Raylib TRON – Project Development Report

Introduction

The Raylib TRON project is a modern recreation of the classic TRON light cycle game, implemented using the C programming language and the Raylib graphics library. The central concept revolves around two players controlling light cycles that leave trails as they move. The objective is to outmaneuver the opponent without colliding with any trails, including one's own. Inspired by the retro aesthetic and competitive gameplay of the original, the development aimed to deliver a fast-paced, visually engaging, and responsive game experience.

Documentation and User Stories

From the outset, we ensured that our documentation supported accessibility and clarity for all potential users. The repository includes detailed installation instructions for both Windows and Linux, considering differences in how Raylib is set up across systems. For example, Linux users are guided to use make via Makefiles, while Windows users are provided with a .bat script for compilation. The documentation further explains core gameplay mechanics such as controls—arrow keys for Player 1 and WASD for Player 2—as well as the game's start and restart flow triggered via the spacebar and R key, respectively. These steps reflect our commitment to user-centered design and clearly defined user stories, such as "As a player, I want to see a prompt before the game starts," and "As a player, I want to restart after losing."

Evolution of the Project Concept

Initially, the project was scoped to deliver a basic two-player interaction with simple rectangular sprites and minimal logic for trail rendering. However, as development progressed, we identified opportunities to enhance the experience. We decided to redesign the players as rotated triangles instead of rectangles to better simulate dynamic movement and to distinguish the game visually. Furthermore, we implemented a custom trail-saving system using arrays of Vector2 coordinates, which allowed us to render real-time trails efficiently and check for collisions manually.

One major deviation from our original concept occurred when we realized that Raylib's built-in collision detection was insufficient for our needs, especially when dealing with triangle-based movement and irregular trail geometry. This led us to design a manual collision checker that compares the tip of one player's triangle with the saved trail points of the other. Although this solution introduced new challenges—such as managing memory efficiently and avoiding frame drops—it significantly improved the responsiveness and realism of the game.

Development Process and Methodology

Our development process was originally modeled on agile methodology, including weekly sprints and iterative feature delivery. However, we encountered several difficulties maintaining a strict schedule due to inconsistent retrospectives and overlapping responsibilities within the team. In the early phases, task ownership was vague, which led to duplicated code and some confusion regarding feature responsibilities. As a result, we modified our approach mid-project. We adopted clearer role assignments and used GitHub Issues and Actions to track tasks and implement basic continuous integration. For example, we automated the compilation of the project on every push, which helped us catch broken builds early.

Despite not integrating unit tests due to the procedural and graphical nature of our code, we conducted manual testing rigorously. During testing, we discovered that certain trail overlaps were not being detected as collisions. This discovery prompted a redesign of the trail-checking logic using a custom loop rather than relying on collision libraries. Additionally, we introduced visual debugging aids, such as drawing colored lines and printing player states in the console, which facilitated rapid identification of issues.

Technology Stack and Design Decisions

We selected C as our programming language due to its low-level control and performance efficiency, both of which are vital for real-time games. Raylib was chosen as our graphical library for its simplicity, excellent documentation, and cross-platform support. We maintained a modular code structure by separating functionality across multiple files—main.c, controller.c, and render.c—which improved code readability and maintainability.

For instance, render.c focuses solely on graphical output, including the rotated triangle logic and trail rendering, while controller.c handles input and game state transitions.

To accommodate multiple platforms, we created a Makefile for Unix-based systems and a batch script for Windows. This cross-compatibility was tested on both Ubuntu and Windows 10 environments. By following this structure, we ensured that contributors or testers could work without friction, regardless of their OS.

Team Collaboration and Reflections

Team dynamics were a blend of proactive collaboration and on-the-fly coordination. Our team members often volunteered to troubleshoot bugs, optimize code, and polish features regardless of their initial assignments. However, we also faced challenges due to overlapping roles, especially in the earlier stages when no member was explicitly responsible for collision logic or input handling. This led to multiple versions of similar functions before we consolidated them. One example was the trail-saving logic, which initially existed in duplicate form for each player, before being unified under a clearer structure.

In retrospect, we could have benefited from structured daily stand-ups and a formal task board. These would have helped us identify blockers earlier and improve overall transparency. If given the chance to work with the same team again, I would propose implementing Trello or GitHub Projects from day one and assigning rotating roles for testing and documentation to ensure a balanced workload.

Visual Demonstration and Gameplay Features

The visual output of the game successfully captures the spirit of the TRON universe. The triangular player models, rendered in red and blue, create sharp, recognizable silhouettes that contrast clearly against the black background. Each movement leaves behind a persistent trail rendered in real time, creating a dynamic playfield that evolves as the match progresses. The game begins with a prompt”Press space to start!"—and concludes with a restart option if a player loses by crashing into any trail. These transitions, though simple, provide a complete and coherent game loop. Demonstrative screenshots and a

gameplay recording are included in the repository's README file to showcase the primary features.

Future Improvements and Expansion

Though our current version is a strong proof of concept, there are several areas we plan to improve in future iterations. First, we aim to implement network-based multiplayer using a lightweight library like ENet, which would allow players to connect remotely. Second, we want to integrate an AI-controlled opponent that can simulate basic evasive behavior for single-player mode. Third, we will redesign the UI to support customizable controls, player color selection, and better accessibility options such as high-contrast modes or text-to-speech prompts. These features will broaden our user base and provide a more inclusive experience.