Project Report: *RustMP*

Team Members:
Kostiantyn Cherniakov – `https://github.com/NotGeorgeLucas`
Veronika Bei – `https://github.com/VeronikaBei`

May 14, 2025

Source Code

The source code for this project is available on GitHub:
`https://github.com/NotGeorgeLucas/RustMP`

1 Introduction

This is a project aiming to recreate features of a local network multiplayer game, such as synchronization and RPC functionality.

2 Requirements

The basic requirements for the project are:

- The game should include basic synchronization of movement and animations
- The game should support multiple characters
- The game should support RPC functionality for synchronization of actions across game clients

3 Design Choices

- UDP protocol was chosen for fast communications between game clients. Some steps were taken to ensure the transfer of important data.
- Contrary to the concept, due to graphical decisions, the game was changed to side view.
- Macroquad was chosen as the game engine for its relative simplicity which was crucial in integrating certain network aspects

4 Dependencies

This project uses several crates to support graphics, UI, serialization, and architecture. The key dependencies are:

- `macroquad` – a simple and fast game framework for 2D games, used for rendering and input.
- `macroquad-tiled` – loader for Tiled maps, enabling integration with ‘.tmx‘ files for level design.
- `macroquad-platformer` – used for platformer-specific utilities such as collision handling and entity control.

- `bincode` – binary serialization for saving/loading game state efficiently.
- `eframe` and `egui` – for building interactive graphical interfaces, in the launcher.
- `serde` and `serde_json` – serialization/deserialization for configuration files and messages.
- `once_cell` – provides single-assignment statics and lazy initialization patterns.

The project includes two binaries:

- `rust_mp`: the main application or launcher.
- `game_main`: the actual game runtime with the main game loop.

5 Evaluation

Implementing this project in Rust provided valuable insights into both the strengths and challenges of using a systems programming language for game development and tooling.

One key realization was how much interpreted languages like Python and GDScript simplify concurrency and multithreading. This project's idea was inspired by previous experience with those languages, where setting up communication ensuring data transfer across threads was much simpler and had allowed for projects that were simpler in function but not as time efficient as with Rust. In those environments, spawning and managing threads often requires very little boilerplate, and the language/runtime typically abstracts away many of the low-level details, including Mutex access or abstractions, allowing for simpler implementation of RPC. In contrast, Rust's strict ownership model and emphasis on safety make multithreading much more explicit and verbose. While this ensures correctness and eliminates whole classes of bugs, it also significantly increases development time and cognitive load, especially for more dynamic or exploratory workflows and with the context of low knowledge of the language, ecosystem as well as some technical details.

Beyond threading, working in the Rust ecosystem has been both rewarding and challenging. The compiler is extremely helpful in guiding correct code but can be unforgiving, especially when dealing with lifetimes and certain borrow scenarios. Dependency management via `cargo` is excellent, and the broader ecosystem has mature libraries for many needs. However, documentation quality and examples can be uneven across some libraries. This was also one of the reasons for choosing Macroquad over Bevy, for example.

Overall, the project helped deepen our understanding of low-level programming in advanced ecosystems, put a lot of things into new context and allowed us to personally work with conecpts such as threads and Mutexes, which we had not been able to do within our studies on a low enough level before. Rust is a powerful tool, but it demands discipline and a deeper understanding of architecture compared to more permissive languages such as Python, but makes up for that in powerful tools compared to more low-level languages such as C.