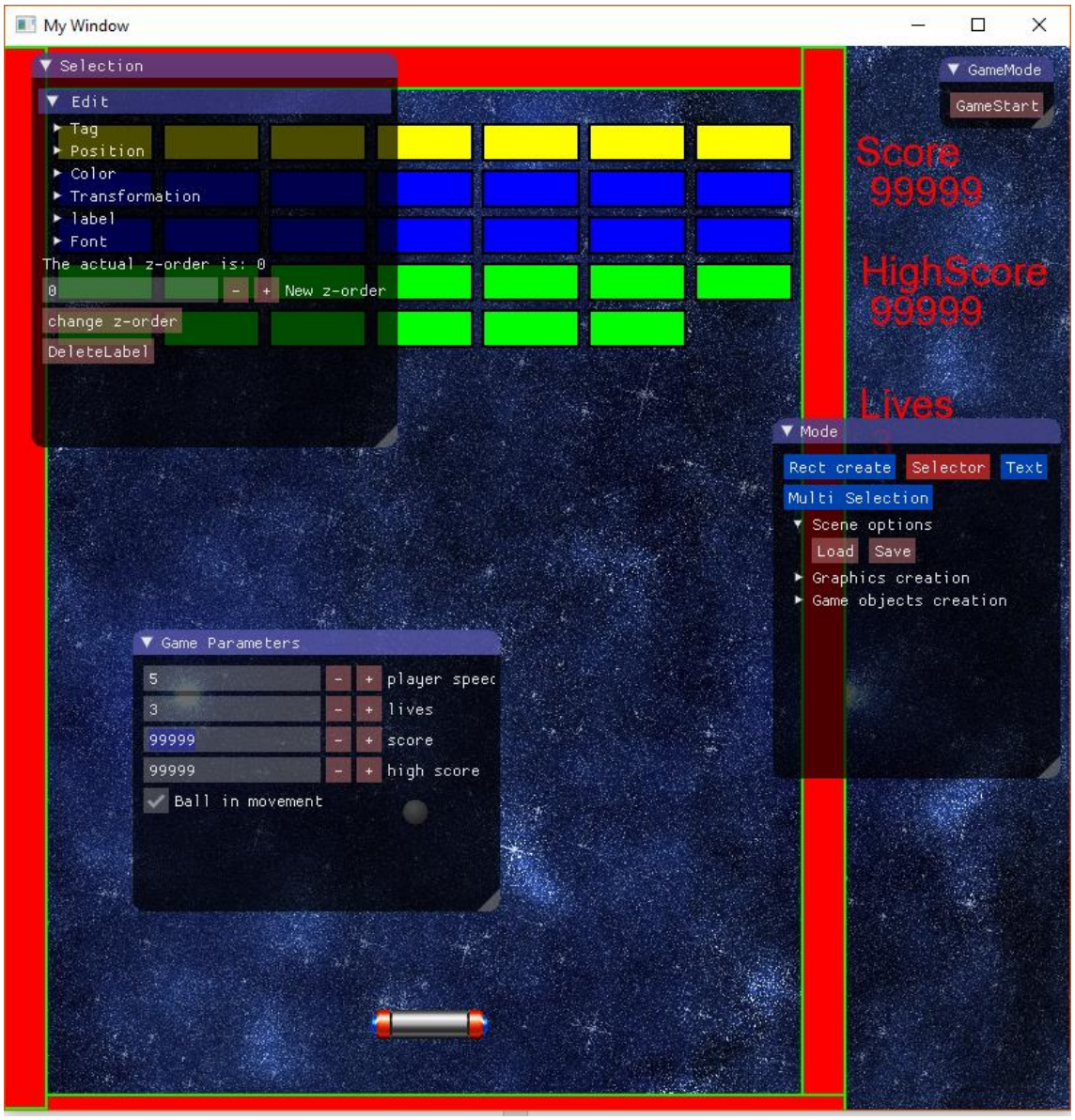
Once we execute the exe we will see the main window of the application. The game is saved at the scene Arkanoid.json. So, the first thing that needs to be done is go to Mode->Scene options->Load and select the scene. Once opened we will see the following:



To start the game, we need to press the GameStart button at the GameMode menu. The first thing we need to do is “throw” the ball using the space key to start. We can move the player using the left and right key arrows. As an arkanoid game the objective is to destroy all the bricks. The color of the brick indicates how many hits are needed to destroy them being green one hit, blue 2 hits and yellow 3 hits. The more hit points the brick has the more score it gives, of course in edit mode we can cheat and adjust how many points we want the brick to give us.

The game can be paused at any time pressing again the GameStart button. Once paused we can iterate with the different game objects and change their properties.

We can see the score of the game at the right of the screen as well as the lives we still have.



Note that we can cheat the game through the game parameters menu and set the score to the one we wish. Once we lose all our lives we need to press enter to restart the game. Once restarted the total score will be reset and the high score will update to our highest score once we throw the ball again.