FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Taller de Programación I (75.42)
Facultad de Ingeniería
Universidad de Buenos Aires

WORMS REMAKE

Final group project

Technical Documentation

2^0 Semester 2023

First Submission: 21/11/2023

Second Submission: 5/12/2023

Members	Padrón
Facundo Huttin	107854
Theo Lijs	109472
Ivan Erlich	105989

Introduction

In this documentation, you'll find a basic but complete explanation of the project structure illustrated by diagrams with their corresponding explanation. You will see how the threads are handled, how the main and most common classes are modeled and how the protocols are defined.

Content

Software Requirements

OS: GNU/Linux
Dependencies:

- SDL2
- SDL2_IMAGE
- SDL2_TTF
- SDL2_MIXER
- QT5
- QT5_MULTIMEDIA
- BOX2D
- yaml-cpp
- CMake versión 3.8

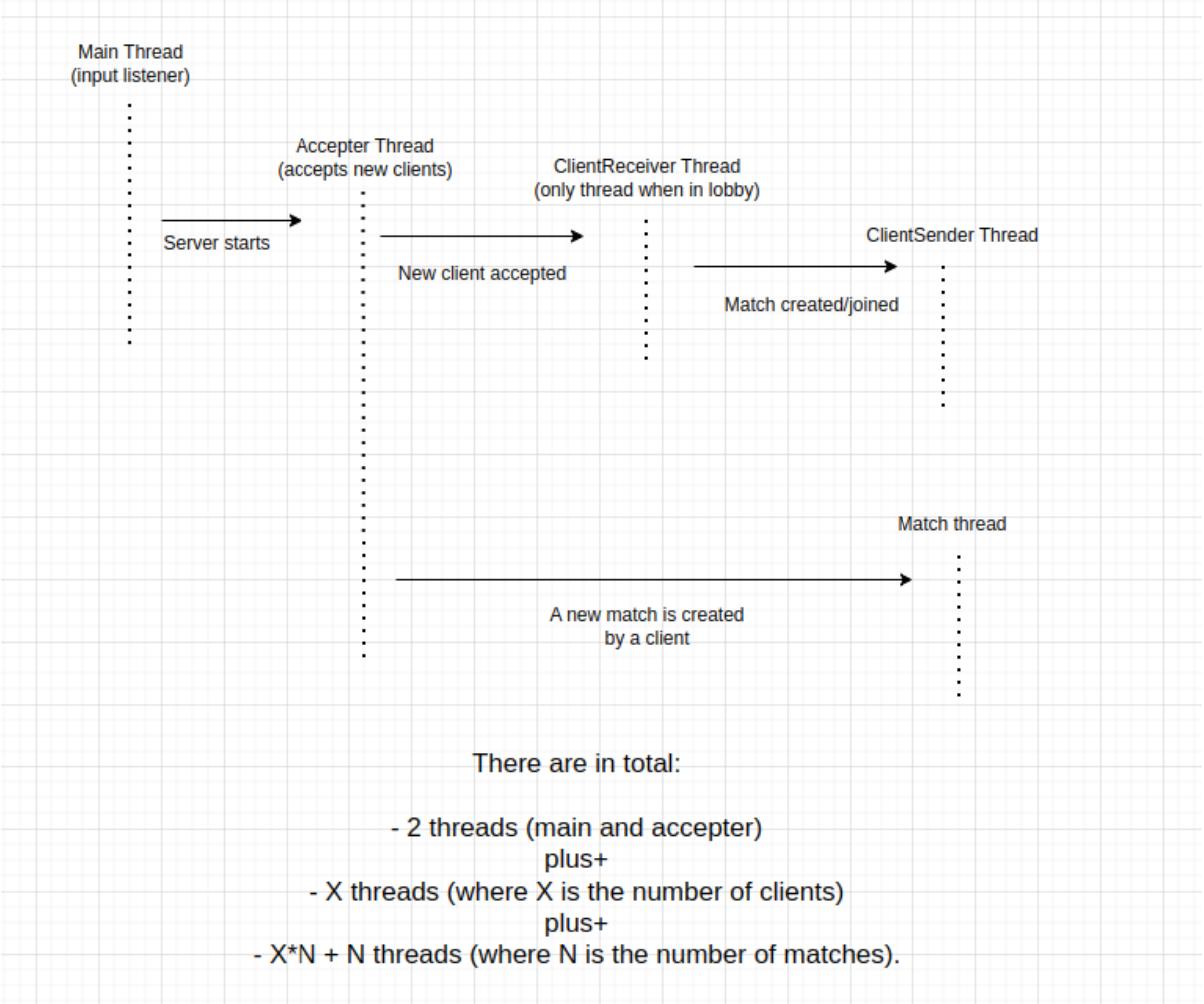
Depuration:

- Pre-commit (cppcheck, cpplint, clang-format)
- GDB y Valgrind

Threads:

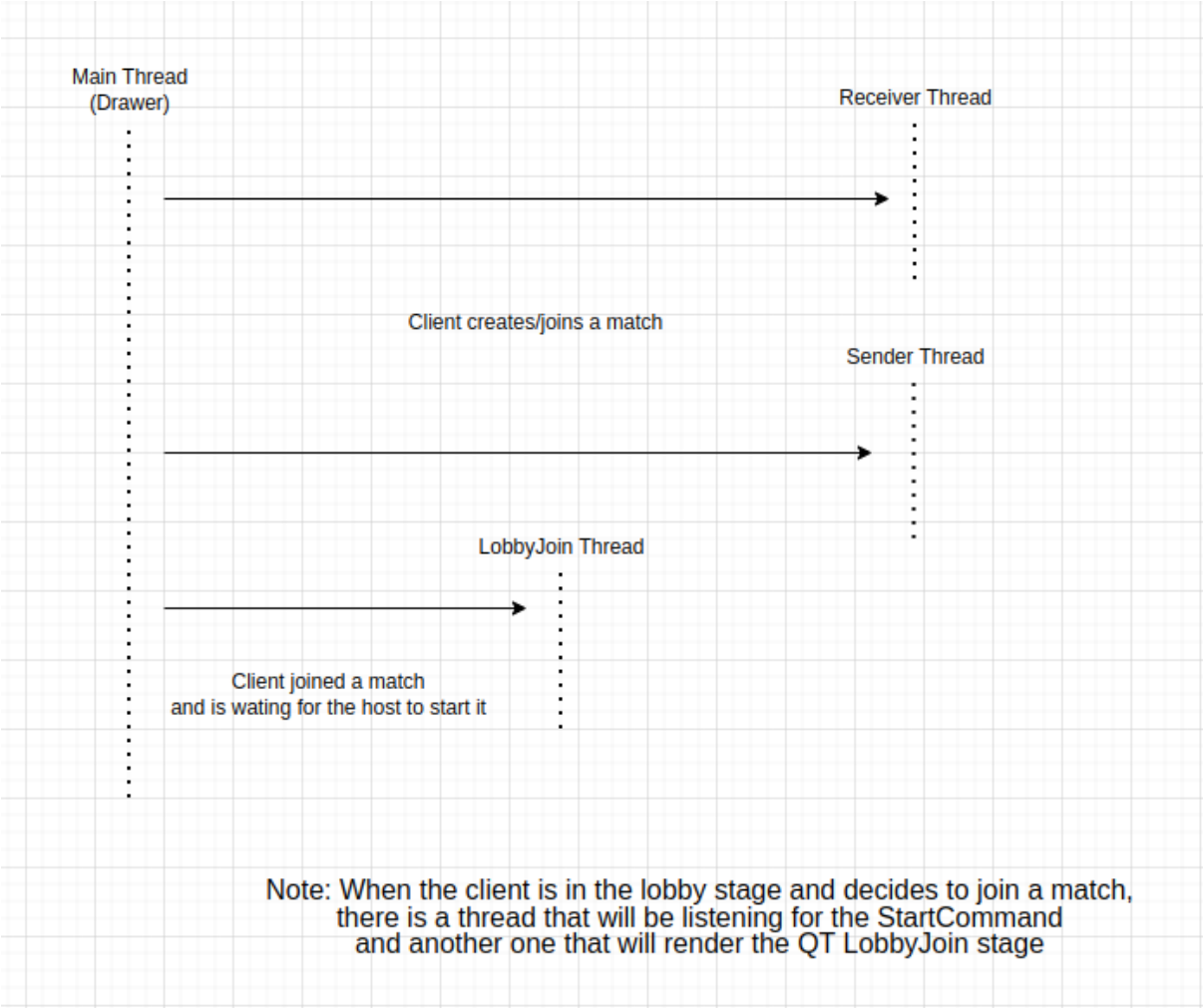
Server:

Lets begin with what we believe is most important, the thread handling in the server side:



In the diagram above , we can see that as soon as the server starts, there will be at least two threads running, the Main and the Accepter. One will be listening for a terminal's prompt ('q') to stop the server and the other one is responsible for accepting new clients.
Once a client is accepted, a new thread will be created by the Acceptor and the communication between that new thread and the client will be synchronous until a Match begins.
As soon as a match starts, a new thread will be created by the client's thread, which will now be a "receiver thread" and the new one will be the "sender thread", leaving us with two threads per client instead of one (asynchronous communication now).
On the other hand, when a client decides to create a match, the Acceptor thread (the MonitorMatches to be precise) will create a new thread for each match.
If we do some math, we can see that the number of threads will be: 2 + X + XN + N
Where X is the number of clients and N the number of matches. Here is an example:
Imagine we have 2 clients already playing one match, if we count how many threads there are alive we'll have: 2 (Main+Accepter) + 2 (two clients) + 2*1 (one match, then 2 extra threads) + 1 = 7.

Client:



We can observe that the client is very similar to the server clients. There will be three main threads during the match, but, different from the server, in the lobby stage we will have one thread except when a client wants to join a match. In that case, a temporary thread will be thrown so it can listen to the starting match while the main thread renders the menu. Therefore, there will be no more than three threads running at the same time on the client side.

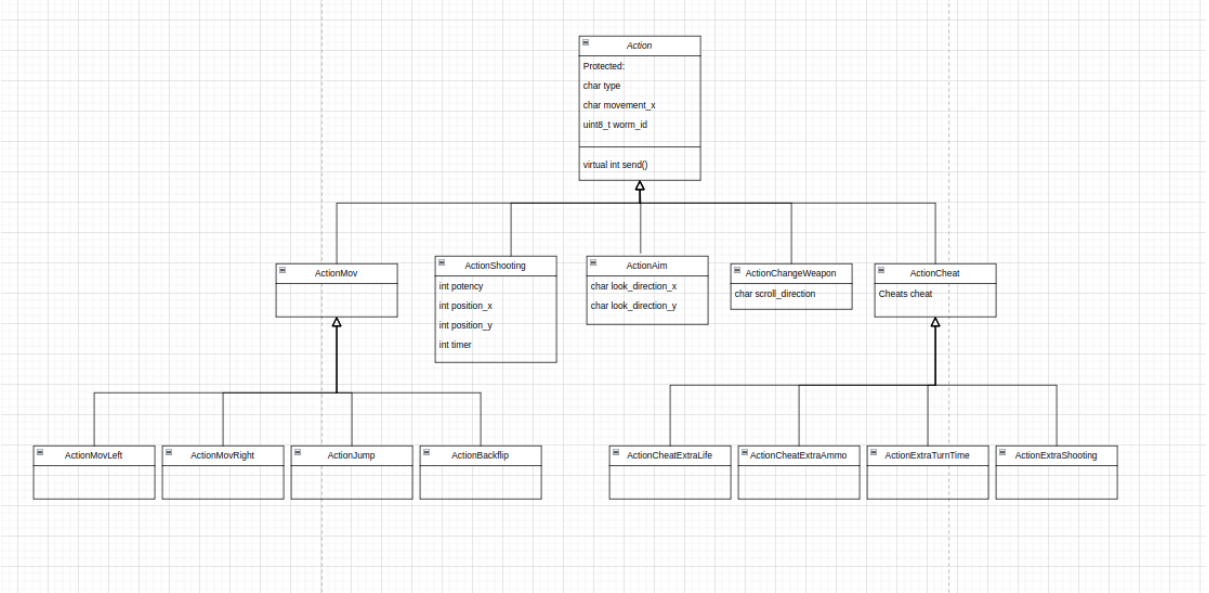
Queues

It is important to give knowledge about how the threads communicate between each other (Match and clients in the match, server side).
We use Blocking queues to achieve a proper and Race Condition free resources sharing.
To summarize how it is being done, each client will have a queue of its own, which each match will be pushing data to it and only the match will be the one pushing to those queues, the clients will only pop from their queues. And on the other side, each match will have its own queue which will only pop data from it, the ones pushing to that queue will be the clients and only the clients. Same concept goes for the client side.

Protocol:

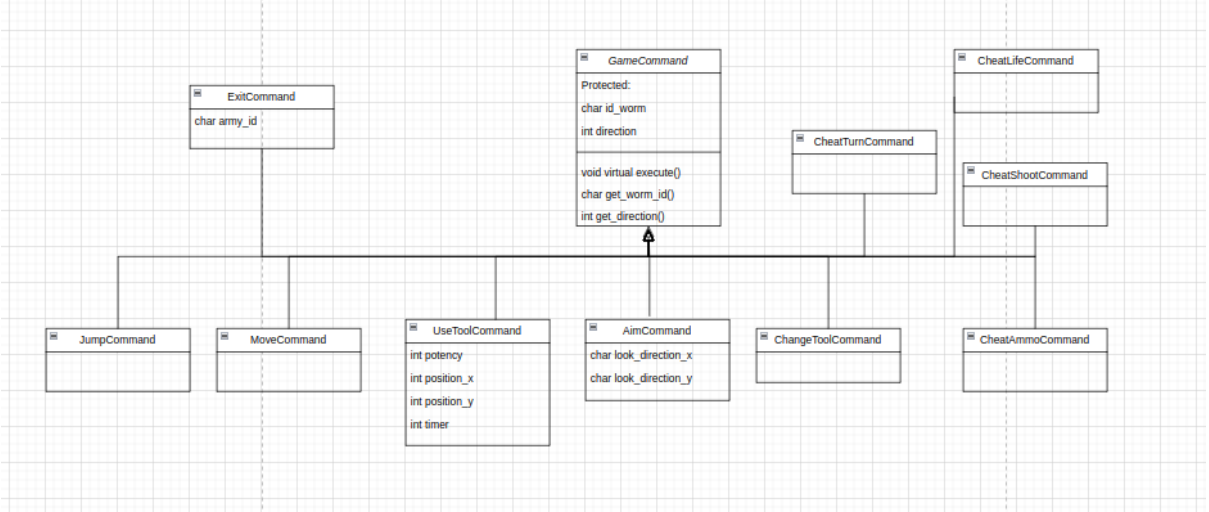
For the protocol we have done two protocol classes, one for the server and one for the client. In the lobby stage they both communicate to each other by a common class called Command. What is being sent in the protocol are the attributes of the class and in the receiver side, a new Command with those passed attributes is created and returned to the corresponding caller.
During the Match we are sending different classes depending on the case. When the server needs to send data, we send a “Snapshot” of the game which will have all the data related to the game’s state that the client needs to render. This is a class that is being shared between the client and the server, the server sends it and the client receives it. On the other hand, when the client needs to send data to the server, we are using a polymorphic class called Action (shown and explained below) and in the server we are translating each Action to a polymorphic GameCommand which will be used by the Game to interpreter what the clients wants to do. The polymorphic classes are needed because the structure of the projects (the queues to be more precise) use the parent classes and only its methods.
Finally, we have some tests that verify each Command, Snapshot, Action and GameCommand we have to certify that no data is being lost in the middle.

Action:



We can see in the image that there is a parent class called Action, which has multiple derivative subclasses which some of them also have more derivative subclasses. Notice that the amount of attributes each class has are few to none and that the only function that they use is the parent class method send, which each class overrides it to satisfy their goal.

Game Command:



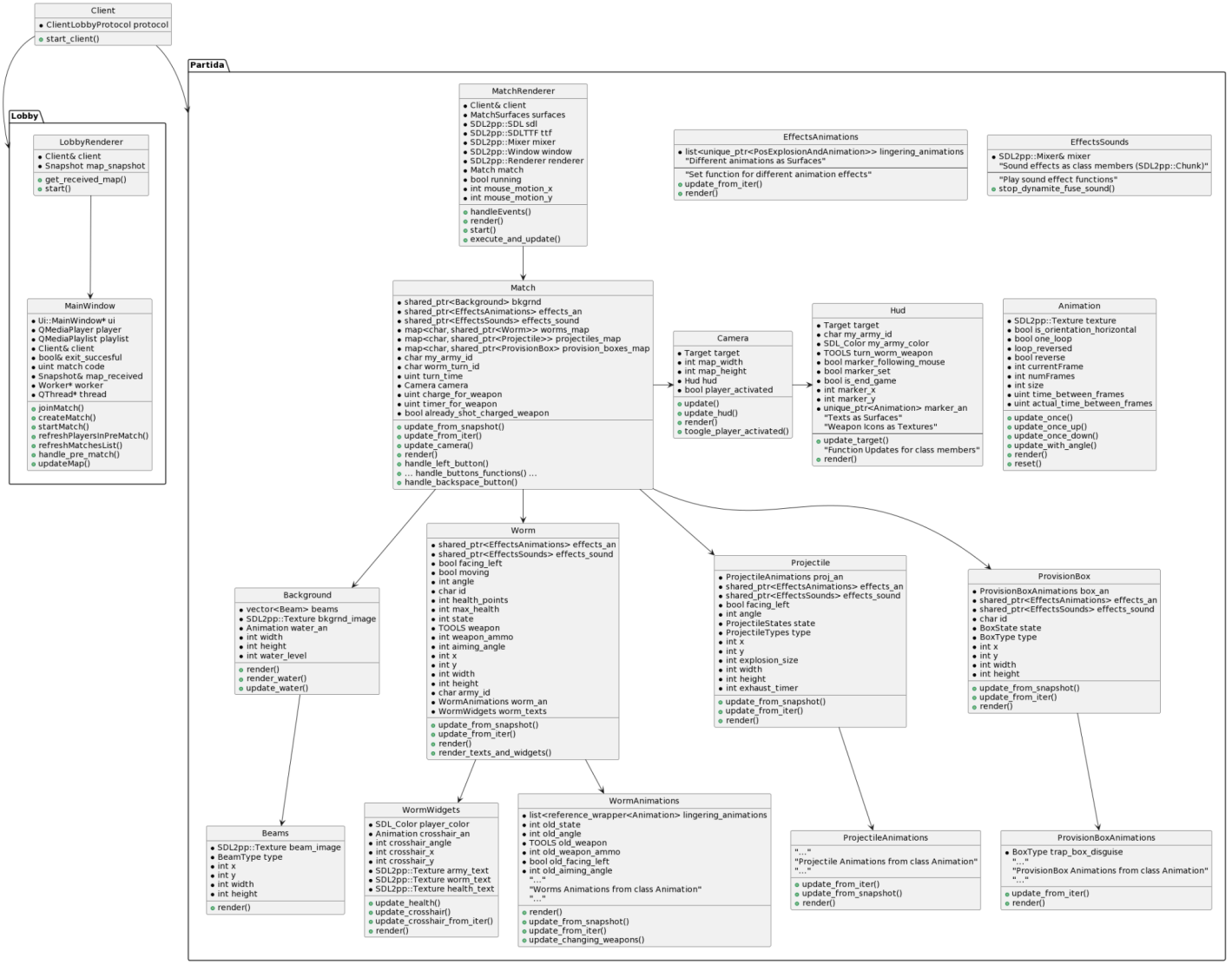
The GameCommands are very similar to the Actions as you can see. They are even more simple because each class only inherits from the master/parent class GameCommand. The majority of classes do not have their own attributes and all of the derivative classes only use the parent methods, they override as they please.

Clock:

To handle the frames and rate of the game we use a method called “If behind, drop & rest”. This method is perfectly explained here: https://book-of-gehn.github.io/articles/2019/10/23/Constant-Rate-Loop.html.
To summarize it, what it does is to “step” (func(it)) or loop normally until the rate that it is supposed to loop at is smaller than the time that took to “loop the game”. In that case, the algorithm tells you to stop, drop the loops that are behind, rest until the new loop commences and continue as normal.
We created a class called Clock that includes this algorithm which is being used both in the server (game) and client side.

Client Structure:

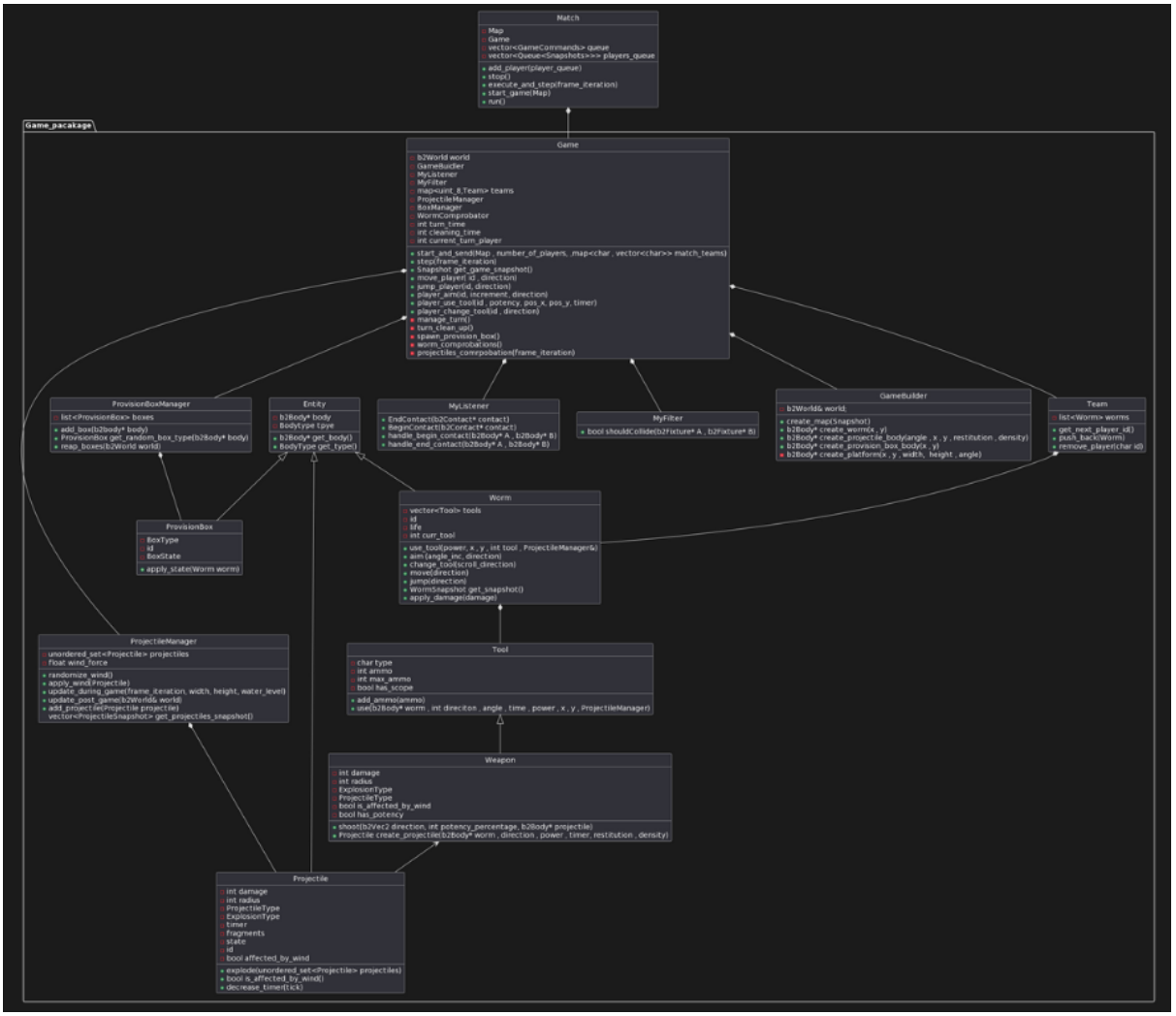
The structure is divided into two main parts. The lobby section created in QT and the match section created with SDL2. Here we attach a detailed UML diagram to see how the client operates. To summarize, the MatchRenderer has all the match information and is the one in charge of receiveing all the user inputs to then send to the server and also receive the snapshots from the server. The match has access to all the sprites, sounds and animations but also has access to the camera and the entities (worms, beams, boxes) of the world. The lobby section as seen in the diagram operates differently than the match itself. It only has a main window with all the information that it needs, and handles the I/O itself.



uml link: [uml]

Game structure:

When a new match is created a new game instance is initialized but it doesn't start running until the match starts. This game class holds the information for all the game entities that have a body in box2D. It's also the one responsible for updating the game logic using the engine of box2D each frame iteration. There are also the gameCommands mentioned previously as well as the implementation of each tool that a worm can use in the game. Finally there are a listener and a filter in which collisions are handled with the box2D collision and callback system. Here's a diagram of the structure itself for further understanding.



uml link (recommended): [uml]