Group number: 117   Therrense Lua (782578), Tianlei Zheng (773109)

Design Rationale

Introduction

The problem domain of this project consist of 2 parts (searching for keys and driving the car).

This is how we broke down the problem:

1. Searching: 3 types of **PathFinder**s
    a. Exploring Phase: using a wall following algorithm until we find position of all keys
    a. Finishing Phase: using A* search to find a path to all the keys and then the finishing goal
    b. Recovering Phase: to find closest health tile that has been seen using A* search when health hits a certain threshold
2. Driving: 5 types of **AutoPilot**s
    a. Maintain constant speed
    b. Turning 90 degrees
    c. Reversing for a certain distance (after colliding with a wall)
    d. Moving straight for a certain distance
    e. Recentring (re-adjust diverted path)

Next, we will explain each subsystem/component of our driving system and the design decisions related to each one.

Components of our driving system (mycontroller)

1. **MyAIController** is a *Facade Controller* that serves as the interface between the simulation environment and our driving system. It is also a *State Machine* that monitors the state of the game, and decides high-level goals. It delegates the actual work to **PathFinder** to find paths and to **Navigator** to execute the instructions.
2. **AutoPilot**[1] subsystem's purpose is to abstract away the lower-level controls of the car and provides discrete, atomic operations to higher-level subsystem. Each **AutoPilot** is responsible for driving the car for a segment of the path.
    a. The inputs and outputs to and from AutoPilot are encapsulated with the *Command* pattern
        i. **SensorInfo** wraps information of the car (speed, location, etc) and is created in MyAIController
        ii. **ActuatorAction** wraps the decision of **AutoPilot** (should we accelerate, should we turn, should we brake, etc.) and send them back to **MyAIController**. This way **AutoPilot** never directly invoke methods on **MyAIController**. (*Low Coupling*)
        iii. For the creation of both **SensorInfo** and **ActuatorAction** we use the *Factory Method* pattern**.**
    b. Some AutoPilots reuse or combine other AutoPilots to perform more complex driving (*Composite* pattern)
3. **PathFinder**s are responsible for finding the best routes depending on the stage of the game. The PathFinder system views the map as a discrete grid and outputs a path consisting of a list of coordinates to be on (e.g. (1,1) -> (1,2) -> (1,3) -> (2,3) ->...). We use the *Strategy Pattern* here, allowing us to use different PathFinder's depending on the state of the game:
    a. **WallFollowingPathFinder** is used to explore the map to find key locations

[1] An analogy is the autopilot systems on airplanes: the autopilot systems are not completely autonomous, rather, it accepts input from human pilots ("go to altitude xxx metres"; "turn to yyy degrees") and decide the force to apply on the airplanes' thrust / flaps / wings etc. to reach the goal.

   b. **FinisherPathFinder** & **HealthPathFinder** makes use of **AStarPathFinder** to find an optimal path to get all the keys and a health tile respectively (<u>*Composite*</u> pattern)

4. **RouteCompiler** component acts as a "translator" between PathFinder and AutoPilot subsystems. The reason we have this, is because our AI/pathfinding algorithm outputs a list of coordinates, but our actuators (abstracted by AutoPilots) need more information to control the car (i.e. "Turn left at (1,1)" is more meaningful to AutoPilot than "Go from (1,1) to (0,1)"). Also, by grouping together coordinates on the path that are on the same line, we are able to achieve things like accelerating on a straight path.
   a. **RouteCompiler** initiates the creation of the AutoPilots (with a level if <u>*Indirection*</u> via **AutoPilotFactory**), because it has the overall view of the path and (therefore each segment of the path) and is the <u>*Information Expert*</u> providing the information needed by the constructors of **AutoPilot**. Although PathFinder also have the information of the path, it would violate <u>*High Cohesion*</u> and <u>*Single Responsibility Principle*</u> if we were to let PathFinder do both the the algorithmic calculation *and* the "translation" of the path.
   b. The use of **AutoPilot*<u>Factory</u>*** is as a <u>dependency injection container</u> to inject the MapManager instance needed by AutoPilots, therefore decoupling **RouteCompiler** from this pure implementation concern. (<u>Low Coupling</u> and <u>High Cohesion</u>)

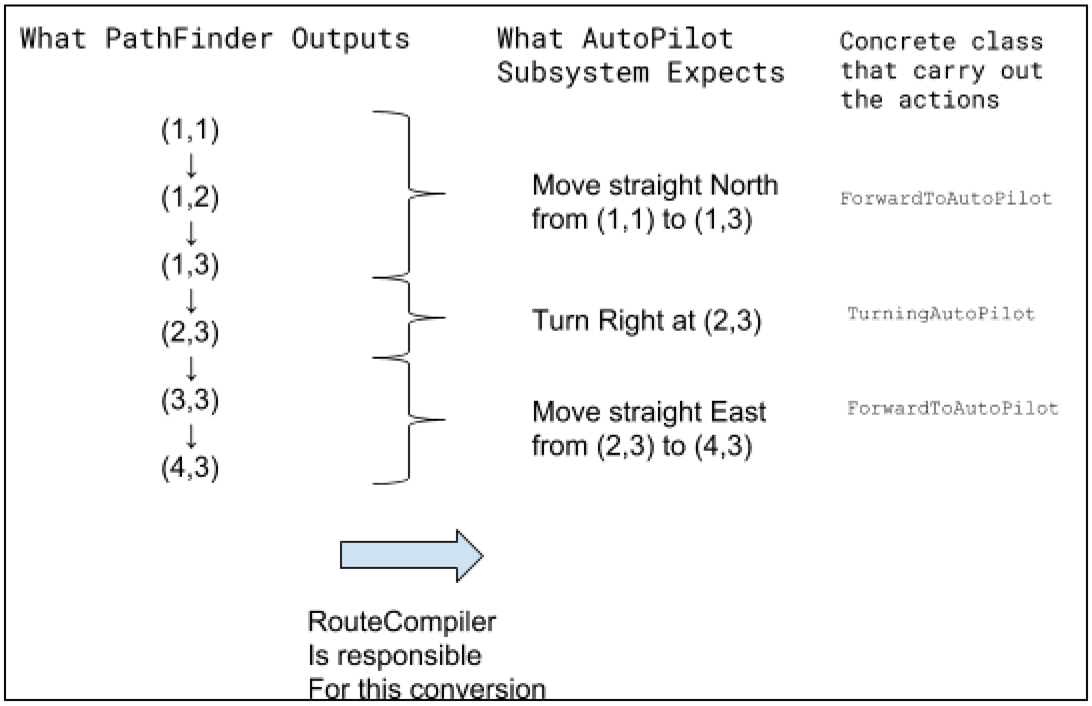
What PathFinder Outputs
What AutoPilot Subsystem Expects
Concrete class that carry out the actions
(1,1)
(1,2)
(1,3)
(2,3)
(3,3)
(4,3)
Move straight North from (1,1) to (1,3)
ForwardToAutoPilot
Turn Right at (2,3)
TurningAutoPilot
Move straight East from (2,3) to (4,3)
ForwardToAutoPilot
RouteCompiler Is responsible For this conversion

Figure 1: Illustration of the relationship amongst `PathFinder`, `RouteCompiler` and `AutoPilot`

5. **Navigator** component is a high-level "coordinator" of the Driving (not Searching) responsibilities.
    a. Similar to a navigator on a ship, it asks its minions (i.e. **AutoPilot**s) to do the heavy-lifting. It supervises **AutoPilot**s, making sure that they can carry out their duties one-by-one and no **AutoPilot** is acting when it should not be activated. However, unlike a GPS "navigator", our **Navigator** does not perform the path finding, that job is delegated to other subsystems.
    b. The **Navigator**'s responsibilities are:
        i. Keeps track of how far we have travelled on the calculated path
        ii. Make sure that the **AutoPilot**s are performing their duties in sequence.
        iii. When an unpredictable change occurs (interrupts) and when requested by MyAIController (e.g. car collided with wall, health below threshold, or current path navigation is done). **Navigator** finds a suitable location to

stop the car. Afterwards, MyAIController can consider re-routes (by calling relevant **PathFinder**'s to find a new path).

6. **MapManager** component, is the "memory system" of our car driving systems. It is a *Singleton*. It remembers what the car has seen along the way and builds up information about the entire map in the process. Other components (PathFinder, and AutoPilot in particular) query MapManager to get information they need for decision making.

   A side note here: we did not make **MapManager** a static class instead we used an Interface because we want the *flexibility & extensibility* of swapping out different implementations.
   a. For example, when unit testing our path finding algorithms, we can use a "mock" implementation of **MapManager** that loads the map from a text file (rather than getting it through exploring the interface). That way we can test our algorithms without the simulation system.
   b. Also, by defining an interface explicitly, it is clearer to our team (other developers) what features a **MapMananger** should provide.
   c. **Navigator** (also a *Singleton*) is the same case

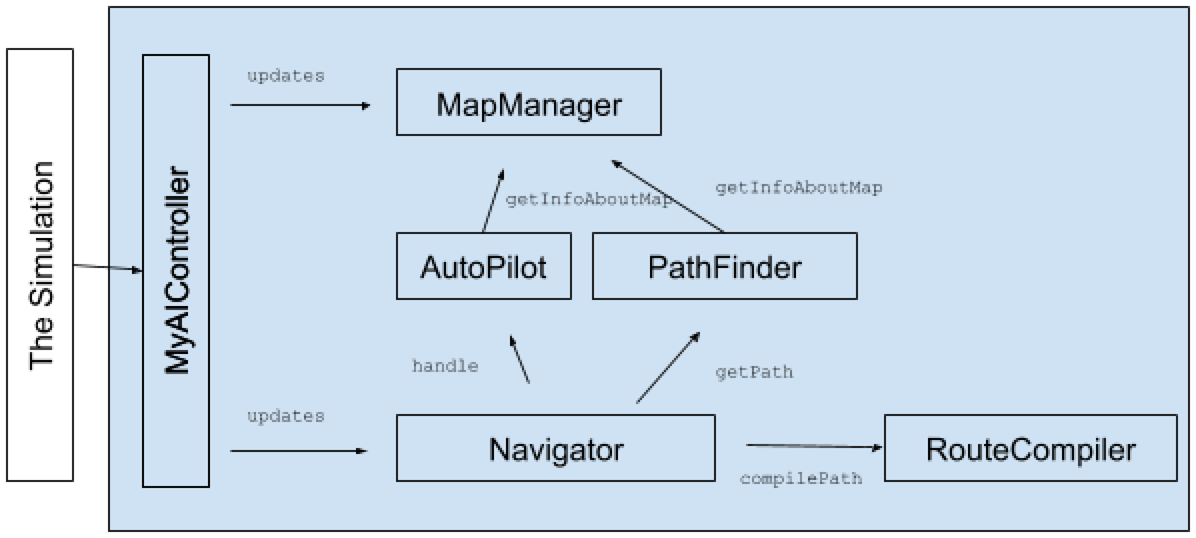

Figure 2: A conceptual overview of our subsystems (not UML). With all the responsibilities delegated, our MyAIController is basically just a mediator between our subsystems and the Simulation.

7. We used **ActuatorAction** (the low-level actions) and **SensorInfo** (the status of the car) as mediators to provide *Indirection* between MyAIController, Navigator and AutoPilot. This way our Navigator and AutoPilot don't need to directly call methods like Controller.applyForwardAcceleration(). Hence, they don't have references to the Controller. This way we can achieve low coupling and protected variation. It also allows us to modify/filter the output of AutoPilot or combine multiple AutoPilots to a composite AutoPilot.
8. **Cell** class is our internal representation of MapTile. Here is the justification:
    a. We wanted to design a cleaner and more concise way to access and check tile types.
    b. Without having our **Cell** class, to get position of the key from MapTile, we need to first check if it is a TrapTile and then cast it to TrapTile before calling getKey(). The MapTile.getType() method does not distinguish between HealthTrap and LavaTrap, so we need to use "instanceof" multiple times to determine its specific type. This way of accessing and checking can make the code of our search algorithm and might cause performance issues.
    c. After all, the internals of our driving system (the mycontroller subpackage), and specially the algorithmic part of it can be considered as belonging to a different *domain* as the domain of the simulation system. So it is fair to have different representations of the same concepts in different domains.
    d. An alternative to this, is that we could have represented every data structure as plain strings and arrays for different type of tiles, but this would not allow for easy change if we needed to consider more Tile types, also it would be very cumbersome to find every instance of the String should we need to change them.

Other implementation details

- **Util** class contains common reused methods and constants
- **Logger** class is for centralised control debugging output, with varying degrees of verbosity