Switchback Rails

CS-1002: Programming Fundamentals (AI&DS)

November 11, 2025

Due Date: 30 Nov 2025 **Time: 10:00 PM**

Good news: The course project will be completed in **group of two**, and cross-section groups are allowed within the AI & DS department.
Please form your pairs. We will share a sheet for group registration along with the registration deadline. After the deadline, **no new groups** will be registered and **no changes** in group members will be allowed.
Note: The remaining instructions related to implementation and submission are provided at the end of this file. Late submissions will not be accepted.
Advice: Please start working early to ensure timely submission. Good luck!

Learning Outcomes

- CLO-1: Apply logic constructs (loops, conditionals) to analyze and solve small problems.
- CLO-2: Use functions, arrays, and pointers to design a modular C++ solution.
- CLO-3: Develop a **deterministic** project that produces reproducible outputs.

1 Introduction

Design and implement a small railway simulation game in C++ that demonstrates how **structured programming** can control objects that move, wait, and interact using only fundamental programming concepts. *Switchback Rails* is a mini-world of tracks on a 2D grid where trains move one tile per time step (**tick**) and must reach their destination stations without collisions. The grid is rendered using sprite images (32x32 or 48x48 pixels per tile) loaded from the `Sprites/` folder. The simulation uses a **distance-based priority system** for collision resolution (trains further from their destination have higher priority and proceed, while closer trains wait).

Some tracks are **switches** (A–Z) that can route trains differently. Each switch flips its direction only after a certain number of entries (**K-value**) and always flips **after movement** for that tick (a **post-routing flip**). This ensures deterministic behavior where switch state changes do not affect trains that have already computed their routes.

Players configure switch states, place limited **safety buffers** (`=`) to slow trains, and start the simulation. The system must log each event, print the grid state to the terminal at each tick, and verify that all trains reach correct stations without crashing.

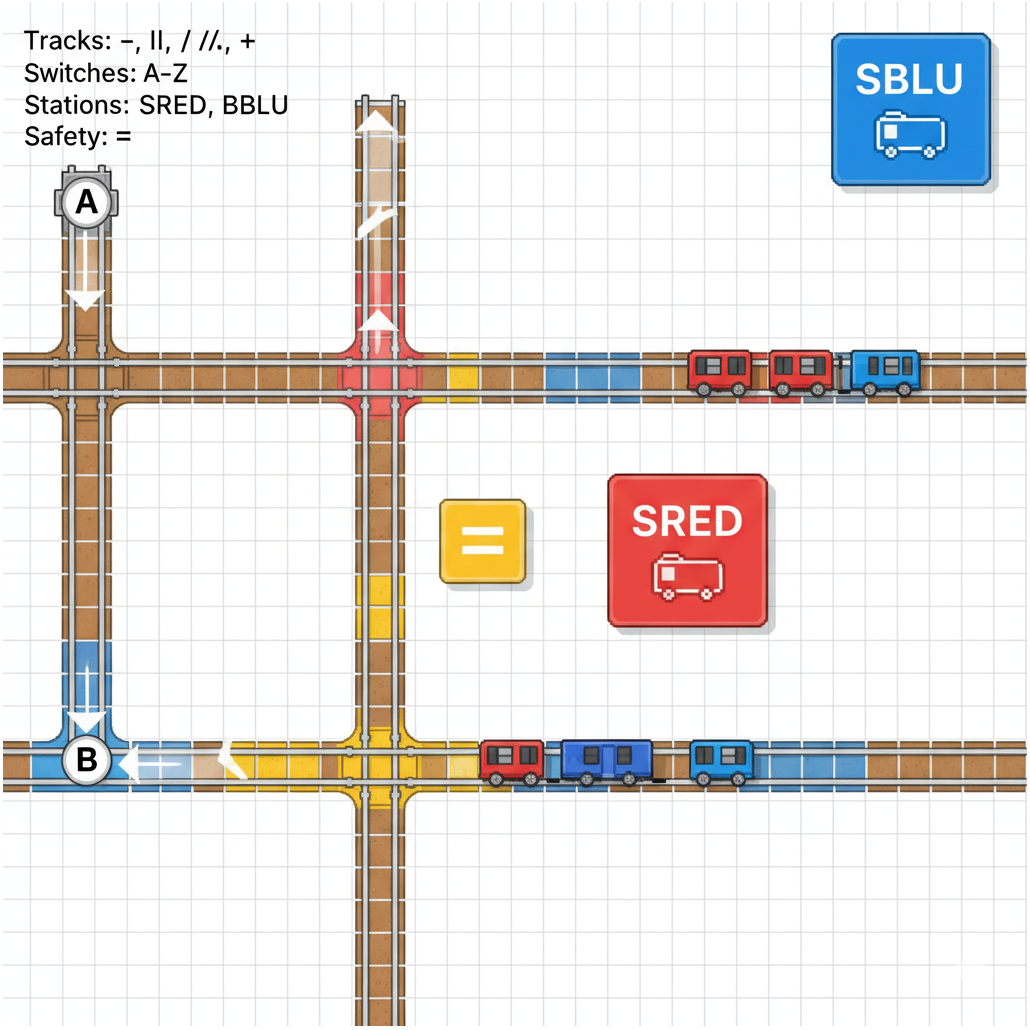

Figure 1: Game overview of the 2D railway grid: tracks, spawn points (S), destination points (D), switch tiles (A, B), safety tile (=), and crossings (+).

2 Core Game Rules

Track and Train Interaction

- **Tiles**: `-` (horizontal track), `|` (vertical track), `/` & `\` (curves), `+` (crossing), `S` (spawn point), `D` (destination point), `A-Z` (switch groups), `=` (safety buffer).
- **Grid Rendering**: Each tile is rendered as a sprite image (32x32 or 48x48 pixels) loaded from the `Sprites/` folder. Trains are represented by colored circles or sprites.
- **Train Movement**: Trains move one tile per tick in their current direction. Direction changes occur when entering curves, switches, or crossings.
- **Safety Buffers**: Entering `=` adds a one-tick delay before the train can move again (train stays on `=` during the delay).
- **Terminal Output**: At each tick, the complete grid state must be printed to the terminal, showing all tiles, train positions, and switch states in ASCII format.

Switch Logic

- All tiles with the same letter share a *single* state and counters.

- **Entering a switch** increments a counter keyed by the entry direction (PER_DIR) or a global counter (GLOBAL).
- When a counter reaches its **K-value**, the switch is *queued* to flip; the flip applies *after* movement in the same tick.
- This prevents a train from changing its own route mid-tick, ensuring deterministic behavior.

Switch Modes & Encoding

In `SWITCHES:`, each line is:

```
<Letter> <Mode> <Init> <K_UP> <K_RIGHT> <K_DOWN> <K_LEFT> <State0> <State1>
```

Mode: `PER_DIR` uses the four direction-specific K's; `GLOBAL` uses the first K for all directions (the remaining K fields may repeat for readability).
Init: initial state index (`0` ⇒ `State0`, `1` ⇒ `State1`).
State labels: human-readable names (e.g., `LEFT/RIGHT` or `STRAIGHT/TURN`).
Example:

```
A  PER_DIR  0  2 1 3 1  LEFT RIGHT
B  GLOBAL   1  4 4 4 4  STRAIGHT TURN
```

3 Tick Timing

A **tick** is a fixed step (e.g., 0.1s). The engine performs these phases *strictly* in order:
1. **Spawn**: Allign trains scheduled for this tick (see spawn rules below).
2. **Route Determination**: each train computes its next tile from current heading & the switch's *current* state.
3. **Counter Update**: entering a switch increments its counter(s).
4. **Flip Queue**: switches at K are flagged to toggle.
5. **Movement**: all trains advance simultaneously; detect invalid moves/collisions using distance-based priority.
6. **Arrivals**: record station arrivals.
7. **Terminal Output**: print the complete grid state to terminal showing all tiles and train positions.

4 Determinism Guarantees

Spawn Rules

- If a spawn tile is *occupied* at its scheduled tick, the train waits and retries next tick.
- If the spawn tile is free but its immediate next tile is invalid (off-map/blocked), the train still spawns and will crash only if it attempts an invalid move on the movement phase.
- Scheduling offsets are pseudo-random but seeded by `SEED` from the level file; identical inputs ⇒ identical schedules.

Collision & Conflict Resolution with Distance-Based Priority

The collision system uses a **distance-based priority** mechanism to resolve conflicts intelligently:

- **Distance Calculation**: For each train, calculate the Manhattan distance (sum of absolute differences in x and y coordinates) from the train's current position to its assigned destination point.
- **Priority Rule**: The train with the **higher distance** to its destination gets priority and proceeds, while trains with **lower distance** wait for the next tick.
- **Same-destination collision**: If multiple trains target the same tile in a tick:
  - Calculate distance to destination for each train.
  - Train with highest distance moves; others wait.
  - If distances are equal, both trains crash.
- **Head-on swap collision**: If two trains are swapping tiles in one tick:
  - Calculate distance to destination for each train.
  - Train with highest distance moves; the other waits.
  - If distances are equal, both trains crash.
- **Crossing '+' collision**: If multiple trains enter the same '+' tile simultaneously:
  - Calculate distance to destination for each train.
  - Train with highest distance moves; others wait.
  - If distances are equal, all involved trains crash.
- This priority system ensures trains closer to their destination yield to trains that have further to travel, creating more efficient traffic flow and reducing unnecessary crashes.

Switch Semantics

- The flip occurs *after* trains move; the flipping train is unaffected this tick.
- Counters reset on flip (recommended) to avoid immediate re-flip loops.

5 Gameplay Features

1) Signal Lights

Each switch tile shows a light for safety visualization:

- `Green`: next tile free along planned route.
- `Yellow`: train within two tiles ahead.
- `Red`: next tile blocked/occupied or would collide this tick.

2) Dynamic Train Scheduling

Trains spawn at scheduled ticks based on the level file configuration. Each train is assigned a spawn tick, position, direction, and color index.

3) Weather Effects

- **Normal**: constant speed (1 tile/tick).
- **Rain**: occasional slowdowns (extra wait tick after n moves).
- **Fog**: delay in *displayed* signals (logic remains correct).

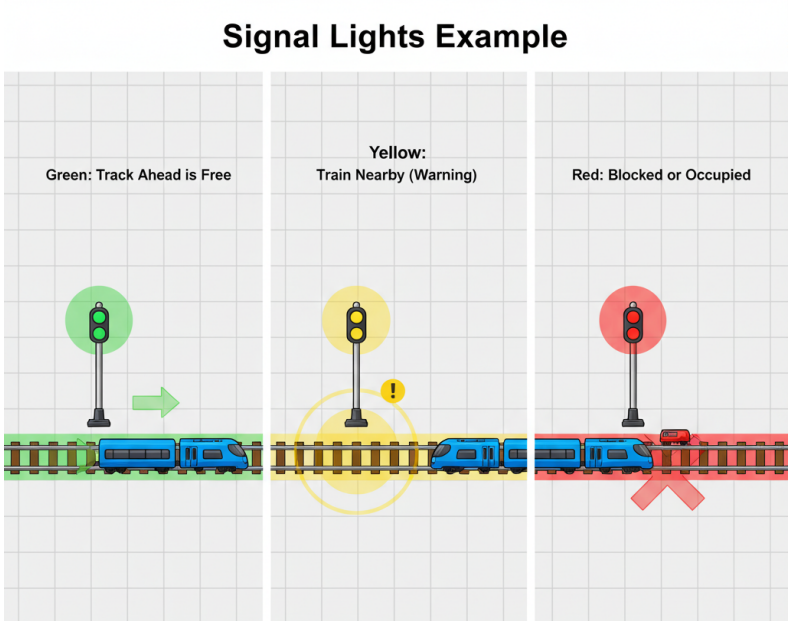

Figure 2: Signal lights: green (safe), yellow (warning), red (blocked).

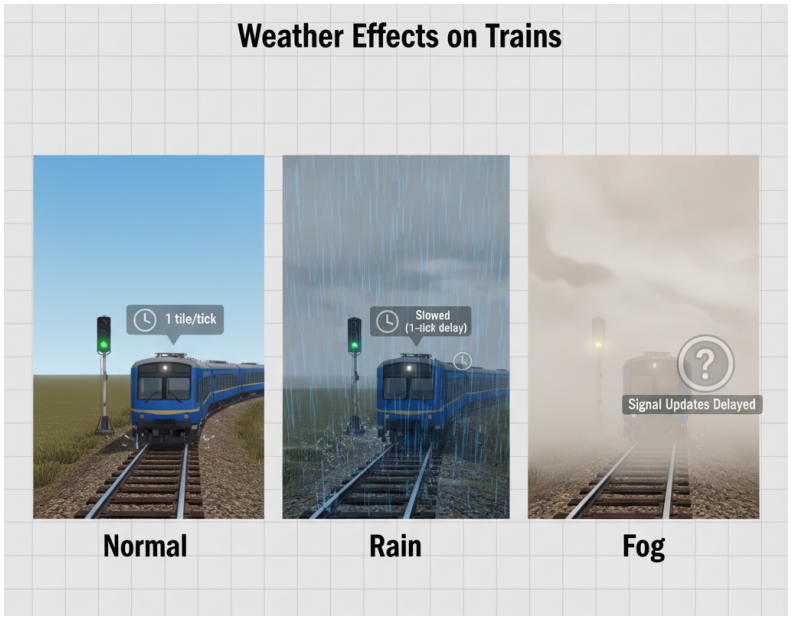

Figure 3: Weather effects: rain and fog impacting speed or signal latency.

4) Emergency Halt

Triggering on a switch group pauses trains in its 3×3 neighborhood for 3 ticks; counters and states continue to evolve normally.

5) Metrics and Evidence

- **Throughput**: trains delivered per 100 ticks.
- **Average Wait**: mean idle ticks.
- **Signal Violations**: entries against red.
- **Energy Efficiency**: (trains $\times$ ticks)/buffers.

- **Switch Flips**: total number of switch state changes.

6 File Input/Output

Level File (.lvl)

Defines grid, switches, stations, limits.

```
NAME: INTRO
ROWS: 12
COLS: 18
SEED: 4711
WEATHER: NORMAL
MAP:
..A---+---B.......
..|...|...|.......
..|...S...|.......
..|...|...|.......
..B---+---A.......
      D
SWITCHES:
A  PER_DIR  0  2 1 3 1  LEFT RIGHT
B  GLOBAL   1  4 4 4 4  STRAIGHT TURN
TRAINS:
0 2 2 1 0
5 2 5 1 1
```

Numbers after `PER_DIR/GLOBAL` are K-values; see "Switch Modes" above. The `TRAINS:` section specifies spawn schedule: `<tick> <x> <y> <direction> <color>`.

Output Files

- `trace.csv`: `Tick,TrainID,X,Y,Direction,State` - Complete train movement history.
- `switches.csv`: `Tick,Switch,Mode,State` - Switch state changes per tick.
- `signals.csv`: `Tick,Switch,Signal` - Signal light states (GREEN/YELLOW/RED).
- `metrics.txt`: Final statistics including trains delivered, crashed, average wait time, energy used, switch flips, and efficiency metrics.

7 How to Run the Project

Prerequisites

Before running the project, ensure you have installed all required libraries:

1. **Install Libraries**: Run the installation script to install GCC, SFML, and all dependencies:

   ```
   bash libraries.sh
   ```

This script will install:

- GCC and G++ compilers
- SFML development libraries
- Build tools (make, cmake)
- Additional dependencies (OpenGL, X11, audio libraries, fonts)

2. **Clean Previous Build**: Remove any existing build artifacts:

```
make clean
```

3. **Build the Project**: Compile all source files:

```
make
```

This will create the executable `switchback_rails`.

4. **Run a Level**: Execute the simulation with a level file:

```
./switchback_rails data/levels/easy_level.lvl
./switchback_rails data/levels/medium_level.lvl
./switchback_rails data/levels/hard_level.lvl
./switchback_rails data/levels/complex_network.lvl
```

Terminal Output Requirement

At each simulation tick, the program must print the complete grid state to the terminal in ASCII format. The output should show:

- All track tiles (`-`, `|`, `/`, `\`, `+`)
- Spawn points (`S`) and destination points (`D`)
- Switch tiles (`A-Z`)
- Safety buffers (`=`)
- Train positions (represented by train ID numbers or symbols)
- Current tick number

Example terminal output format:

```
Tick: 5
  S===A===+===B===D
          |   |
          |   |
  S===C===+===D===D
          |   |
          |   |
          D   D
Train 0 at (2,2) moving RIGHT
Train 1 at (5,2) moving RIGHT
```

8 Using Sprites in the Game (for Bonus)

Sprite Requirements

The game must use sprite images from the `Sprites/` folder to render the grid visually. Each tile in the grid should be rendered using a sprite image.

Sprite Specifications

- **Tile Size**: Each grid tile can be rendered as either **32x32 pixels** or **48x48 pixels**. Choose one size consistently throughout the game.
- **Sprite Files**: The `Sprites/` folder contains PNG image files (e.g., `1.png`, `2.png`, `3.png`, `4.png`, `5.png`) that represent different track types, trains, switches, and game elements.
- **Loading Sprites**: Use SFML's `sf::Texture` and `sf::Sprite` classes to load and display sprite images:
  - Load each sprite image file using `texture.loadFromFile("Sprites/filename.png")`
  - Create sprite objects and set their positions based on grid coordinates
  - Scale sprites appropriately to match the chosen tile size (32x32 or 48x48)
- **Sprite Mapping**: Map different tile types to appropriate sprite images:
  - Horizontal tracks (`-`) → appropriate sprite
  - Vertical tracks (`|`) → appropriate sprite
  - Curves (`/`, `\`) → appropriate sprite
  - Crossings (`+`) → appropriate sprite
  - Switches (`A-Z`) → switch sprite with state indication
  - Spawn points (`S`) → spawn point sprite
  - Destination points (`D`) → destination sprite
  - Safety buffers (`=`) → safety buffer sprite
  - Trains → train sprite (colored based on train color index)
- **Rendering**: Draw sprites in the correct order (background tiles first, then trains and overlays) to ensure proper visual layering.

9 SFML User Interface (Visual Layer Only)

Controls: Left-click = place/remove safety tile.
Right-click = toggle switch state.
Space = run/pause.
Mouse wheel = zoom.
Middle drag = pan.
. (period) = step one tick forward.
ESC = exit and save metrics.

10 Glossary

Grid: Rows and columns forming the map, rendered using sprites (32x32 or 48x48 pixels per tile).
Tile: One square cell in the grid, represented by a sprite image.

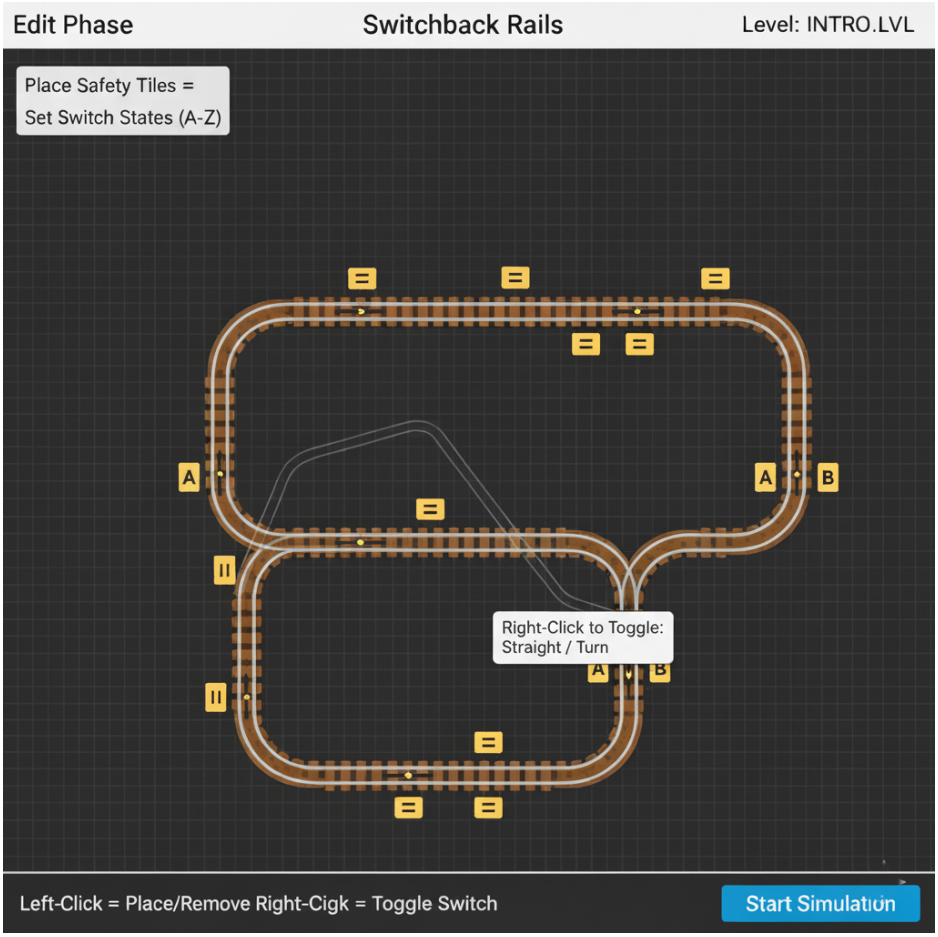

Figure 4: Edit phase: place safety tiles; set initial switch states.

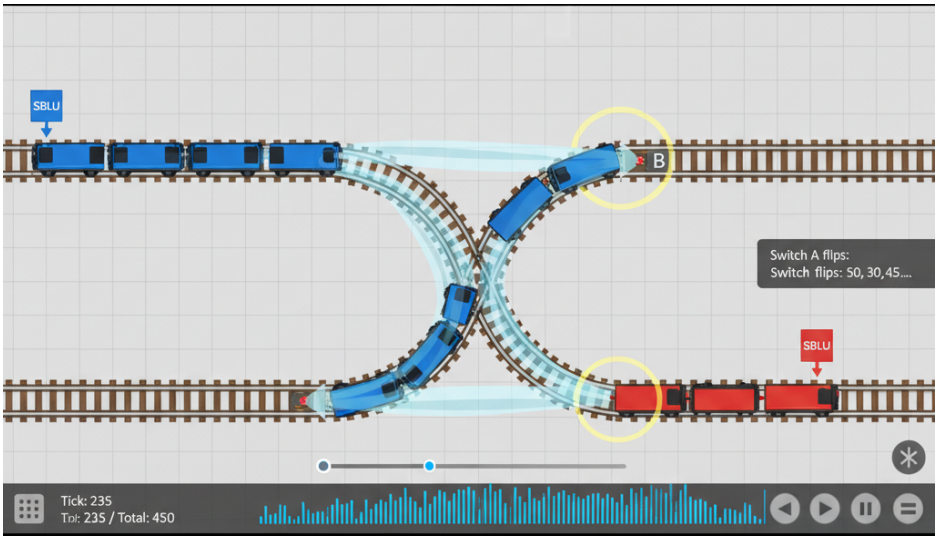

Figure 5: Review phase: scrub through ticks; blue bars mark flips.

Switch: Track that can flip between two routes based on entry count (K-value).
K-value: Number of train entries needed before the switch flips.
Deferred flip: Flip happens after trains move this tick, ensuring deterministic routing.
Safety Tile (=): Slows the train by one tick when entered.

Tick: One unit of time when all trains move together and grid state is printed to terminal.
Signal Light: Shows if a path is free (green), has warning (yellow), or blocked (red).
Distance-based Priority: Collision resolution system where trains further from destination have higher priority.
Manhattan Distance: Sum of absolute differences in x and y coordinates between two points.
Deterministic: Same inputs (level file, seed) $\Rightarrow$ same outputs every run.

11 Submission Instructions

Project Skeleton Code repository: github.com/AdilMajeed/Switchback_Rails

1. **Fork & Name:** Fork the template to your account and rename to `PF-Project-SwitchbackRails-{RollNumber}`.
2. **Required structure (keep as-is):**
   - `/core` – console simulation and logic (no SFML includes).
   - `/sfml` – SFML UI that calls into core headers.
   - `/data` – level/config files; `/out` – generated logs (commit at least one sample run).
   - `/Sprites` – sprite image files (PNG format, 32x32 or 48x48 pixels).
   - `/docs` – `Edit.png`, `Simulate.png` (screenshots from your build).
3. **Commit:** After making any changes, commit your code regularly. Keep in mind that your code should remain private until the demo is conducted.
4. **README:** Include minimal build/run steps:
   - How to install libraries (`bash libraries.sh`),
   - How to build (`make clean`, `make`),
   - How to run a sample level (`./switchback_rails data/levels/level.lvl`),
   - Where outputs are written (`out/` directory),
   - How to use sprites from `Sprites/` folder.
5. **Presentation File:** Prepare a PowerPoint presentation named `SwitchbackRails-{RollNo}.pptx` to be presented on the day of demos. The slides should clearly explain:
   - overall program structure and design decisions,
   - demonstration of determinism (show identical logs on multiple runs),
   - distance-based collision priority system implementation,
   - sprite rendering system (32x32 or 48x48 pixels),
   - terminal output of grid state,
   - key results and performance metrics with screenshots.

   The PPT will be evaluated during the viva/demonstration.
6. **Submission:** Submit a single compressed file `SwitchbackRails-{RollNo}.zip` (containing your full repository including code, data, output folders, and Sprites folder) **and** share your public GitHub repository link on **Google Classroom (GCR)** before the deadline.
7. **Demo/Viva:** Reproduce logs live from your tagged commit; be ready to answer questions on tick order, flips, distance-based collision resolution, sprite rendering, terminal output, and I/O.

12 Implementation Notes

- Use `g++` (C++11 or newer). Link `sfml-graphics`, `sfml-window`, `sfml-system` in the UI target.
- The core must compile *without* SFML and be testable from a console `main.cpp`.
- Keep modules small: `moveTrains()`, `updateCounters()`, `queueFlips()`, `applyFlips()`, `detectCollisions()`, `calculateDistance()`, `writeLogs()`, `printGrid()`.
- Use relative paths (`./data/`, `./out/`, `./Sprites/`); seed-based jitter must be deterministic.
- Validate indices and file reads; prefer fixed-size arrays from limits.
- Implement `printGrid()` function to output complete grid state to terminal at each tick.

- Load and render sprites from `Sprites/` folder using SFML, choosing either 32x32 or 48x48 pixel tile size.
- Use only Programming Fundamentals concepts.

13 Academic Integrity & Reproducibility

- **Original Work Only:** Sharing or copying code (including AI-generated code) is forbidden. **Plagiarism results in an F grade for the course.**
- **Explain Your Code:** In demo/viva, you must explain your own code.
- **Reproducible Results:** Given the same level file and seed on your tagged commit, outputs must be identical.
- **Similarity Checks:** Repos with substantially similar structure/naming/logic will be flagged; all involved parties face penalties.

14 Grading Rubric

Criterion	Excellent (10–9)	Good (8–7)	Average / Poor (6–4 / 3–0)	Weight
Correctness & Determinism	All rules correctly implemented including distance-based collision priority; identical outputs across multiple runs; passes all public and hidden tests; grid state printed to terminal correctly.	Minor logic mistakes or slight timing inconsistencies; mostly correct outputs with small deviations; distance-based priority mostly working; terminal output present but incomplete.	Frequent logic errors, crashes, or non-deterministic results; fails tests or produces inconsistent logs; distance-based priority not implemented; no terminal output.	30%
Implementation Quality	Code is modular, readable, and memory-safe; functions well-structured using only arrays, loops, conditionals, pointers, and file I/O; clear naming and proper use of fundamental concepts.	Mostly modular with some redundant code; occasional unsafe pointer use or unclear naming; mostly uses fundamental concepts correctly.	Monolithic or disorganized code; unsafe memory use; poor readability; uses advanced features not taught in PF; missing functions.	20%

Dynamic Features & UI	Signal lights, scheduling, weather, emergency halt, distance-based collision priority all implemented; SFML visuals with sprites (32x32 or 48x48) synchronized with tick order; terminal grid output at each tick.	Most features working but partial visual feedback or minor timing mismatches; sprites loaded but inconsistent sizing; terminal output present but incomplete.	Static or missing dynamic features; UI unsynchronized or not functional; sprites not used; no terminal output.	15%
Collision Priority System	Distance-based priority correctly implemented for all collision types (same-destination, head-on swap, crossing); Manhattan distance calculated accurately; priority resolution works as specified.	Distance-based priority implemented but minor errors in calculation or resolution; works for most collision types.	Distance-based priority not implemented or incorrectly implemented; collisions resolved incorrectly or cause crashes.	5%
Robustness & Validation	Handles boundary cases gracefully; all inputs validated; no runtime crashes or leaks; grid bounds checked; file reads validated.	Minor input validation issues; handles common cases correctly.	Frequent crashes, unhandled inputs, or invalid memory access.	10%
Testing & Evidence Files	Generates complete and correct `trace.csv`, `switches.csv`, `signals.csv`, and `metrics.txt`; terminal output matches logged data.	Produces most logs correctly with small mismatches or formatting issues; terminal output mostly matches logs.	Missing or incorrect logs; incomplete or inconsistent evidence files; terminal output missing or incorrect.	10%
Documentation & Presentation File	Comprehensive and well-structured PPT; includes system design, determinism proof, distance-based priority explanation, sprite rendering details, and metric analysis.	PPT submitted but missing one key aspect (design explanation, proof, priority system, or metrics).	PPT unclear, incomplete, or not submitted.	10%

15 Outcome Alignment: Assignment Outcomes (AOs) ⇒ CLOs ⇒ SA

Assignment Outcomes (AOs)	Mapped Course Learning Outcomes (CLOs)	Seoul Accord Attributes (SA)
AO1 – Apply logic constructs (loops, conditionals) to control tick order, switch counters, distance-based collision resolution, and grid printing.	**CLO-1** – Apply fundamental programming constructs to solve structured problems.	SA1 (Computing Knowledge), SA2 (Problem Analysis)
AO2 – Design a modular program using arrays, pointers, functions, and file I/O to implement train simulation, sprite rendering, and terminal output.	**CLO-2** – Develop modular, maintainable C++ code using PF topics.	SA3 (Design/Development of Solutions), SA4 (Modern Tool Usage)
AO3 – Produce deterministic, verifiable outputs via logs (`trace.csv`, `switches.csv`, `signals.csv`, `metrics.txt`) and terminal grid state output.	**CLO-3** – Implement deterministic and testable solutions with evidence.	SA3 (Design/Evaluation), SA4 (Tool Usage), SA6 (Communication)
AO4 – Demonstrate originality and responsible practice (no plagiarism; reproducibility, proper attribution).	**CLO-3** – Reflect on software ethics and reproducibility.	SA7 (Professionalism & Society), SA8 (Ethics), SA9 (Lifelong Learning)