Advanced programming classes #2

— IMAC 3rd year —

Path finding

Part 2/4 of « Boids Tower Defense »

Objectives:

- To use existing code and documentation
- To use path finding algorithms, from the BGL (Boost Graph Library)
- To address multithreading concepts
- To look for optimizations solutions

Work to be done in two-persons teams, to be then sent to your teacher (neilb@free.fr).

A complete path finder class (“`PathFinder`”) is provided. It contains the following features:

- Creating a graph that corresponds to the game’s terrain (height field)
- Path finding using Dijkstra algorithm
- Modifying the graph by adding and removing obstacles. Those are 2D points or circles projected onto the graph.
- Documentation
- And a “debug” function which creates a displayable object representing a path

Furthermore, the path finder class has a proposed encapsulating manager (“`GamePathFinder`”). This manager allows:

- Running the path finder in a separated thread
- Handling concurrent accesses to the path finder (thread safety)
- Accept commands[1] the path finder will sequentially process (commands are queued by a thread, and processed in another)
- “Warn” every registered object about the modification of the path finder’s graph, using callbacks

However, execution time of the path finder is still extremely long, especially when adding or removing obstacles. You will thus have to implement solutions to achieve acceptable respond times for the system.

Please note that the proposed path finder is simplified, and is not intended to be used for simulations. Indeed, it cannot be updated with a fixed time step. As a consequence, code can produce different results on different machines, depending on the available processing power (more paths updates on a fast machine for example).

1 *Design pattern*

Exercise 1 – Adding/removing obstacles asynchronously

In the current system, two kinds of terrain obstacle implementations exist:

- Temporary obstacles, which are fast to add and remove. Technically speaking, they simply change the weight of graph's edges (making distance "infinite" where obstacles are present).
- "Permanent" obstacles. They are designed to be permanent, though they can be technically removed. They are slower to add and remove, as they actually remove graph edges (see code documentation about `addSizedObstacle(GamePathFinder &, const osg::Vec3 &, float)`). Future path finding queries will thus have less edges to scan. Those obstacles are thus supposed to stay for a "long" time.

When building a defense tower on the ground, we should logically add a "permanent" obstacle, as the tower is not intended to be removed in a short period of time. Unfortunately, adding such an obstacle is way too costly (CPU time).

The way to "properly" update the path finder's graph (with a "permanent" obstacle) will thus to do it in a separated thread ("`GamePathFinder`" class) to avoid blocking the main thread. And in order to make the obstacle appear instantly, we can safely perform a "double" obstacle:

1. First, we should synchronously (main thread) add a temporary obstacle. This is already implemented, and uses the `addSizedObstacle()` function.
2. Then we would asynchronously (separated thread) add a permanent one over the temporary one (ie. remove the edges with "infinite" distance).

You will implement such a "double" obstacle, by adding a command (point 2) for it. The command system is already implemented and tested, but the actual command doesn't exist yet. See "`PathFinderCommand`".

As for all exercises, implementation quality, documentation, and coding rules respect will be particularly important.

Secondary objective – Asynchronous removal

Add asynchronous removal, as you did for asynchronous addition.

Exercise 2 – A* finding

Dijkstra algorithm is an exact one, but can take a significant amount of time on a graphs like the one we use (65536 points, ie. 256×256), or larger. Moreover, our game will certainly have to support much more path finding queries (more units) and a larger and/or more detailed terrain.

You will complete the projetct by adding an A* ("a-star") path finding (approximated algorithm). Add it in `PathFinder::compute()` using A* BGL[2] functions. The subtler your heuristic, the best it would be.

Secondary objective – Continuous A*

A* is an approximation. Add a re-computation at regular time intervals. A moving unit may thus have increased accuracy as it approaches the goal, especially if using finer heuristics (or even Dijkstra) when close.

Exercise 3 – Near-optimal path finder

Most computed paths (and especially long ones) are thrown away, because units may receive a new order before reaching the goal, because of added obstacles, or whatever. Propose a theoretical solution (ie. a paper) to both reduce this waste and keep fine grained precision for short paths.

Important: Your answer must be a technical study, not a vague idea in a dozen lines. I expect an engineer's solution to a concrete issue – based on the prerequisite, of course, that this optimization is required.

Secondary objective

Implement your paper. This is optional, but will be really appreciated. You may even simplify your implementation, regarding to your original and theoretical solution.

2 *Boost Graph Library – Documentation is available at boost.org, or locally with provided libraries.*