README

This is a dynamic pathfinding simulation. The board consists of 8-12 obstacles of T or U shape. The user could generate one agent and one destination at random position, and the agent would follow the optimal path to the destination.
There are 5 buttons on the UI. 3 of them will generate agents with different radius, small (2f), medium (3f), and large (4f), at random positions. The test button creates a random small agent/destination that is temporarily stored, and the RVPmode button switches between Naive RVG and Improved RVG, which could be used to compare the path costs using the same test points.

Features:

1. Obstacles (ObstacleGeometry.cs)

Each obstacle is a prefab constructed by cubes. Along the mesh for the actual obstacle, a transparent collider that surrounds the obstacle is added, and its vertices are used in stage generation and RVG computation. This is to ensure that all paths that the RVG constructs leaves enough space for the largest agent to pass, without colliding with the actual obstacle.
Reflex vertices are calculated by taking the bottom vertices of the transparent collider, and selecting only the points with an extreme x or z value in the local space, then transformed to the world space. This does not follow the definition of reflex, but it works. The U shape is additionally processed to remove redundant points. An enclosing circle is also computed using reflex points, first finding a centre, then expanding the radius to include all points.
For computation simplicity of the RVG, there is also a method that shifts all reflex points of the obstacle outside 0.01f by averaging the normals of the neighbouring edges.

2. Level (LevelGenerator.cs)

A level is filled with 8-12 obstacles, each instantiated at random position and orientation. An edge padding is added when generating to allow the agent to pass between border edge and obstacles. The script uses the enclosing circle of the obstacle, randomly attempts 1500 times per obstacle and verifies that it doesn't collide with existing obstacles.
The board is also divided into 6 areas and each is assigned a random terrain cost between 0.5-5.

3. RVG (RVGGenerator.cs)

This script waits a short interval to allow the LevelGenerator to populate the board. Then, it takes all the reflex vertices from the obstacles, adds them to the RVG, and creates pairwise edges between them. An edge is checked to be "visible" before adding.
To be "visible", 2 raycasts are shot. One starts at vertex v1 and shoots towards v2, with length of the distance between v1 and v2 plus a slight extension of 0.5f. The other shoots backwards from v2 to v1 with slight extension. If any of the rays hits an obstacle, it is pruned. This ensures all edges left are bitangent.
After the creation of the base RVG, once an agent and a destination is instantiated, they will be added to the RVG and corresponding edges are added. When they disappear, the RVG will remove those edges.

4. Pathfinding (Pathfinder.cs, ImprovedPathfinder.cs)

The naive pathfinder implements a standard A* algorithm on the RVG. It follows the standard A* search pattern, using the Euclidean distance multiplied by the average sampled terrain cost between nodes as the heuristic. Because the heuristic is admissible, the solution guarantees the optimal path under the current RVG connectivity and terrain sampling resolution.
The improved pathfinder builds on the results of the naive approach. It applies a recursive simplification strategy similar to the RDP algorithm.
For each consecutive segment of the original path, it attempts to replace intermediate vertices with a direct connection if:

1. the line segment is unobstructed (checked via raycasts against colliders tagged "Obstacle"), and
2. The direct segment cost (terrain-aware) is strictly lower than the original subpath cost by epsilon threshold.

Otherwise, the algorithm splits the segment at the furthest point and recurses on both halves. This process produces a simplified path with equal or better total cost without violating obstacle or terrain constraints.

5. Agent (AgentManager.cs)

The agent manager provides buttons to spawn agents of varying sizes, toggles between Naive and Improved pathfinding modes, and triggers new randomized test scenarios.
When a scenario starts, it deletes the previous agent/destination, re-generates valid start/goal positions free of collisions with obstacles or each other, updates the RVG with dynamic points, requests a path from the selected pathfinder, and animates the agent's motion along it at a speed inversely proportional to terrain cost.

6. PathfindingTester.cs

This automated tester performs batch evaluations of the two pathfinding strategies to estimate consistency and improvement rate. For each of the total 100 iteration, it:

1. Randomly samples start/goal pairs on valid terrain,
2. Computes both naive and improved paths,
3. Records their cost, point count, and percentage improvement,
4. Logs whether the improved path achieved cost reduction or simplification.

At the end of all iterations, it aggregates statistics such as improvement success rate, average percentage gain, average number of points saved, and best-case improvement sample.

Limitation:

- The method used to retrieve reflex points of obstacles is not universal.
- Terrain cost estimation relies on sparse sampling.
- The Improved Pathfinder's performance may be related to epsilon.