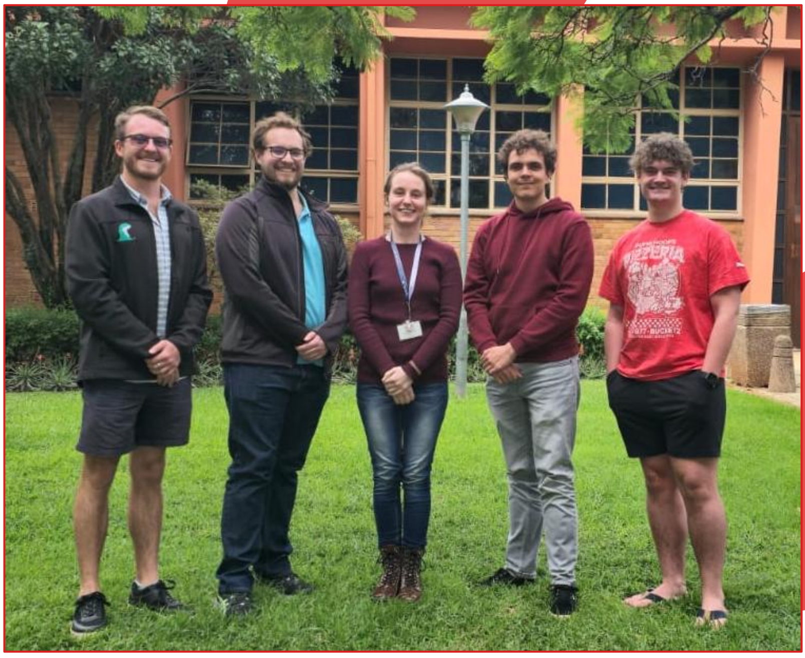
SuperLap Racing Line Optimization System

EPI-USE



Quintessential

Amber Ann Werner [u21457752]

Milan Kruger [u04948123]

Qwinton Knocklein [u21669849]

Sean van der Merwe [u22583387]

Simon van der Merwe [u04576617]

Specifications and Standards

Coding Standards

Naming Conventions

File Names: A mix of `PascalCase` and `camelCase` is used.

Folder Names: Generally, use `PascalCase`. However, some folders follow lowercase naming conventions for system compatibility – for example, the docs folder is lowercase to enable GitHub Pages hosting.

Class Names: All class names follow `PascalCase` for clarity and consistency.

Special Cases:

- API and `RacelineOptimizer` follow `PascalCase` as they are core modules.
- `image_processing` uses `snake_case` to align with external library conventions and improve readability in multi-word module names.

File and Folder Structure

The project is organized into modular folders to separate concerns and support scalable development. Below is the structure of the repository:

Repository Root

- `Backend/` – Contains core backend components including:
  - `API/`: Handles external communication (e.g: Unity and MongoDB).
  - `HelperScripts/`: This file contains the `AlignPlayerLine`, `CreateMasksFromJson` and `TrainingDataGenerator` scripts that our system needed in addition to function correctly.
  - `ImageProcessing/`: Processes images received from Unity, converting them into usable track data.

        - `CNN/`: This one of our AI subsystems used to detect the track images and extract the track from the image itself.
    - `MotoGPTelemetry/`: This part of our wow factor. We use to track the user on a given MotoGP track and export the data to our system.
    - `RacelineOptimizer/`: Uses processed images to determine the optimal raceline.
- `docs/` – Stores documentation and static site files (used for GitHub Pages hosting). Subdirectories include:
    - `css/, js/, images/, wordDocs/, and index.html.`
- `scripts/ – Contains setup scripts and developer utilities:`
    - `setup-act.sh`: Installs `nektos/act` to run GitHub Actions locally.
    - `ACT.md`: Documentation for using local workflows.
- `Unity/` – The front-end Unity project used for rendering and interaction.
- `Website/` – Web-related files for convenience and deployment purposes.
- `README.md` – Project overview and general instructions.

Docker and Testing

- Each service folder (except Unity) contains its own `Dockerfile`.
- A global `docker-compose.yml` file is located in the project root.
- `.dockerignore` files are placed in each relevant directory.
- Testing directories (e.g: `tests_integration/, e2e/, unit/`) are found within service folders for modular test execution.

Formatting Standards

- **Indentation**: Tabs are used for indentation across the project for consistency.
- **Line Length**: No strict limit has been enforced, but lines are generally kept concise for readability.

- **Braces**: Opening braces are placed on the same line as control statements (e.g: `if (...) {}`, with the block content starting on the next line.
- **Spacing**: Standard spacing is followed, including spacing around operators and within brackets (e.g: `{ int = 0; }`).
- **Comments**:
  - Both single-line (`//`) and block (`/* */`) comments are used.
  - Single-line comments are used for short explanations, while block comments provide contextual or functional documentation.
- **Docstrings**: No specific docstring format is used in this project.

Coding Practices

- **Naming**: Functions and files are named to clearly reflect their purpose or output. Descriptive naming is prioritized over name length limitations.
- **Structure**: Code is kept modular and functions are designed to handle specific tasks where possible.
- **General Practices**: Standard coding practices are followed, including avoiding deeply nested logic, keeping code readable, and minimizing redundancy.

Version Control Guidelines

Commit Messages: All commit messages must be clear, descriptive, and explain what the commit does.

Branching Strategy:

The primary branches are:

- `main`: Stable production-ready code.
- `dev`: Integration branch for completed features.

Feature branches are categorized by function:

- `UI/`: Frontend and website-related work
- `Backend/`: Backend processing and API
- `CICD/`: Continuous Integration and Deployment scripts/tests

Branch naming follows a consistent format:

- Example: Backend-PSA-start, UI-Web-LandingPage

Commit Frequency: Developers are expected to make a minimum of 10 commits per week, ideally after every significant update on their feature branch.

Pull Requests:

- Pull requests must be submitted once a branch feature is complete.
- Each PR must be reviewed by **at least two team members** before being merged.
- Branches are merged progressively: `feature → category (e.g: UI) → dev → main`.
- Direct commits to main are not allowed.

CI/CD: The main branch runs the CI/CD pipelines to ensure stability.

Tools and Configurations

CI/CD: A basic CI/CD setup is implemented, currently running automated tests from the various tests folders.

Docker:

- Each backend component (API, ImageProcessor, RacelineOptimizer) has its own `Dockerfile`.
- A root-level `docker-compose.yml` is used to orchestrate the containers.

Scripts: Utility scripts are stored in the scripts/ directory for local tool setup and CI/CD helpers.

Linters/Formatters: Not strictly enforced, but individual team members may use personal formatting tools suited to their language. There is also currently linting present in our C# code.

Language/ Framework-Specific Conventions

Unity and RacelineOptimizer: Written in C#. Follows typical Unity/C# naming and structure conventions.

API: Implemented in Node.js using JavaScript/TypeScript.

Image Processor: Written in Python, using common Pythonic conventions (e.g: `snake_case`, modular scripts).

Website: Built with standard HTML, CSS, and JavaScript, organized within the docs/ folder for GitHub Pages compatibility.