2.2 Important Functions

The programming language used (and referenced here) is C#, chosen on account of the extensiveness of libraries and functionality associated with the .NET environment (as a consequence, the application will run exclusively on Windows). All code segments here are written in pseudocode.

2.2.1 Export Integrity Checks

Lines 166 & 180, DesktopMapper/test/NodeClass.cs

At the end of layout creation, several criteria must be met in order for the layout produced to be acceptable for use in the smartphone application. Among other criteria, the graph must be **connected**; it must be possible to form a path from any node to any other node, or some rooms may be unreachable.

In order to check this, the procedure described below uses graph traversal to verify connectivity. Firstly, a queue (of nodes to traverse) is established; an arbitrary node is selected (e.g. the first node of the adjacency list), and enqueued, representing the first node to inspect. Secondly, an array of Booleans (whose length is equal to the quantity of nodes) is initialised, with each element representing whether each node has been visited; since the node chosen before is being inspected, it is marked as visited (true) in this array.

Now the following loop is repeated;

* Dequeue an element from the queue, and store it as the current node to inspect.
* For each unvisited node connected to this node:
  + Mark it as visited in the Boolean array,
  + Enqueue it in the queue.

This process is repeated until the queue is empty; the procedure then returns the Boolean array. If the graph is connected, every element will be *true* (all nodes have been visited). Otherwise, the checking process can halt and fail, and the *false* elements (unvisited nodes) can be printed to alert the user of which nodes to edit.

This is summarised in the following pseudocode, taking an adjacency list as input where each element is a list of tuples (as described in section 1.1.1).

traverseConnectivity(List<float> Input)

Initialise Queue;

Initialise BoolArray (with length equal to that of Input);

Initialise all elements of BoolArray as *false*;

Queue.Enqueue(0); //arbitrarily choosing first node of list to begin with

While (Length of Queue != 0)

currentNode = Queue.Dequeue;

foreach(node in Input[currentNode]) // i.e.for each connected node

if(BoolArray[node] == false)

Queue.Enqueue(node);

BoolArray[node] = true;

Endif

Endfor

Endwhile

Return BoolArray;

2.2.2 Managing Nodes

Lines 50 & 56, DesktopMapper/test/NodeClass.cs

Nodes can be added and deleted from the graph at any point. To allow this, procedures for each must be considered.

Creation, when a space on the layout view panel of the General Editing view not covered by a node is clicked: A new instance of the ‘Node’ class described in 1.1.1 is appended to the node list; its values for each parameter are initialised as follows:

* OffsetPosition: First element of tuple is equal to , and the second element is equal to .
* Floor: the floor currently displayed.
* Visible: *true* if clicked with the SHIFT keys held, *false* if clicked with the CTRL key held only.
* Description: All elements initially empty strings (later bound to the text boxes of the node panel).
* AdjacencyList: An empty list (not yet connected to any nodes).

Deletion, when the delete button in the node panel is clicked:

* The index of the node is temporarily recorded.
* The node is removed from the node list.
* All AdjacencyList parameters of all other nodes are now scanned: if any have a connection with the node just deleted (i.e. contain a tuple with first element being the deleted node), remove that tuple from the list.

2.2.3 Managing Node Connections

Lines 127 & 149, DesktopMapper/test/NodeClass.cs

In the Node Edit state, connections between nodes can be created and deleted. This is managed in the adjacency list in the following ways;

* Creating a connection between nodes *a* and *b*:
  + First, calculate the distance between the two nodes;

Let = a.leftOffset – b.leftOffset (horizontal distance between nodes),

and = a.topOffset – b.topOffset (vertical distance between nodes);

Distance = .

* + In the connection list for *a* (i.e. the *a*th element of the adjacency list), add a new tuple (b, Distance) representing connection between *a* and *b* with edge weight Distance.
  + Repeat the above, swapping *a* for *b* (if *a* is connected to*b*, *b* is connected to *a*).
* Deleting a connection between nodes *a* and *b*:
  + In the connection list for *a* (i.e. the *a*th element in the adjacency list), iterate over the tuples in the list; if the first element of the tuple is *b*, remove the tuple from the list.
  + Repeat the above, swapping *a* for *b*.

2.2.4 Managing Floors

Lines 95, 103, DesktopMapper/test/NodeClass.cs

Similar to the above, the user must also be able to create/delete floors, handled as such:

* Adding a floor above the *a*th floor:
  + Insert the floor filename (in string format) into the floor list at the index of *a* + 1, thereby inserting the string between the *a*th element and the element proceeding it.
  + Iterate over the list of nodes; if a node has Floor **greater than** *a*, increment the node’s Floor value by one (it is now a floor above what it was previously).
* Adding a floor below the *ath* floor follows the same procedure as above except for the following:
  + The floor name string is inserted into the floor list at index *a* instead of *a* + 1.
  + The currently iterated node’s Floor value should only be incremented if the Floor value is **greater than or equal to**, as the floor previously represented by the *a*th element is also moving up a floor.

2.2.5 User-Manipulated View

Line 851, DesktopMapper/test/MainWindow.xaml.cs

The facility for the user to be able to move the view of the current floor must be available, in the form of panning the view of the floor (holding down mouseclick and moving mouse) and zooming in/out (mousewheel scroll), implemented as a new *PanZoom* view class inheriting from the base view class, with the following methods:

* Declare the integer variables *panX*, *panY*, and *Zoom*.
* If the mouse is moved (and is held down, having been initially clicked at point (x, y)), *PanX* is increased by the mouse’s change in horizontal position (ditto for *PanY* for vertical change).
* If the scroll wheel is moved, the *Zoom* variable is increased/decreased depending on whether the scroll wheel was scrolled forwards/backwards, respectively.
* Every time the view class is rendered, the child view belonging to the *PanZoom* view has its horizontal/vertical position inside the *PanZoom* view assigned the values specified by *PanX* and *PanY* respectively, and ditto for the child view’s zoom scale factor with the *Zoom* variable. Note that the position of the *PanZoom* view itself is not altered as doing so would affect how mouse positions are interpreted, leading to unexpected behaviour.