Code Challenge 03

Alejandro Mujica

- **Read carefully all this document before asking questions.**
- **Range of dates to submit: from 03/14/2022 to 03/18/2022.**
- **The range of dates will not be changed under any circumstance. Be aware of that.**
- **You must submit two files, they are: `Geom.h` and `Pathfinder.h`. Please, at the top of those files write the id numbers and the first and last name of the couple.**

1. System requirements

The mandatory requirements are:

- C++ version 14 or higher.
- DeSiGNAR.
- make.
- clang.

You can also use the following optional requierements to debug:

- DDD.
- valgrind.
- Qt6

2. Introduction

The goal of this challenge is to learn some of the applications of graph theory and computational geometry, specifically, in the video games field[1].

The first problem you have to solve is a set of geometrical-supported routines. Those routines are used by an agent to be able to move by itself into an environment with obstacles and avoid (as possible it can) being seen by a camera.

[1]Although, this type of model is used also in the field of mobile robotics to model environments..

The other problem is a pathfinder. That is a data type that will help the agent to compute its mission path fulfilling all of the above.

Note: The class `Vector2D` is an extension of `Point2D`. In some cases, you will see operations and types that contain vectors and others that contain points. Both of them have different semantic meanings. The geometric algorithms work with points, however, to solve kinematics problems, we use vectors to represent the position, velocity, acceleration, forces, etc.

Some of the methods have different meanings when using them either on vectors or points. E.g: determining whether a point is to the left or the right of a pair of points (a segment). When working with vectors, that operation compares a vector with only one other vector because they are like a segment from the origin.

Pay attention to the methods of `Point2D` hat are overridden on `Vector2D`.

Also, consider the inheritance meaning. In an operation that receives vectors as parameters, you can pass points. The opposite is not correct.

3. The game

The game consists of a rectangle-shaped flat environment that contains some obstacles. In the middle of this, it is placed a security camera with peripheral vision defined by an angle and limited by a distance radius. In this environment are a set of agents and a set of resources that are spawned randomly. The agents must try to take resources as much as can without being seen by the camera. If an agent is seen by the camera, then it dies. The camera is looking at a region of the environment and, each 5 seconds, it rotates counterclockwise. When a resource is taken, it will be respawned after 10 seconds.

4. The challenge

You are provided with a `Makefile` and test cases for each of the parts. They are in the files `GeomTest.cpp` and `PathfinderTest.cpp`.

Also, you are provided with the files `Definitions.h` and `Definitions.cpp` which have the abstractions to use to solve the problem. **Do not modify those files. I encourage you to study them deeply**. The abstractions provided in those files are the following:

- `EuclideanGraph`: an alias for the concrete type `Graph<Point2D, double>`, EuclideanGraph: an alias for the concrete type Graph¡Point2D, double¿, that is a graph with nodes representing points on the plane and arcs representing connections between them. An agent will move through the arcs. The weight of the arcs will be the Euclidean distance between the connected points.

- `Obstacle`: an alias for the concrete type `Polygon`.

- `sinX` and `cosX` are constants that hold the results of the trigonometric functions sine and cosine for notable angles (0, 30, 45, 60, 90).

- `degree_to_radian(a)` is a function that converts to radians an angle given in degrees.

- `dsin(a)` and `dcos(a)` are functions that compute the sine and the cosine of an angle given in degrees.

- `Mat2D` is an abstract type that defines a 2×2 matrix. It is useful to do transformations on points on the plane by setting it up with the proper values. This type exports a constructor that receives its four components in order $(m_{11}, m_{12}, m_{21}, m_{22})$. It also exports the operator `*` that receives as a parameter a vector `v` (a column vector) and computes the dot product with it. The dot product with a vector applies the transformation in it, then returns the transformed vector.

- `Camera` is an abstract type that defines a camera. This is composed of a position vector, a unitarian vector that defines where the camera is looking, a vision radius, and a vision angle. Those values are exported through the following methods: `get_position()`, `get_front()`, `get_vision_ratio()` y `get_vision_angle()`.

- `VisionArea` is an alias for a tuple that contains three vectors and a scalar. This tuple represents a camera vision area. Each element at the tuple is defined as follows:
  - The first vector is the camera position.
  - The second vector is the position of the other side of a segment that starts with the camera position. This segment should have the vision radius as length and it should have an angle *alpha* clockwise with the unitarian vector that defines where the camera is looking.
  - The third vector is similar to the previous one. It defines the position of the other side of the segment that starts with the camera position. The only difference is this vector has an angle *alpha* counterclockwise with the unitarian vector.
  - The scalar value is the vision radius.

- `Resource` is an abstract type that defines a recourse to be taken by the agents.

- `Terrain` is an abstract type that defines the environment. It is compose by a width and a height which could be retrieved by the methods `get_width()` and `get_height()`. It also contains a list of agents, a list of resources, and a list of obstacles that could be retrieved by the method `get_obstacles()`.

Moreover, it is provided a directory called `GraphicGame` that contains a little project developed with `Qt5`. It shows a graphic interface that shows as the game evolves.

Your job here is to modify the files Geom.h and Pathfinder.h by programming the empty routines in order to make the game work.

In the file `Geom.h` you should solve the following problems:

- Solve

  ```
  inline bool is_inside(const Point2D& p, const Obstacle &o)
  ```

  It checks whether or not the point `p` is inside the obstable `o`.

  Hint 1: All of the obstacles are built (by convention) clockwise.

  Hint 2: Some of the quantifier methods on segments that are provided in the class `Polygon` could help you.

- Solve

  ```
  inline bool is_inside(const Point2D& p, const SLList<Obstacle>& os)
  ```

  It checks whether or not the point `p` is inside of any of the polygon in `os`.

  Hint: This routine could use the previous one as support.

  The same hints apply to the next two problems.

- Solve

  ```
  inline bool intersects(const Segment& s, const Obstacle &o)
  ```

  It checks whether or not the segment `s` intersects with the obstable `o`. `o`.

- Solve

  ```
  inline bool intersects(const Segment& s, const SLList<Obstacle>& os)
  ```

  It checks whether or not the segment `s` intersects with any of the obstables in `os`.

- Solve

  ```
  inline Vector2D translate(const Vector2D& v, double dx, double dy)
  ```

  It returns the resulting vector of translating the vector `v` by `dx` units horizontally and `dy` units vertically.

- Solve

  ```
  inline Vector2D rotate(const Vector2D& v, double a)
  ```

  It returns the resulting vector of rotating the vector `v` in `a` degrees respect to the origin of coordinates $(0, 0)$.

  Hint: The type `Mat2D` is useful for this operation. See `https://en.wikipedia.org/wiki/Rotation_matrix`.

- Solve

```
inline Vector2D rotate_around(const Vector2D& v, double a,
                              const Vector2D& u)
```

It returns the resulting vector of rotating the vector `v` in `a` degrees respect to the position vector `u`.

Hint: Note that if the vector `u` were the origin, then you could apply the previous operation. Thus, think about these questions:

1. How could I translate the vector `v` in order to simulate that `u` is the origin?
2. What should I do with `v` after the rotation? Perhaps, translating it again?

- Solve

  ```
  inline VisionArea build_vision_area(const Vector2D& p,
                                      const Vector2D& f,
                                      double r, double a)
  ```

  It builds the vision area of a camera that is in the position `p` looking at `f` with vision radius `r` and vision angle `a`.

- Solve

  ```
  inline bool is_inside(const Point2D& p, const VisionArea& va)
  ```

  It checks whether or not the point `p` is inside the vision area `va`.

  Himt: Remember that the vision area is defined by position vectors and a radius. Those vectors are points by inheritance.

The file `Pathfinder.h` contains the class `Pathfinder`. On it, you should solve the following problems:

- Solve the constructor

  ```
  Pathfinder(Terrain* t, double num_hnodes, double num_vnodes)
  ```

  It should copy the argument `t` on the attribute `terrain` and builds a grid-shaped graph (on the attribute `graph`). For the graph, each node should be connected with all its neighbors (left, right, up, down, and the diagonals).

  To solve that, consider that each node should have a suitable coordinate according to the terrain. Let's think that you split the terrain by `num_hnodes` cells horizontally and `num_vnodes` cells vertically. That would generate rectangular cells. Each node of the graph should be placed in the center of each cell. You should set to the arcs the distance between the nodes that it connects.

  Hint: Check the function `build_grid` in th file `buildgraph.H` in DeSiGNAR.

- Solve the method

  ```
  void apply_obstacles()
  ```

  It should remove from the graph all of the nodes that are inside of any of the obstacles. Also, it should remove the arcs that intersect with any of the obstacles.

  Hint 1: Each node has a coordinate. That is a point.

  Hint 2: Each arc could be seen as a segment formed by the points inside the nodes that it connects.

- Solve the method

  ```
  SLList<Point2D> search_path(const Point2D& s, const Point2D& t)
  ```

  It returns a list of points that represents a path between the points `s` and `t`. You should solve this problem with the $A*$ algorithm (`https://en.wikipedia.org/wiki/A*_search_algorithm`) by using the info of each arc as the weight and the Euclidean distance between the visited node and `t` as the heuristic.

5. Evaluation

- This challenge must be solved by couples. The matching will be written on Discord.
- To submit your solution, you must send the files Geom.h and Pathfinder.h to the following email:

  `alejandro.j.mujic4@gmail.com`

- The valid dates to submit your solution are from 03/14/2022 to 03/18/2022.
- Your submissions are limited to only one time a day into the valid range of dates.
- I will do my best to send you a response to your submission ASAP.
- The subject of the email is **"CC03"** without the quotation marks.
- The mail body must contain the name and the id number of both of you.
- Do not compress the files and do not send any other files.

6. Advices

1. You have been provided with test cases. Do not assume that your solution is right given the fact that you pass the test. Build your own test cases, verify boundary conditions and high-scale handling.

2. Be carefull with the memory management. Make sure that no leaving memory leaks in the case that you use dynamic memory. For every `new` should exist a `delete`. DDD and valgrind are good friends.

3. Use the proper channel to ask questions and comments. I will not answer private messages.

4. **DO NOT, DO NOT, DO NOT, and DO NOT** Do not include any preprocessor directive in your files `Geom.h and Pathfinder.h`. **If you do, your submission will be ignored**.

5. **DO NOT, DO NOT, DO NOT, and DO NOT** Do not print messages in your solution. Use the proper debugger or use the provided macro `LOG`. **If you do, your submission will be ignored**.

6. **Do not forget that you accepted the Honesty Agreement**.

7. Enjoy this challenge and happy coding!