First Assignment (Group Project)
AI Programming for Games
COMP09041

Issue Date:	Monday, 7th February, 2022
Due Date:	**5pm, Friday, February 25th, 2022**

Steering and A* Path Following

In this assignment you are provided with an interactive C++ program which uses the steering classes from the first weeks' labs; along with the `Graph` class introduced in week 4, through the code from Red Blob Games. You are now asked to modify the provided program in a number of specific ways, and these should each be explained in a short accompanying report. Your zipped submission should include your report along with the modified source code.

This is a group project. You are each assigned to a team. Team members are listed within the teams-assignment1.md file stored in the Files area on MS Teams: here.

Background

Upon starting the program, you are presented with the red player ship following a path through a graph, with nodes labelled from 'A' to 'G'. The graph class `Graph` is defined in graph.gpp, and adopts the simple Red Blob Games interface requirement of two member functions called `neighbors` and `cost`.

The main data member of the `Graph` class is an `std::unordered_map` called `edges`, which is basically a *hash map*. This hash map can be used like an array, except the index type will vary; in this case, we are using a `char` as the index (type aliased as `node_t`), and the value returned is a `std::vector<char>`, which provides the *neighbours* to the current node.

This is a simple but effective representation of a graph. Two things are missing though: information on each node's 2D/3D spatial coordinates; and edge costs. These are provided by two global variables defined in graph.hpp as shown below:

```
using node_t = char;
using edge_t = std::pair<node_t, node_t>;
std::unordered_map<node_t, Vector> node_info;
std::unordered_map<edge_t, double> edge_info;
```

The `node_info` object provides additional information about each *node* in a graph; and here this provides the node's coordinates as a `Vector`. So, `node_info[’A’].x` would provide the *x coordinate* of a node called ‘A’. Meanwhile, `edge_info` provides similar auxiliary information about each *edge* in a graph. An edge is defined as a *pair* of nodes. As the graph is directed, there may be one edge from, say, ‘B’ to ‘C’; and another edge back from ‘C’ to ‘B’. The `std::pair` class template is then used to enquire from `edge_info` on the *cost* of traversing an edge; so `edge_info[std::pair<node_t>(’B’,’C’)]` would return the `double` value corresponding to the travel cost.

```
void add_node(Graph& g, const node_t& n, const Vector& v);
void add_double_edge(Graph& g, const node_t& n1, const node_t& n2);
```

The function declarations for `add_node` and `add_double_edge` are defined in graph-util.hpp, and shown above. The `add_node` function will both add a node `n` to the `Graph g`; as well as associate it with the position encoded in the value held in `v`. Meanwhile, the `add_double_edge` function will add *two* edges to the graph, between nodes `n1` and `n2`; *in both directions*. The *cost* associated with each direction is simply the distance between the two nodes.

The `a_star_search` function provided in Red Blob Games' implementation.hpp can be used with this graph. You should look at the example program in a_star.cpp from week 4's lab as a guide when considering how to apply A* here. Note especially the need for a subsequent call to the `reconstruct_path` function; which returns an `std::vector<node_t>` object. This vector of nodes needs to be translated to a vector of path coordinates before the waypoints of `FollowPath`'s `path_` member need updated. Code such as the following may be useful at this step:

```
std::vector<node_t> p = reconstruct_path(start, goal, came_from);
std::vector<Vector> v;
for (const auto& c : path) {
  v.push_back(node_info[c]);
}
```

Assignment Brief

Modify the code provided via the astar-steering-assignment project, according to the instructions below:

Attempt the following 8 tasks. In addition, a 1000 word report should be provided in pdf format. The report should start by briefly introducing the context of the assignment, before describing the approach taken for each of the completed tasks. You are encouraged to include figures, which might include screenshots, or short code excerps (say 2 or 3 lines for each one). Include a conclusion.

1. Before the `while` loop in the `main` function, use `add_node` to add a new node 'Q' to the `Graph` object `g` at coordinate $(100, 0, 100)$. **(1 point)**
2. Use `add_double_edge` to connect 'Q' with nodes 'A' and 'B'. **(2 points)**
3. Find the nearest node in the graph relative to the position of the mouse when the button is clicked. Use `DrawText` and `TextFormat` from Raylib to display this node (e.g. 'A') and its coordinate (e.g. (384,0,312)) at the bottom left of the screen. **(3 points)**
4. Next, print the nearest node in the graph relative to the position of the red ship (also when the button is clicked). Again use `DrawText` and `TextFormat` to display this nearest node and coordinate at the bottom left. **(3 points)**
5. Now use A* to obtain a path from the node nearest the ship to the point clicked. Use this information to move the ship via the `FollowPath` component of the `blend` object. The `set_waypoints` member function of `FollowPath`'s `path_` data member will be useful. **(5 points)**
6. Currently, the A* heuristic function in graph.hpp uses the *manhattan distance*. Change this to use the *euclidean distance* (`Vector::length` may be useful). **(2 points)**
7. Double the *cost* of the connection between 'F' and 'B' (leave the cost from 'B' to 'F' unchanged) by updating `edge_info`. You should see that A* will now avoid using this edge in any routes it calculates. **(2 points)**
8. So far, the `FollowPath` class (in follow-path.hpp) has used simple *vanilla* path following. Add a new class called `PredictvePath` to a new header file called predictive-path.hpp that *predicts* where the character will be. Refer to the pseudocode on slide 55 of the **Movement and Steering** lecture slides. **(2 points)**

Resources

As well as the main C++ file astar-steering-assignment.cpp, a complete set of steering classes (now including `FollowPath` and `Path`) are included in the `steering` subdirectory. The implementation.hpp header file from Red Blob Games is also included, along with two header files which help with the non-grid graph that we need (graph.hpp and graph-util.hpp). The usual Raylib C++ library, and an updated `raylib-extras` directory are also included; the `Vector` class defined in vec.hpp now includes support for equality comparisons, and support for streaming to standard output via `std::cout`. Audio resource files from Raylib are included too.

The assignment is worth 30% of the marks awarded for the entire COMP09041 module. The following provides a summary breakdown of the marking scheme:

1000 word report with figures	10
1. Add a new node ‘Q’	1
2. Connect ‘Q’ with ‘A’ and ‘B’	2
3. Display the nearest node to the mouse position	3
4. Display the nearest node to the red ship	3
5. Use A* to navigate the ship via the nodes	5
6. Modify the heuristic function used by A*	2
7. Double the cost of travelling from ‘F’ to ‘B’	2
8. Deploy *predictive* path finding	2

Plagiarism

Ensure your work is developed only by your own team. You can discuss ideas with other teams, regarding how to prepare a solution, but the *copying or sharing of code is not permitted*.

Anonymity

Please use only the Banner IDs of your team members to identify yourselves in your submission. Ensure the Banner IDs of all team members are on the first page of your report.