Robot

1. Refactor the Robot Class into MVC Structure

1.1. Separate Model, View, and Controller

- **Model (`RobotModel`)**: Handles the logic and state of the robot.
- **View (`RobotView`)**: Responsible for rendering the robot visually.
- **Controller (`RobotController`)**: Updates the view based on the changes in model.

1.2. Extract Robot Movement Logic into `RobotModel`

- Move all robot movement calculations and event handling logic into `RobotModel`.
- Keep `RobotView` only for rendering.
- Ensure `RobotController` updates the view based on the changes in model.

---

2. Implement Time-Based Movement

2.1. Add a Clock to the Simulator

- Create a **`Clock`** class in the simulator that keeps track of the elapsed time.
- The clock should store timestamps and calculate time deltas.

2.2. Modify Robot Movement to Use Elapsed Time

- Store the last timestamp of movement updates.
- Calculate the robot's position based on:
    - The time elapsed (`delta_time`).
    - The current speed (`linear_velocity` and `angular_velocity`).
    - The distance traveled = speed × elapsed time.

2.3. Ensure Continuous Movement Based on Time

- Instead of moving every simulation tick, integrate `delta_time` into movement equations.
- Update robot coordinates based on **time passed**, not just simulation ticks.

---

3. Implement Time-Based Speed Calculations

3.1. Store Movement Start and Duration

- When speed is set, store the timestamp (`start_time`).
- When the speed changes, compute how far the robot has moved based on the duration.

3.2. Update Robot Position Using Time-Based Kinematics

- `new_position = old_position + speed × elapsed_time`
- `new_angle = old_angle + angular_velocity × elapsed_time`
- Use the `normalize_angle` function to keep the angle within `[-π, π]`.

---

4. Refactor the Event System for Updates

- The controller should fetch time-based positions from `RobotModel`.
- `RobotView` should update the visual representation accordingly.
- Ensure smooth UI updates in `RobotView`.

---

5. Implement Clock-Based Simulation Update

5.1. Modify the Simulation Loop

- Instead of `TICK_DURATION`, use real-time timestamps.
- Use `time.time()` or another clock mechanism to compute `delta_time`.

5.2. Ensure the Robot Moves Smoothly Over Time

- If a speed is set for **2 seconds**, the movement should be calculated over those 2 seconds, even if the simulation frame rate changes.

---

6. Handle Edge Cases

- **Stopping Movement:** Ensure speed is set to `0` when needed.
- **Collision Handling:** Adjust to time-based calculations.
- **Simulation Speed Changes:** Ensure smooth movement updates.

Final Expected Changes

✅ **RobotModel**: Handles movement logic & calculations.
✅ **RobotView**: Only draws the robot.
✅ **RobotController**: Manages interactions.
✅ **Clock-Based Movement**: Time-based speed calculations.
✅ **Smoother Simulation**: More natural movement updates.

Map

1. Refactor Map Class into MVC Structure

1.1. Separate Model, View, and Controller

- **Model (`MapModel`)**: Manages grid, obstacles, and start/end positions.
- **View (`MapView`)**: Handles rendering of the map and obstacles.
- **Controller (`MapController`)**: Updates `MapView` based on changes in `MapModel`.

---

2. Fix Obstacle Deletion Bug

2.1. Current Issue

- Clicking an obstacle does not remove it from the grid.

2.2. Possible Causes & Fixes

✅ **Incorrect Object Reference**

- Ensure obstacles are stored using unique identifiers in `MapModel`.
- Check that `MapController` correctly identifies the clicked obstacle.

✅ **Model Not Updating Properly**

- Verify that `MapModel` removes the obstacle from its internal storage.
- Ensure the update event is triggered after deletion.

✅ **View Not Refreshing**

- Ensure `MapController` notifies `MapView` after an obstacle is deleted.
- Force `MapView` to re-render the affected grid area.

---

3. Expected Changes

✅ **Obstacle Deletion Works**: Clicking an obstacle removes it properly.
✅ **Model-View Synchronization**: `MapController` updates `MapView` based on `MapModel` changes.
✅ **Efficient UI Updates**: Only affected areas are redrawn.

Simulator-AppView

1. Refactor Simulation into MVC Structure

1.1. Separate Model, View, and Controller

- **Model (`SimulationModel`)**: Manages time-based movement, robot state, and clock.
- **View (`SimulationView`)**: Handles GUI elements (or console output in non-GUI mode).
- **Controller (`SimulationController`)**: Orchestrates simulation updates based on model updates and user interactions.

---

2. Add Time-Based Simulation with Clock

2.1. Introduce a Simulation Clock

✅ Create a `Clock` class to track elapsed time.
✅ Store timestamps for movement calculations.
✅ Ensure `delta_time` is used in robot movement updates.

2.2. Modify Robot Movement to Use Time

✅ Store `last_update_time` in `RobotModel`.
✅ Compute movement based on `speed × delta_time`.
✅ Ensure continuous movement across simulation frames.

---

3. Enable GUI and Non-GUI Modes

3.1. Current Issue

The simulation only runs inside a Tkinter-based GUI.

3.2. Solution

✅ Decouple `SimulationController` from `AppView`.
✅ Introduce a `SimulationRunner` class to manage GUI and CLI modes.
✅ Use `argparse` to allow `--gui` or `--cli` mode selection.