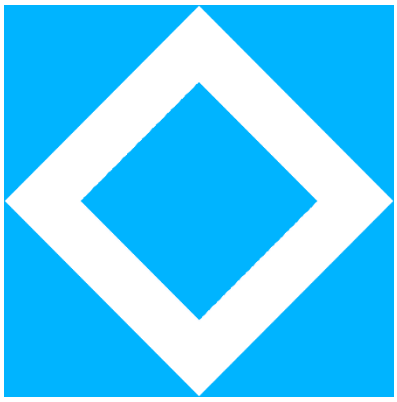


PSTMETRO

The creation of a Metro planning system

NOVEMBER 26, 2013

PSTMETRO
Thomas Nairn, Pascal Meyer, Sean Rozario

Contents

Introduction:

The objective of this task is to create a journey planner for the Prague Metro system. This system should allow a person with minimal technical aptitude to easily access and plan a journey through Prague.

This can be achieved by creating a user-friendly graphical user interface or 'GUI' with the following traits:

- Simplistic: A minimal interface as to reduce the possibility of incorrect user input.
- Elegant: Making the application appealing will ensure that the application is used over of its competitors.
- Manageable: The application must be able to efficiently manage and add 'Metro Systems'.
- Extendible: The interface must be able to easily accommodate extra functionality that will not be included in the first iteration of the application.

These goals are achievable by using the Java programming language. This language is able to be deployed on multiple operating systems ensuring the application can be used by a multitude of computer systems regardless of specifics. It is object oriented allowing an object-based approach to solving the problem.

Due to the vague nature of the task at hand, certain assumption have had to be made. These assumptions are as follows:

- The target platform is able to run java and has the required disk space for the files associated with the application.
- The person running this application also has the privileges to modify a metro system.
  Note: This application was intended for system admin use. Were this application intended for public use ONLY, we would edit the application such that it exclusively opened a planner view.
- The person using the application has fully read the user/help documentation established by our team. Note: We have been unable to incorporate the user/help documentation within the program due to time constraints.

Design of the model

The main objective of this task is to efficiently determine the quickest path of travel between stations. By looking at this objective from another mind-set we were able to apply our knowledge graph theory.

Graph theory is the study of graphs or mathematical structures made up of 'nodes' and 'edges'. In a metro system, we conclude that stations become nodes and the connections between these nodes become edges. Using graph theory we are able to employ many different well documented graph traversal and shortest path theories, the 2 we considered are as follows:

- A* Search Algorithm: This is a single pair algorithm, this means that we must know both the source and destination of the current journey. This can improve speed where the destination is not likely to change and would be very handy should the user of the application be traversing the map.

- Dijkstra's Algorithm: This is a single source algorithm, we need only know the source of the current journey. The algorithm would map the entire metro system and find the shortest paths to all other stations. This is efficient when the user of the application is trying to find the shortest path to a destination neighbouring two or more stations as the application would not have to re-compute the journeys.

After careful consideration we decided to implement Dijkstra's Algorithm. Due to the Metro Systems being relatively small (< 1000 elements) the computational requirements are modest enough to not inhibit or slow the user. This coupled with our knowledge of the algorithm were the overriding factors in our decision.

Dijkstra's Algorithm Performance:

Assigning letters to the mathematical notation as such: V is equal to the amount of vertices shown on the graph and E becomes the amount of Edges.

Using Big-O notation, we can assume that each vertex is checked V amount of times, equal to the amount of vertices on the graph. We can also assume that each edge is checked a maximum of 2 times, 1 check per node connected.

$O(V^2 + 2E)$

If we substitute a large value for V and E, the 2E in the formula would not affect the overall performance as proof is shown below.

` $O(100^2 + 2 * 100) = 10000 + 200$

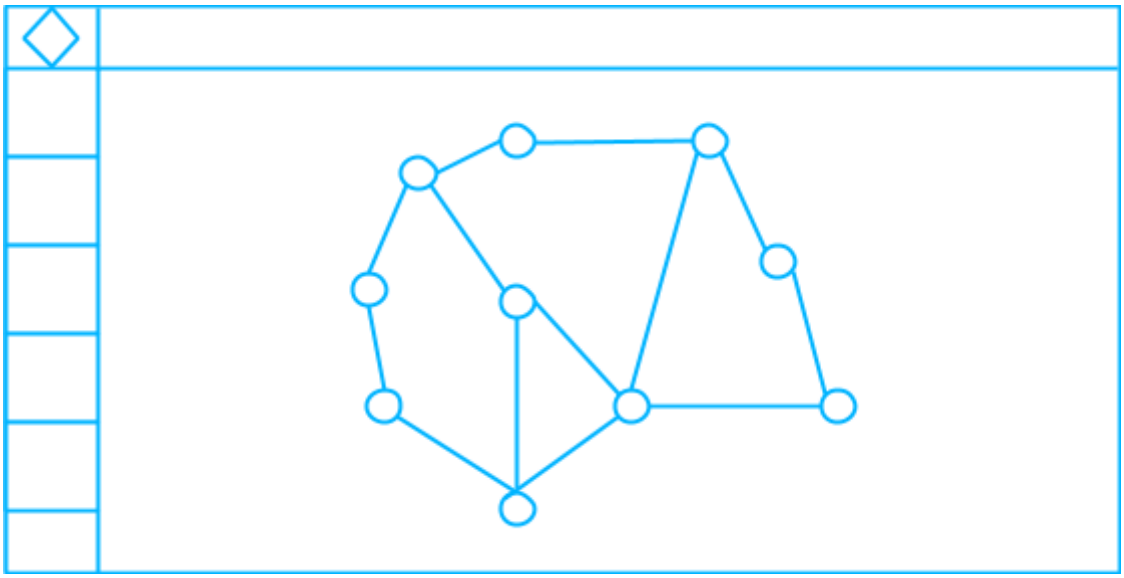
Design of the GUI

From a very early stage we had an unpolished idea of how the GUI would look and act, there were a few revisions but overall we came to following designs. We will implement 2 windows, separating the Design window and Journey Planner thus reducing the risks of an unauthorized user editing a system they should not have access to.

Here are a few design choices that will apply to most of the application:

- The use of light colours in the design removes the generally dark aspect of using a metro system, and as such, the use of white is a very influential design choice.
- Minimal images, we focussed primarily on typography and the use of text to convey information over graphics.
- The usage of shapes will be limited to squares and circles where squares represent stations that are on a single line and circles as changes.

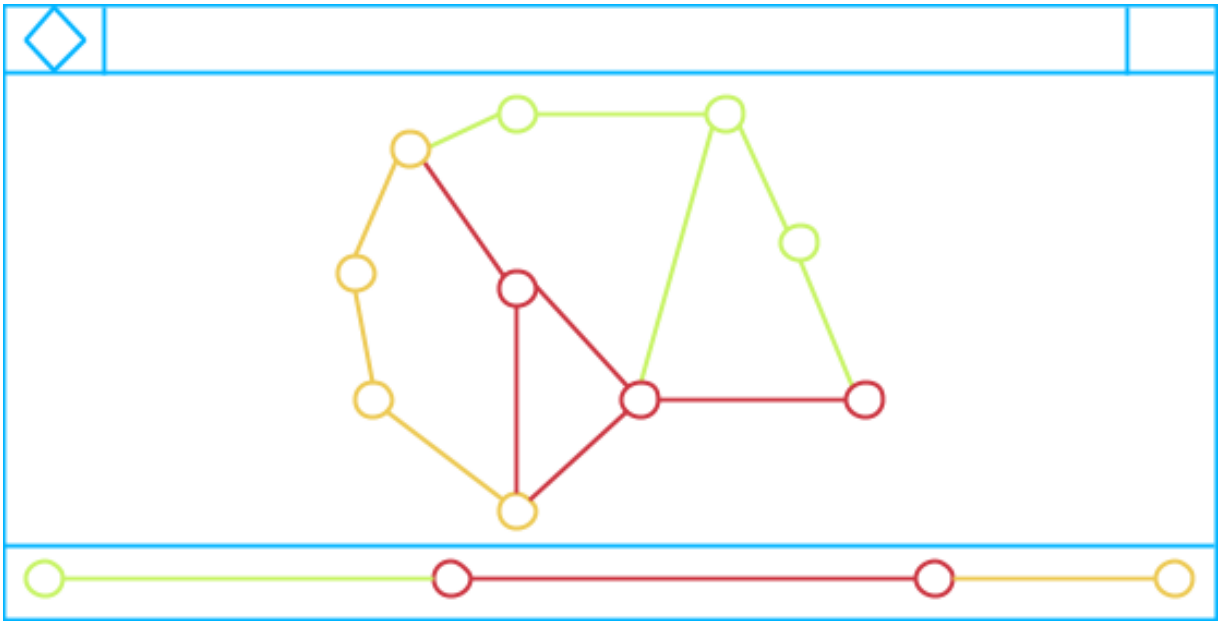
Creator Window:



As with the Journey Planner, the circles displayed in the centre of the window are what make up a metro system, the colours will change depending on the line on which they currently reside.

The squares displayed on the left of the image are contained within a sidebar and will be the main functionality of the Creator Window such as Adding new stations, adding new connections and various toggles such as the removing of items.

Journey Planner:



The elements in the centre of the window will be locked and, attempting to move these individually will instead move the system for better visibility.
The main component of the Journey planner is the bottom bar which shall appear when a station is selected as a source and will fill when a station is chosen as a destination. These lines will be divided equally depending on the amount of stations crossed on a single line and will display the time taken to cross these lines both individually and as a total.

Implementation of the model

As stated in the design, we will be implementing Dijkstra's algorithm in the Java programming language. We can start by determining the requirements for the algorithm to run, once the requirements have been ascertained we will move on to pseudo-code.

- Dijkstra's algorithm requires nodes and edges, translating this into java gives us an Array of nodes (The stations) and an Array of edges (The connections).
- It requires a source node that shall become the station the user will depart from.

The pseudo-code assigned to our design is as follows:
Colour codes: Green = Variable name, blue = keywords, red = numerical operator

```
function DijkstraAlgorithm(DijkstraMap map, Node source)
        define Array[Node] stations = get nodes from map
        define Array[Connection] connections = get connections from map
        define Array[Node] checked
        define Array[Node] unchecked
        define map<Node,Node> previous
        define map<Node,int> previousTime
        insert sourceNode into unchecked

        while unchecked is not empty:
                node station = node unchecked with smallest distance in previousTime not yet
                visited
                remove station from unchecked
                insert station into checked
                define List<Node> neighbours
                for each Connection connection in connections:
                        if (connection source is station)
                                insert connection into neighbours
                end for
                for each Node neighbour in neighbours:
                        define Number newCost = previousTime to station +
                                cost of connection between station and neighbour
                        if (previousTime to neighbour > newCost) :
                                set neighbour in previous to station
                                set neighbour in previousTime to newCost
                                insert neighbour into unchecked
                        end if
                end for
        end while
end function
```

After reviewing the Pseudo-code we resolved to work on making this model as expandable as possible, this would mean taking into consideration the use of Java Generics and taking a look at persistent storage over "hard-coded" nodes and connections.

Generics now make up a very big portion of our Dijkstra model:

- Nodes can be any object by using the DijkstraMap<V> where V is the type of object.
- The DijkstraGraph and DijkstraConnection are now interfaces, in java this allows any object to extend upon and increase functionality of the object such as allowing for Graphical visualizations.
- Custom Lists, Maps and Sets were added to prove that we are able to create a data structure that is expandable, while these sets work it would still be beneficial to use the java default implementations of the Collections due to the Java employees having spent time developing and ensuring that users get the most efficiency when creating said Collections.

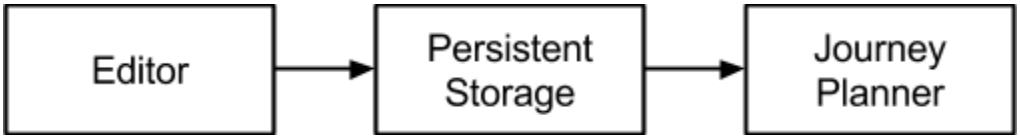
The Lists Interface: *"A List is an ordered Collection (sometimes called a sequence). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:*

- *Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.*
- *Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.*
- *Iteration — extends Iterator semantics to take advantage of the list's sequential nature. The listIterator methods provide this behavior.*
- *Range-view — The sublist method performs arbitrary range operations on the list."*
  (See Bibliography)

The Maps Interface: *"A Map is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical function abstraction. The Map interface includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty), bulk operations (such as putAll and clear), and collection views (such as keySet, entrySet, and values)." (See Bibliography)*

The Sets Interface: *"A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements." (See Bibliography)*

In order to avoid “hard-coding” the stations and connections, we regressed and looked at the system from another perspective agreeing upon a development process:



This process ensures that the structure generated via the code is correct and in doing so, also reduces the time needed to create them. Creating the editor first, completes all of our objectives relating to the creation of the model, it offers a Simplistic, Elegant, Expandable and Manageable view of the raw data required to create and view a Metro Graph.

MetroIO

The first thing created was the Input/Output class for the persistent storage, starting with this class also assumes we know all the data required which is bothersome, should we after some time decide that we would like to add more details to the Input/Output and usually requires some manual tweaking.

Before we can implement the code for this Input/Output, it would first be preferred to create a file Structure to contain any files we may need to create per system which is shown below:

```
+---- PSTMetro.jar
+---- PSTMetro/
+-------- Systems/
+------------ Prague/
+---------------- nodes.txt
+---------------- connections.txt
+---------------- images.txt
+---------------- lines.txt
+---------------- settings.txt
+---------------- Images/
+-------- Resources/
```

```
public static MetroMap readMetroLines(MetroMap map, File file) {
    try (BufferedReader br = new BufferedReader(new FileReader(file))) {
        String line;
        while ((line = br.readLine()) != null) {
            if (line.startsWith("#") || !line.contains("\t")) {
                continue;
            }
            try {
                map.addLine(line);
            } catch (Exception e) {
                System.err.println("Couldn't read line: " + line);
            }
        }
    } catch (IOException ex) {
        Logger.getLogger(MetroIO.class.getName()).log(Level.SEVERE, null, ex);
    }
    return map;
}
```

Shown above is one of the IO methods used, this method adds the MetroLine to the MetroMap and then returns the MetroMap for convenience. All other loading methods for MetroNodes and connections work mostly the same way in that the class associated with the method (In this case MetroLine) handles the way it reads the line.

Dijkstra's Algorithm

Now that have all of the required nodes and connections we can start to create interfaces that will be used as containers for the Dijkstra structure.

```
public interface DijkstraConnection<V> {
   public V getSource();
   public V getDestination();
   public int getCost();
}
```

```
public interface DijkstraGraph<V> {
   public DijkstraConnection<V>[] getConnections();
   public V[] getNodes();
}
```

You will notice that there is no interface for a Dijkstra node and this is because the node requires no real data, allowing us to use generics. The user is able to choose what type of node they would like to use. In our MetroNode, we require only visual data such as X, Y, The line they are on… etc.

Analyzing the Pseudo-code we are able to create real java code:

this.nodes = map.getNodes();
this.connections = map.getConnections();
this.source = source;
this.previousNodes = new HashMap<>();
this.previousTimes = new HashMap<>();
this.uncheckedNodes = new HashSet<>();
this.checkedNodes = new HashSet<>();
	define Array[Node] checked
	define Array[Node] unchecked
	define map<Node,Node> previous
	define map<Node,int>previousTime
	insert sourceNode into unchecked

private V getClosest(Set<V> nodes) {
V closest = null;
for (V node : nodes) {
if (closest == null) {
closest = node;
} else {
if (getFastestPreviousTime(node) < getFastestPreviousTime(closest)) {
closest = node;
}
}
}
return closest;
}
	node station = node unchecked with smallest distance in previousTime not yet visited

checkedNodes.add(node); uncheckedNodes.remove(node);	remove station from unchecked

	insert station into checked

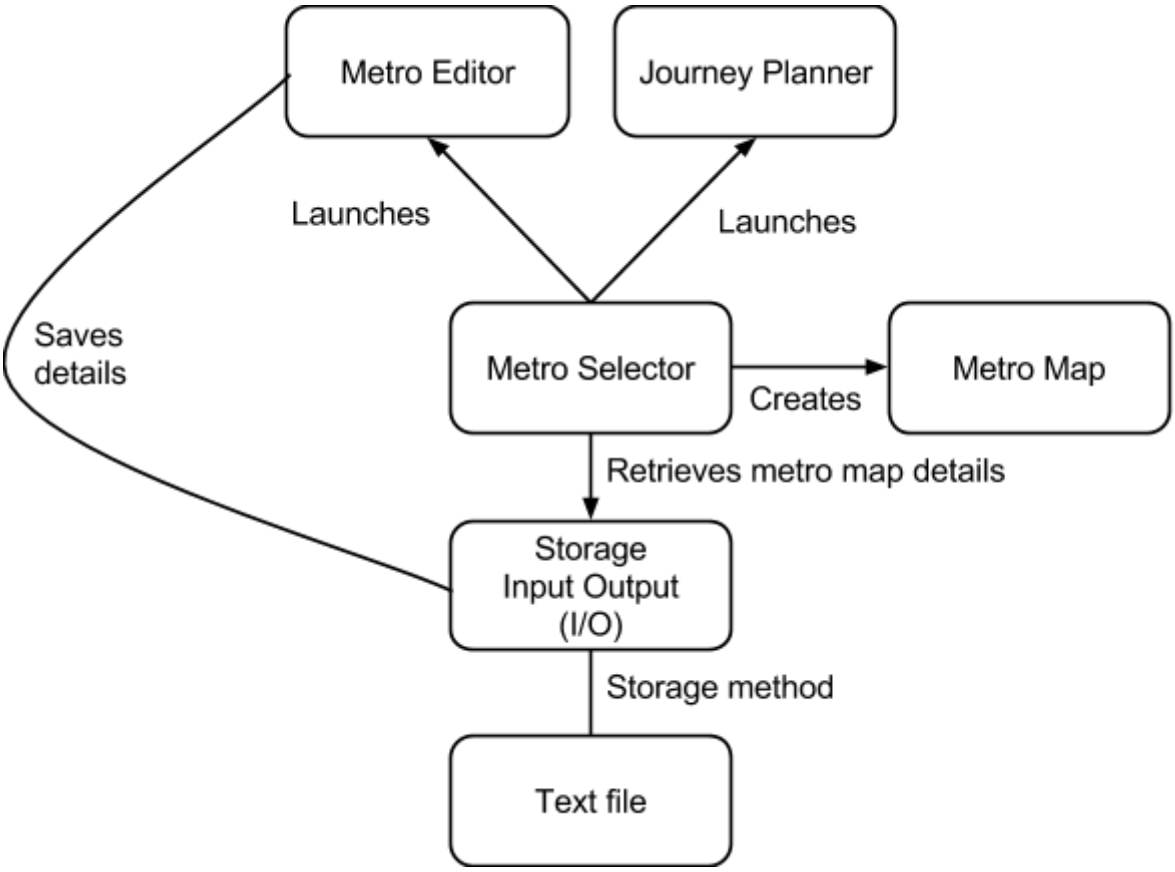
protected List<V> getUncheckedConnections(V source) {
List<V> l = new ArrayList<>();
for (DijkstraConnection<V> conn : connections) {
if (conn.getSource() == source && !checkedNodes.contains(conn.getDestination())) {
l.add(conn.getDestination());
}
}
return l;
}
	define List<Node> neighbours
	for each Connection connection in connections:
	if (connection source is station)
	insert connection into neighbours
	end for

protected int getFastestPreviousTime(V node) {
if (previousTimes.containsKey(node)) {
return previousTimes.get(node);
} else {
return Integer.MAX_VALUE;
}
}
protected int getCostOfConnection(V node, V destination) {
for (DijkstraConnection connection : connections) {
if (connection.getSource().equals(node) && connection.getDestination().equals(destination)) {
return connection.getCost();
}
}
throw new RuntimeException("No Connection found - At all...");
}
	define Number newCost = previousTime to station + cost of connection between station and neighbour

for (V currentNode : l) {
if (getFastestPreviousTime(currentNode) > getFastestPreviousTime(node) + getCostOfConnection(node, currentNode)) {
previousNodes.put(currentNode, node);
previousTimes.put(currentNode, getFastestPreviousTime(node) + getCostOfConnection(node, currentNode));
uncheckedNodes.add(currentNode);
}
}
	for each Node neighbour in neighbours:
	define Number newCost = previousTime to station + cost of connection between station and neighbour
	if (previousTime to neighbour > newCost) :
	set neighbour in previous to station
	set neighbour in previousTime to newCost
	insert neighbour into unchecked
	end if
	end for

Once this code is completed, we can analyse the system as a whole and now work on the GUI with the more technical code out of sight and the requirements for this out of mind.

The following Graph is gives us a clear picture on how the system should behave in relation to the saving and loading of data.

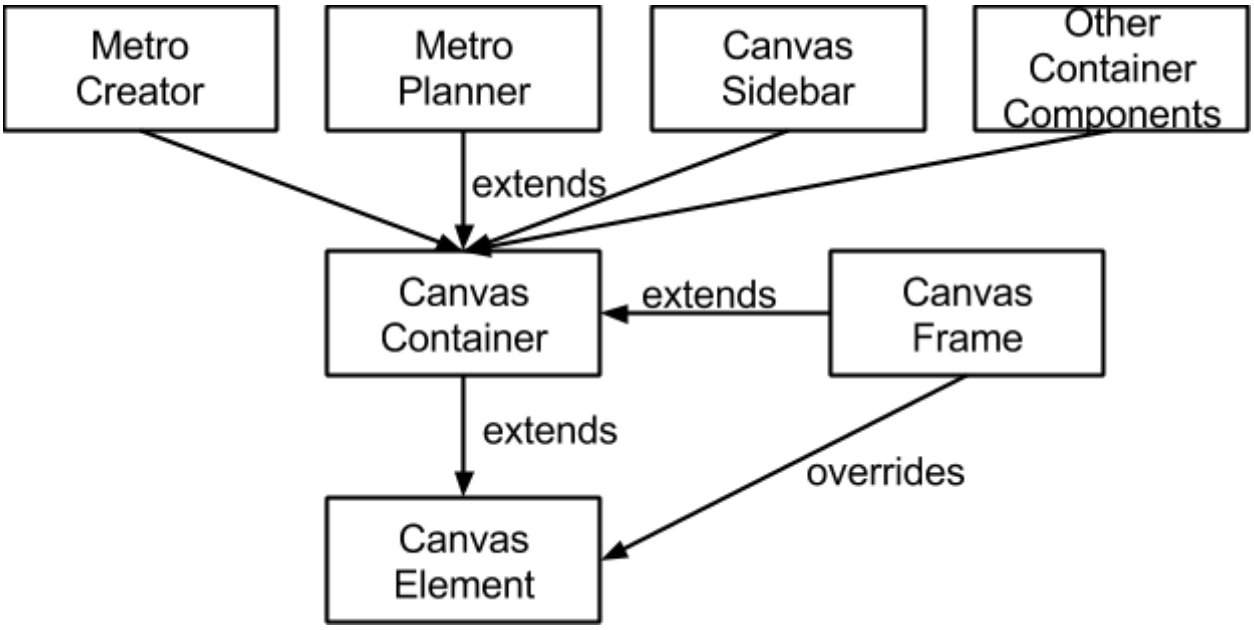


Implementation of the GUI

Canvas GUI

As we know, Java has an inbuilt GUI Library named swing and, whilst the GUI could have been implemented in Swing, we decided to create our own Library based on the Java AWT Canvas. In doing so we were able to understand component systems and the use of different patterns to create an absolutely positioned interface.

This also allows much more customizability in the interface to the same extent of the Java Graphics draw methods, this allows the Animation and fading of components that would not have been directly accessible in Swing without other heavy duty components.



The above design explains the use of the Canvas Element component at its core. Where the CanvasFrame contains a JFrame to do the repainting. The following code is taken from the CanvasElement class and explains the recursion element of the repaint method.

```
protected CanvasContainer parent;
public void repaint() {
        parent.repaintElement(this);
}
    public void repaintAll() {
        parent.repaintAll();
 }
```

This repaint will continue up the chain until it reaches a class that overrides the repaint method such as CanvasWindow, in this class the repaint method is called and the appropriate element is marked as "dirty". When an element is marked dirty, it will be repainted on the next canvas repaint call.

```
@Override
public void repaint() {
	repaintElement(this);
}
@Override
public void repaintAll() {
	markDirty(elements.toArray(new CanvasElement[elements.size()]));
}
public void markDirty(CanvasElement... elements) {
	dirtyQueue.addAll(Arrays.asList(elements));
	canvas.repaint();
   }
```

We separated the 2 projects to ensure that the code is expandable in that it will work for any GUI project we may have. Unfortunately due to time restraints, this code is not commented but will still be provided in the work along with the source and, as a library.

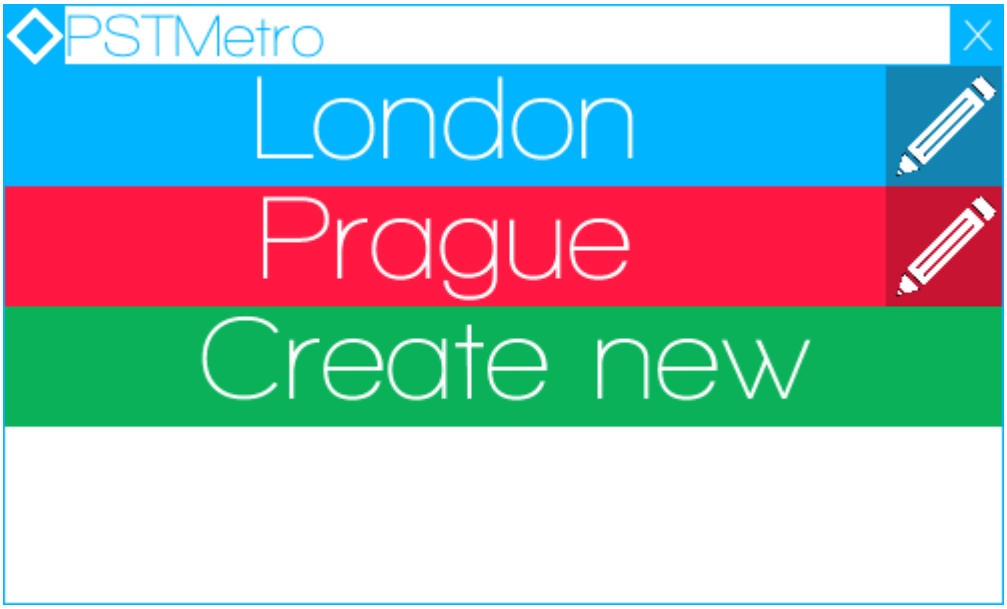
Metro GUI

Each of the Metro GUIs (MetroCreator and MetroPlanner) have the exact same requirements of a Dijkstra map except they contain extra information and methods such as the usage of a grid.

As stated before, our Dijkstra's algorithm is able to integrate any type of Node provided the objects used are distinguishable with Hash codes as such we created a new object named MetroNode, this object overrides the hashCode ensuring the node is unique so long as the name is different. All components extend off of the canvas.

User and help documentation

Metro Selector:



Upon starting the program, the user will be welcomed with the Metro Selector. From here the user will be able to Select, Edit and Create a new Metro.

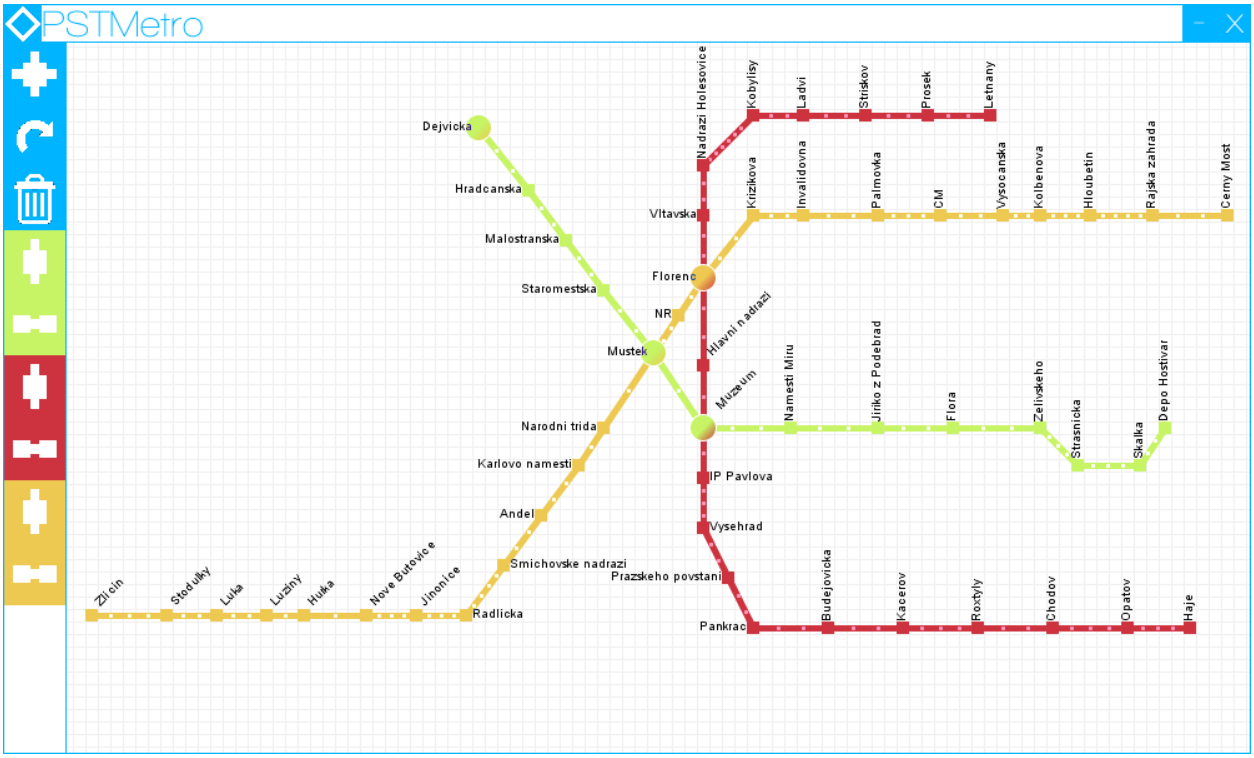
The select function will let the user to view and plan a journey via the Metro Journey Planner, to open this window simply click the desired Metro name.
The edit function allows for the editing of Metro Maps, to access this window, click the pencil icon associated with the Metro name.
The create new function enables the user to create a new Metro map with the desired name, click the Create New button, a Caret will appear indicating that you are able to type the name of your chosen Metro, to complete this action, the user must press enter on their keyboard.

Journey Planner:

To begin planning your journey simply click on the departure station. The station will appear in bottom left of the Journey page indicating that a station has been chosen. To complete the planning of a journey, the user must then click another station. The trip will form in the journey page as shown below.

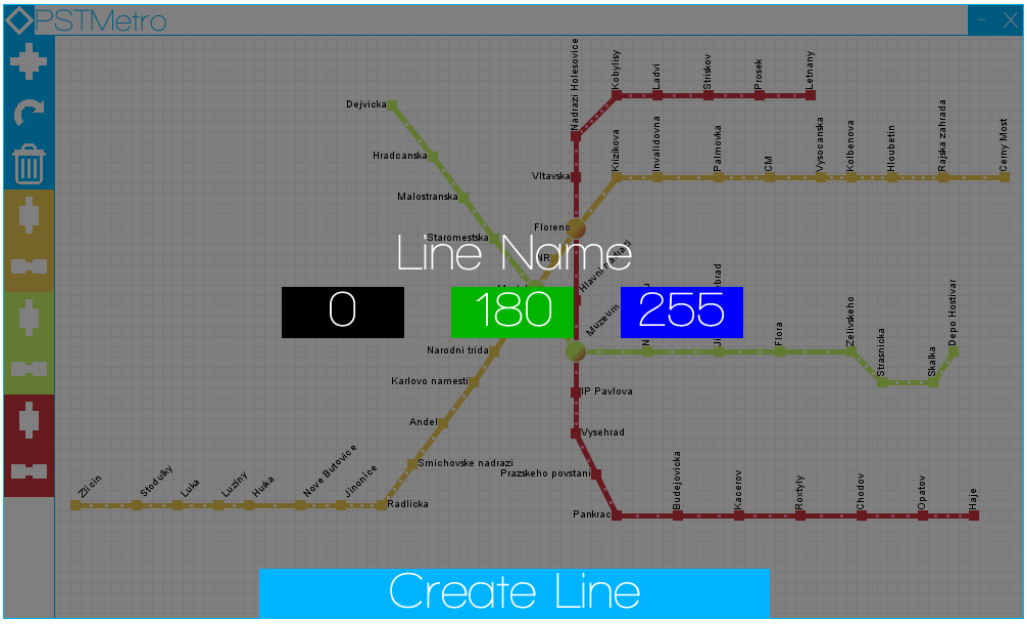
The Journey page will indicate the amount of changes required, along with times and stops per line. A colour change symbolizes a transition to another Metro Line. To select another journey, simply drag the journey page down.

Metro Creator



From here the user will be able to create and remove Lines, stations and connections between them.

This button allows the user to create a new line and will overlay the Create a Line window as shown below:



Upon clicking “Line Name” A caret will display, indicating that the user can then type the name associated with the Line they are creating. The 3 boxes below are the Red, Green and Blue values associated with the colour of the line they are creating.
Clicking the “Create Line” button will add the Node and Connection line buttons to the sidebar of the Creator Window, the overlay shall disappear and the user can continue to edit their metro.


This button toggles the rotation of the text when clicking a station.


This button toggles the deletion of stations and connections when they are clicked. If you wish to stop deleting connections and stations, click this button once again. If this button is pressed, it is also possible to delete lines by clicking button below.


The button allows the addition of nodes associated with the colour of the background. Simply click this button, click where you would like the station to be placed and type the name of the station, followed by enter.


This button allows the connection of 2 stations whose line will be associated with the colour. By clicking on the 2 desired stations a connection will be formed. If you have clicked a single station and do not wish to create a connection, simply click the station again and the connection will not be created.

Test methodology and analysis of results

Test Item:	Expected Outcome:	Actual Outcome:	Priority:	Resolved?
Create New Metro System	User is able to input the name of the desired metro system. New metro system will then be added to the System Selector.	New metro system is created and is added to the System Selector. See Appendix 1.1,1.2 p26,p27	-	-
Editor System Button	User clicks on the editor button. The System Editor page will then appear allowing the user to edit the selected Metro System.	System Editor Page appeared ahead of the System Selector. See Appendix 2 p27	-	-
Create New Line Button	Upon clicking this button the user will be able to create a new metro line. They will be able to give the line a name and a specific colour by inputting the RGB colour.	The create a new line window appeared. See Appendix 2.1 p28	-	-
Edit Line Name (Create)	User is able to input the new line name.	New line name changed successfully. See Appendix 2.2 p28	-	-
Edit Red (Create)	User is able to input numerical value of the colour Red as part of the RGB.	Numerical value for Red was set. See Appendix 2.3 p28	-	-
Edit Green (Create)	User is able to input numerical value of	Numerical value for Green was set.	-	-

	the colour Green as part of the RGB.	See Appendix 2.4 p29		
Edit Blue (Create)	User is able to input numerical value of the colour Blue as part of the RGB	Numerical value for Blue was set. See Appendix 2.5 p30	-	-
Create New Line (Create)	The user will press the Create New Line button, and the new line will appear sidebar of the editor window.	The new line was created. See Appendix 2.6 p30	-	-
Add Station to Map	Once the new line is created then the user will be able to click on the station icon. From here the user will need to click on the window to place the station.	Station was added to the map. See Appendix 2.7 p31	-	-
Set Station Name	After putting the station into the editor window, the user will then be required to set the Station Name. Due to validation, the Station Name is required.	The name of the Station was set. See Appendix 2.7 p31	-	-
Create Connection	The user will press the create connection button. They will then select the first station and then the destination station.	Connection was created. See Appendix 2.8 p31	-	-
Toggle Delete	The user will press the Delete button. This button will then change to a red to	Toggle Delete was linked with toggle Rotate.	1	Resolved

	show that it has been selected.	See Appendix 2.9 p32		
Remove Station	User will be able to press remove a station simply by pressing on the station directly.	Station was removed. See Appendix 2.13 p34	-	-
Remove Connection	User will be able to press remove a connection simply by pressing on the connection directly.	Connection was removed. See Appendix 2.12 p33	2	Resolved
Toggle Rotate Text	The user will press the rotate button. This button will then change to a different shade of blue to show that it has been selected.	Toggle Delete was linked with toggle Rotate. See Appendix 2.10 p32	1	Resolved
Rotate Station Name	User will be able to press on a station and the text will rotate clockwise.	Station Name was rotated. See Appendix 2.11 p33	-	-
Move Station	The user will press and hold on a station, and simply move the mouse to drag the station.	Station was moved. See Appendix 2.14 p34	-	-
Drag System (Editor Window)	The user will press and hold on a blank area, and simply move the mouse to drag all components in the system.	System was dragged. See Appendix 2.15 p35	-	-
Drag Sidebar	If there are too many lines that have been created then the user will be able to press	Sidebar was able to be dragged up and down.	-	-

	and hold on the sidebar and push the mouse/touchpad upwards.			
Minimise Creator Window	The user will press on the minimise button and the window will be minimised.	Creator Window was minimised	-	-
Close Creator Window	The user will press on the close button and the window will be close.	Creator Window was closed.	-	-
Open System Button	User will press on the system they wish to open to plan their journey.	Journey Planner for selected system was opened. See Appendix 3 p35	-	-
Drag System (Journey Planner)	The user will press and hold on a blank area, and simply move the mouse to drag all components in the system.	System was dragged.	-	-
Select Source Station	The user will select the source station.	Source station was selected. The selected station appeared in the journey planner at the bottom. See Appendix 3 p35	-	-
Select Destination Station	After selecting the source station the user will then select the destination station. The journey planner will then appear.	Destination station was selected. The selected station appeared in the journey planner and a journey was calculated.	-	-

		See Appendix 3.1 p36		
Drag down Journey Planner	After the destination station is selected, a journey planner will appear in front of the main Journey Planner Window. The user is able to remove the journey planner from sight by dragging the Planner down, as instructed.	Journey planner was dragged down and disappeared from plain sight.	-	-
Minimise Journey Planner	The user will press on the minimise button and the window will be minimised.	Journey Planner window was minimised.	-	-
Close Journey Planner	The user will press on the close button and the window will be close.	Journey Planner window was closed.	-	-

Testing our system would allow us as developers to see if each sub-system is working as intended. It is essential that all components are thoroughly checked to resolve any issues that arise.
After carrying out extensive testing on our system, we encountered a few problems with the editor page. Upon finding these errors, we prioritised them in order to solve the most important first.
The errors were due to miscommunication between different group members coding the side bar and the metro creator. Apart from these errors, everything ran smoothly and all tests were completed successfully without error.

Evaluation and Reflection

Upon starting this project we had 4 goals in mind: we would make our Simplistic, Elegant, Manageable and Extendible. While not necessarily easy to achieve, we believed these goals were accomplished due to several aspects of our design phase:

- Careful planning, this meant taking into consideration many situations involving user error and system specifics such as OS, platform and Metro features.
- Interactivity, each member of the development team had several ideas on the way users could interact and manage the GUI, this resulted in a user friendly interface.
- Styling, we implemented the Metro design language, a typography based scheme that both diminished our workload and gave the application a sense of identity using rich colours complimenting one-another.
- Structure, perhaps the key element in creating an application with minimal alterations required during the implementation phase.

Bringing these core objectives into the implementation stage enabled seamless creation of the GUI, model and data structure required for the application, both speeding up the time required to achieve our goals and minimizing the risk of encountering errors. Having taken these steps, our project was rendered more reliable and maintainable as any errors faced were quickly eradicated.

As young developers we were keen to implement many ideas that were perhaps out of our reach due to many different reasons, such as time constraints and the lack of manpower. Were this not the case, here are some enhancements that we would have liked to accomplish:

- Make a website dedicated to the journey planner
- Password protection to allow customers to only plan journeys not edit
- Tourist information on the surrounding area of each station situated in the city
- Sponsorship & Advertising
- Porting the current desktop application onto a mobile platform allowing journey planning on the go.
- Usage of the Web API to enable real-time train information and train times
- Additional features to the Metro API allowing for engineering works and other unplanned events that can occur in the real world.

Overall, the development on the application helped each member of group achieve many goals that were not an option before the start of this coursework, it inspired creativity and has shown that teamwork is a vital component in the IT industry. The following statements were written by each member giving a basic overview on their thoughts of the development process, implementation and design of the system, as well as what they feel they have achieved.

Thomas Nairn

The process in which the application was created has given me great insight into some of the more time consuming aspects of the java programming language, this is not a bad thing and I now feel as though I accomplished a great deal. The Canvas GUI Project ,separate to the Metro Planner project, helped me learn certain patterns that I had not yet grasped and will play a vital role on my portfolio.
My design skills have always been somewhat lacking and working with individuals more design oriented than me has taught me a little about colour harmony and interactivity between applications.

Overall, I believe the project was an overwhelming success have completed tasks well beyond that which we originally set out to do and I would gladly work with such like-minded individuals once again.

Pascal Meyer

I have no qualms in saying that my mathematical ability is not the strongest, however upon completion of this report and the software program, I now believe that I have a more in depth understanding of Dijkstra's Algorithm as well as how to implement multiple data structures in java code. This will not go to waste as we as a group are in the planning stages of home projects, in which I will be able to fully show the extent to which my skills have improved.
As a group we unanimously decided at the beginning of this project, that we wanted to create something unique, to stand out from the all the other completed projects. The perfect way to do this was to create a custom GUI. I feel we both accomplished and succeeded in what we set out to achieve.

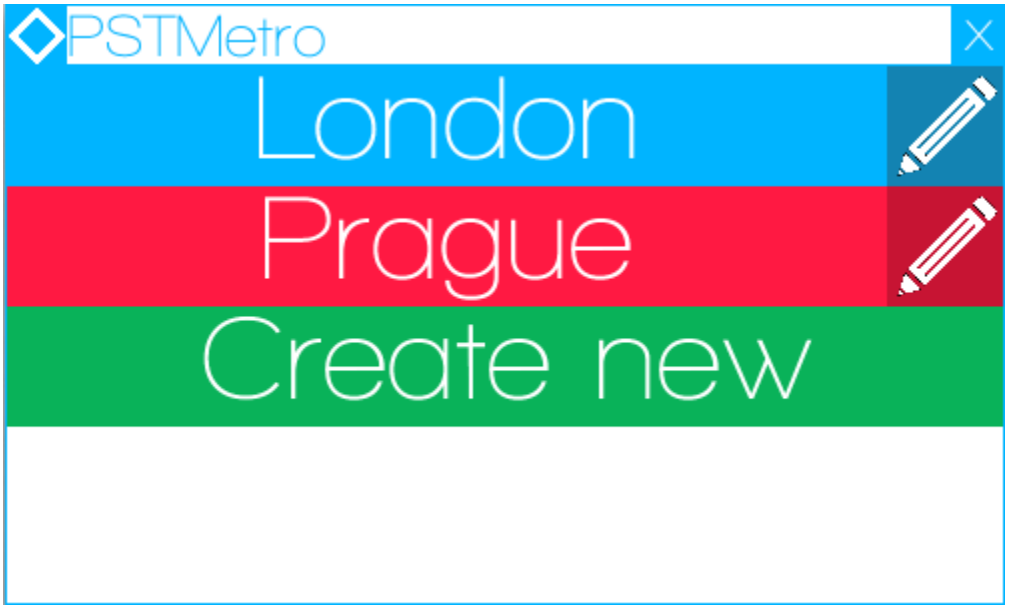
Sean Rozario

Upon first glance, this task seemed straight forward, and at parts too simple for a year 2 student. I was surprisingly mistaken. This task encouraged me to think logically and analytically to solve different tasks and to take into consideration my group members opinions. My main strength is mathematics, I assisted my group members in understanding the concept of Dijkstra's Algorithm and, in turn, my colleagues helped me in areas in which I was struggling in. I wish to improve my knowledge in applying common java techniques into fully working models of complex tasks.
Without my team-mates, this project would not be possible, we used our strengths to complement each other's weaknesses. I would happily work with my group again.

Task Pro-forma

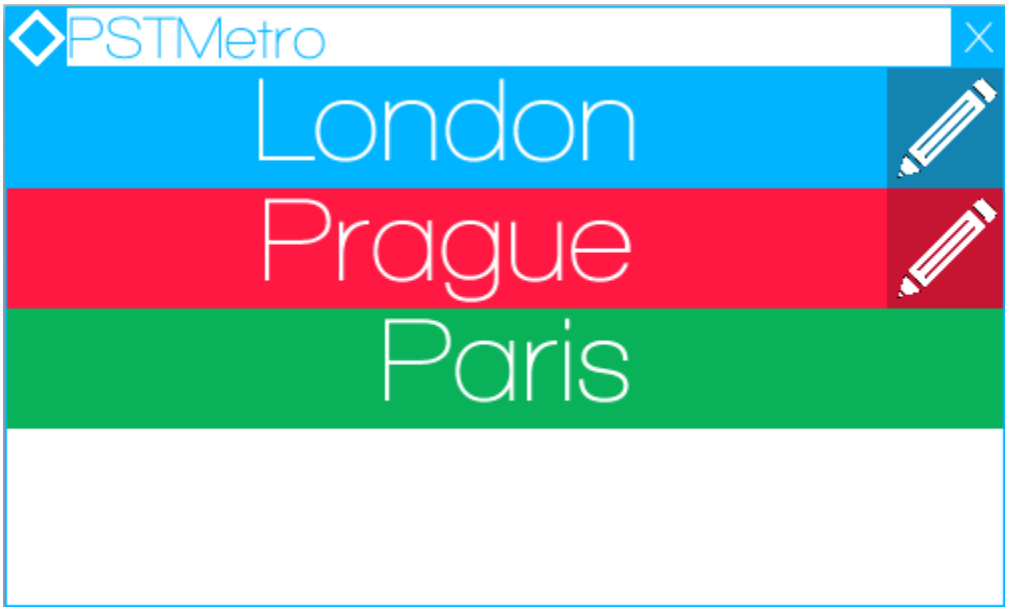
Name	Contribution
Thomas Nairn	100%
Pascal Meyer	100%
Sean Rozario	100%

Appendix 1



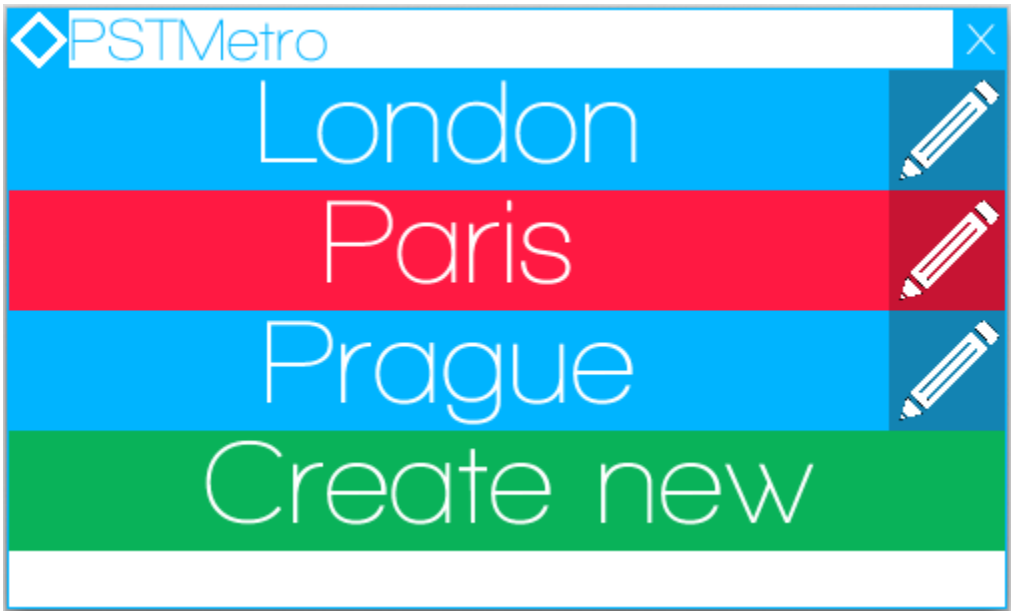
Screenshot showing the system selector window.

Appendix 1.1



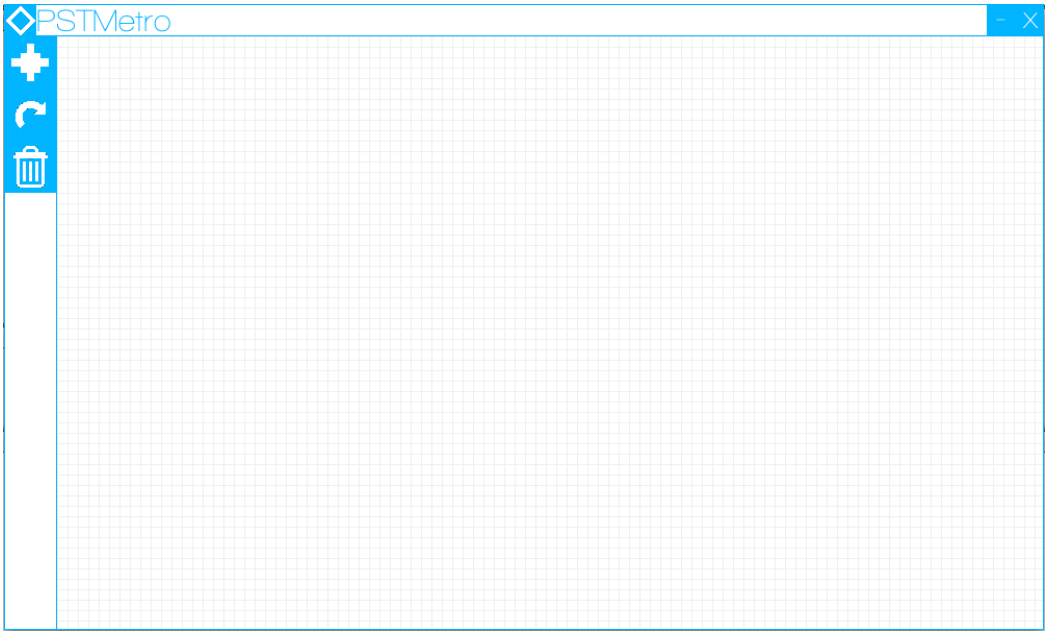
Screenshot showing new system being created.

Appendix 1.2



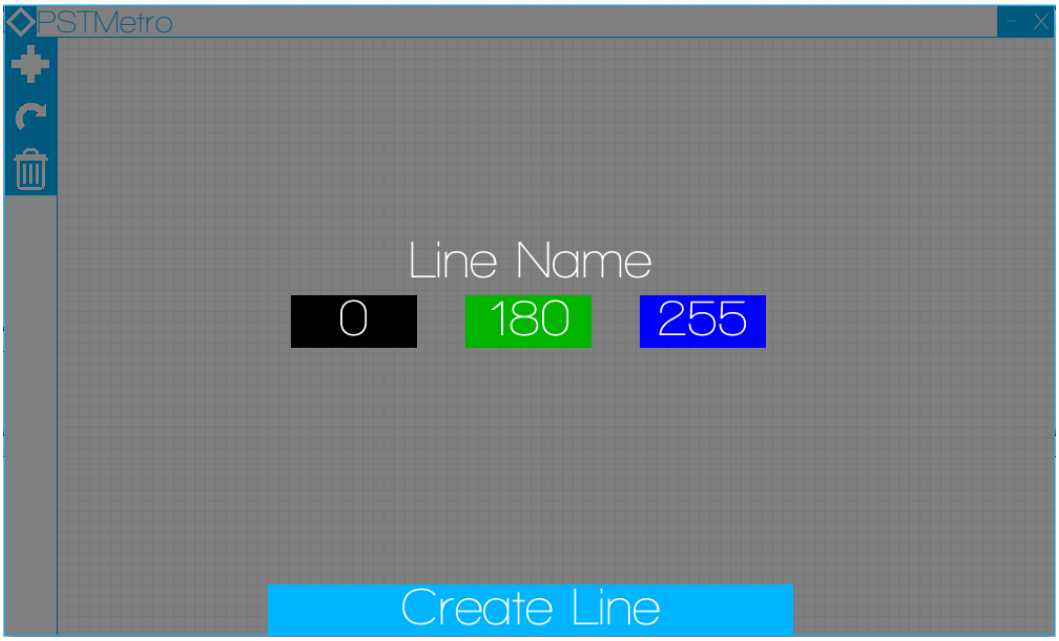
Screenshot showing that the new system has been created.

Appendix 2



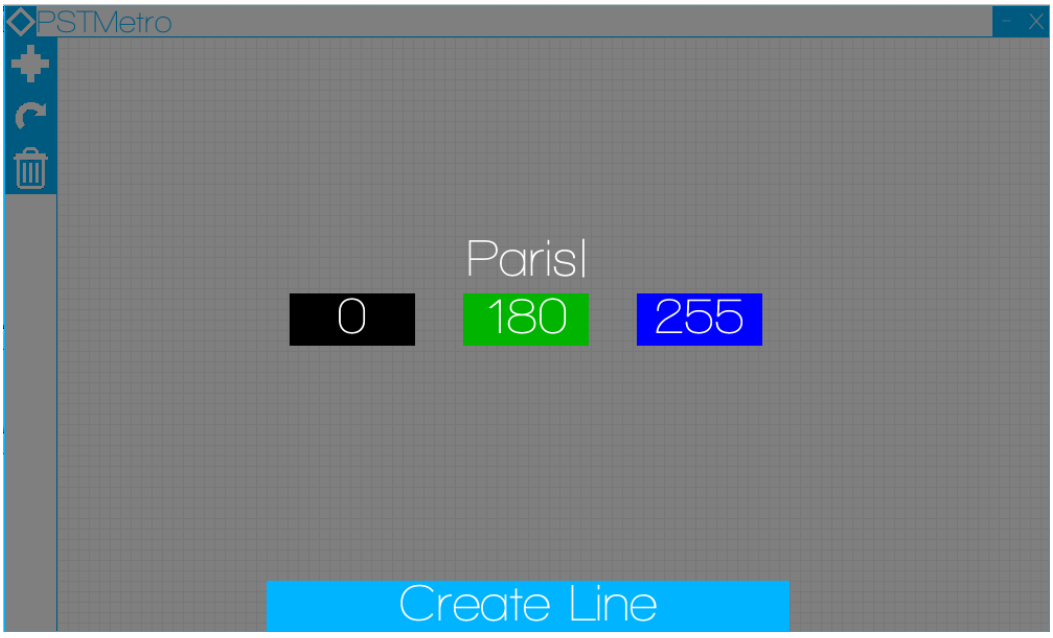
Screenshot showing the Metro Creator/Editor window.

Appendix 2.1



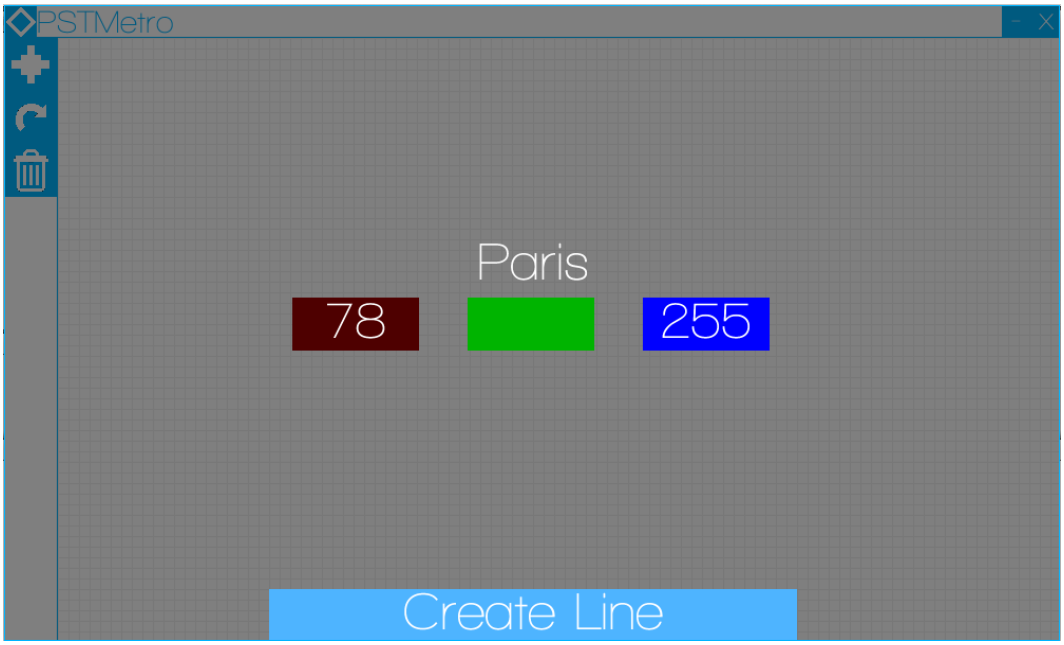
Screenshot showing the create a new line window.

Appendix 2.2



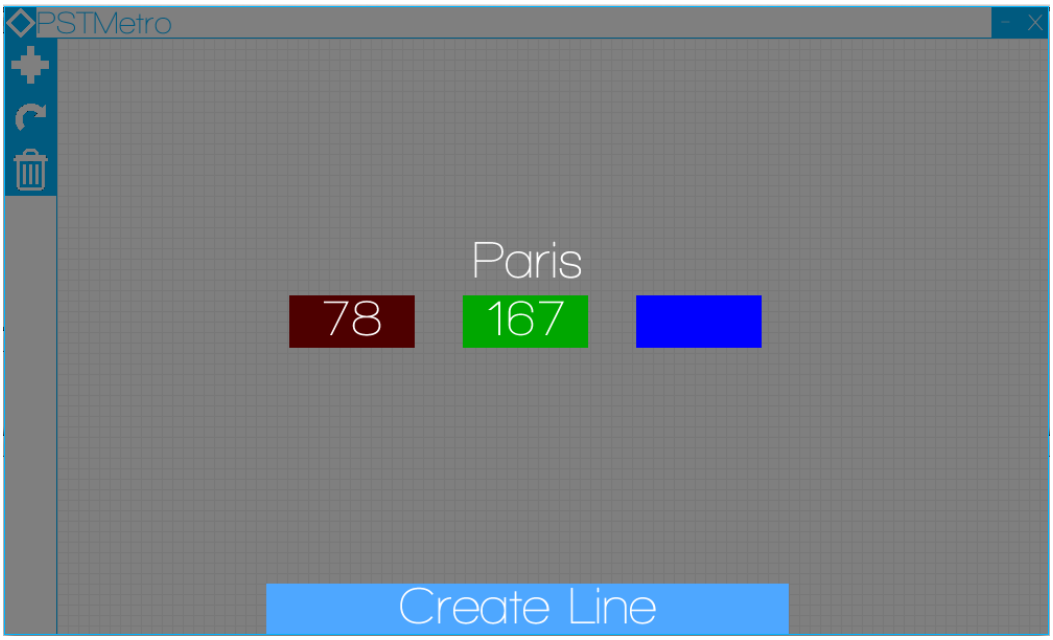
Screenshot showing the line name is being set to “Paris”.

Appendix 2.3



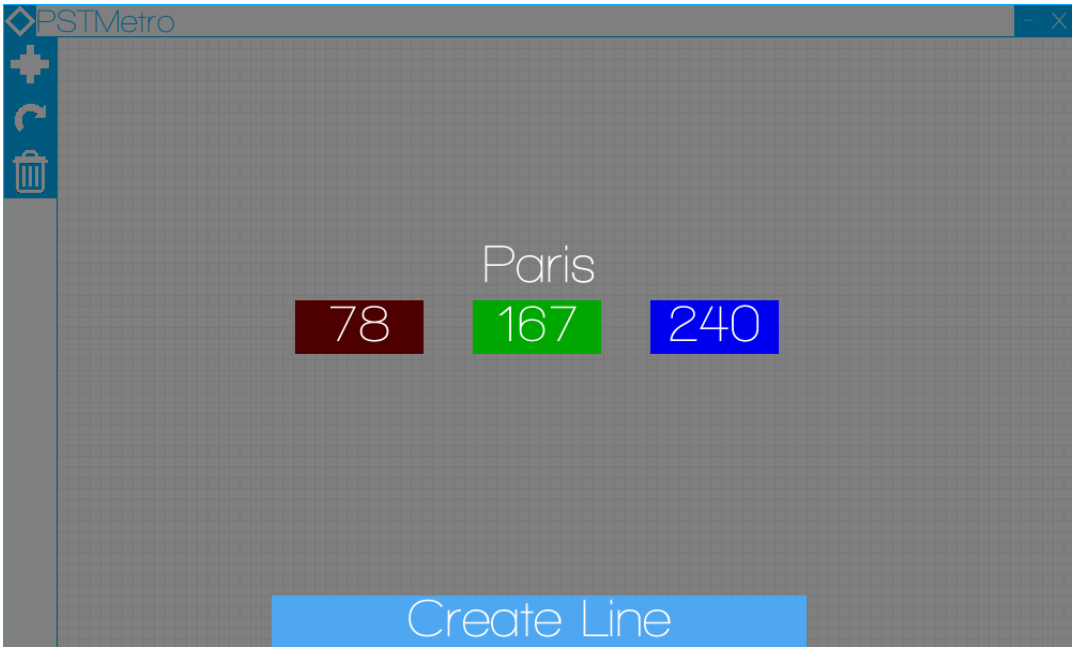
Screenshot showing the gradient of red being set.

Appendix 2.4



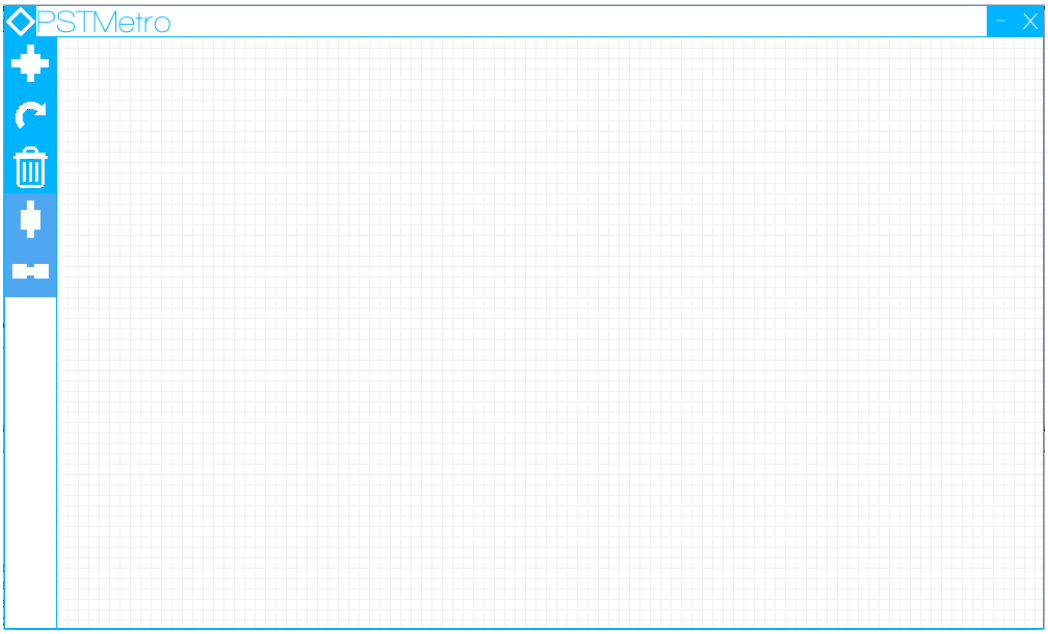
Screenshot showing the gradient of green being set.

Appendix 2.5



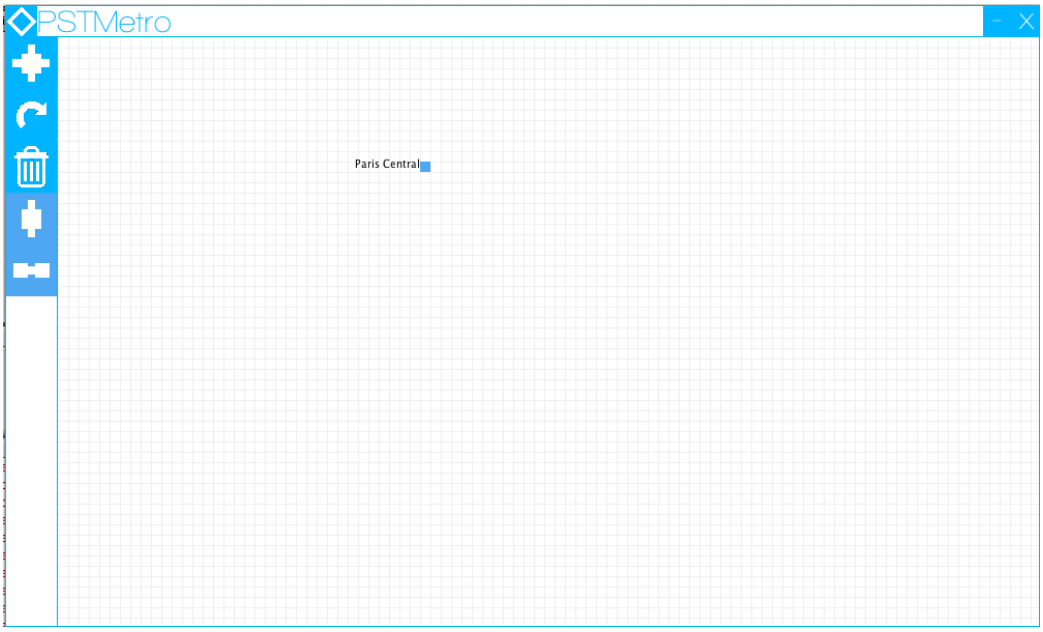
Screenshot showing the gradient of blue being set.

Appendix 2.6



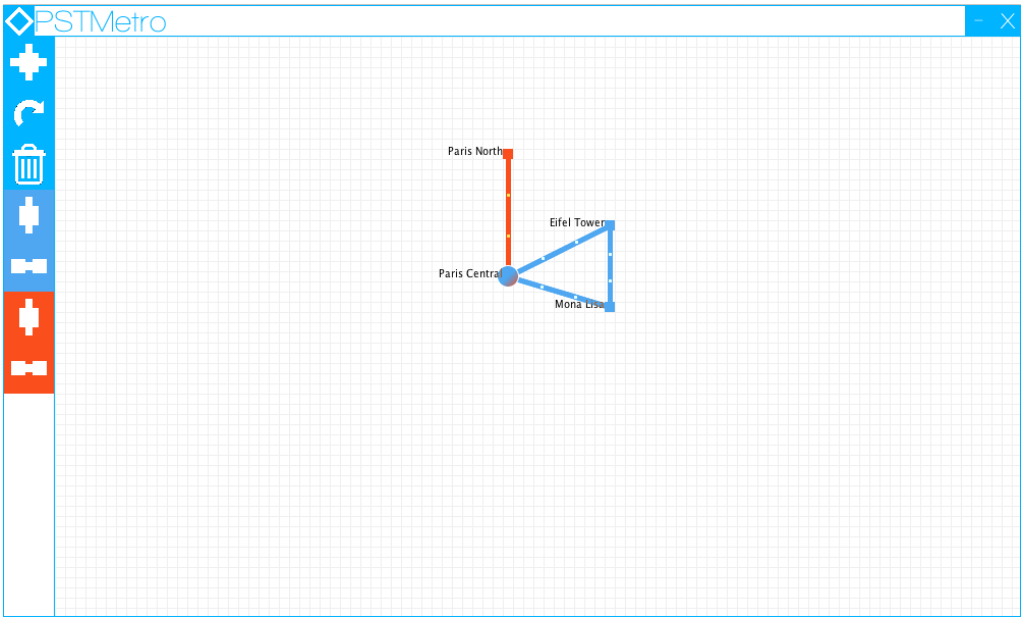
Screenshot showing the new line that has been created. The new line is situated in the side bar.

Appendix 2.7



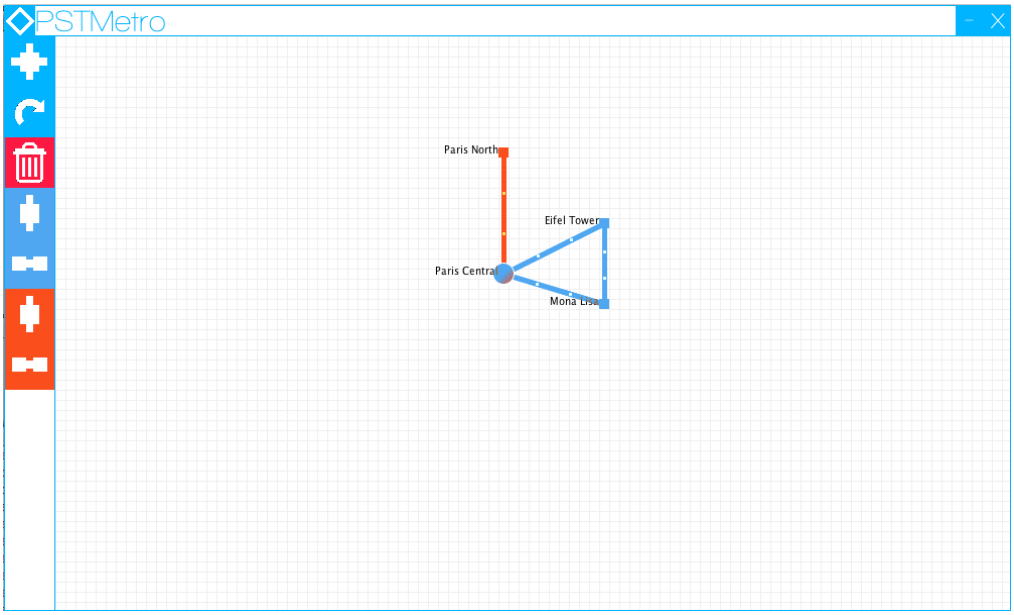
Screenshot showing a new station/ node being added to the map/grid. The text has been set for the new station.

Appendix 2.8



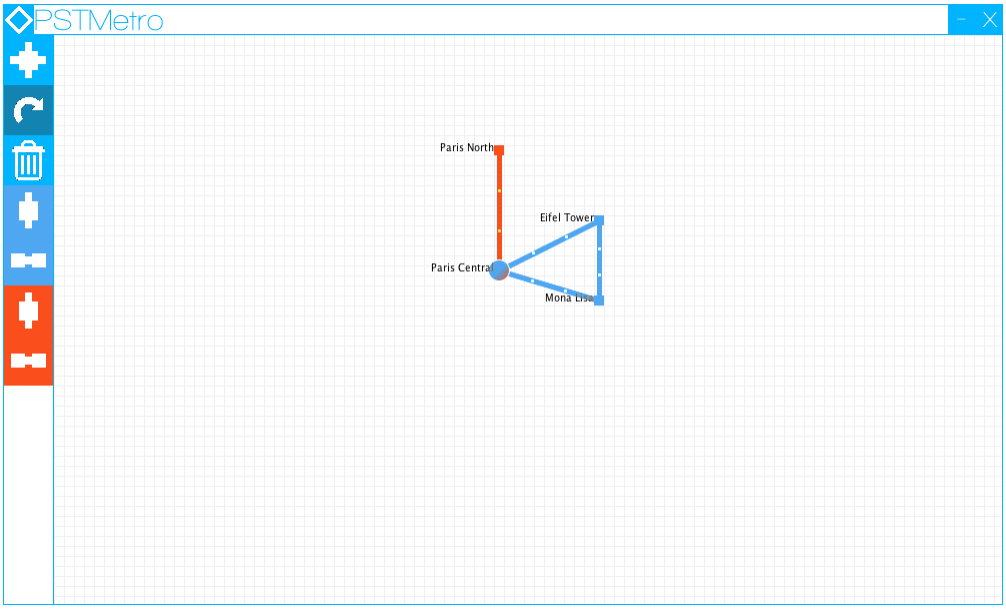
Screenshot showing new connections between the added stations.

Appendix 2.9



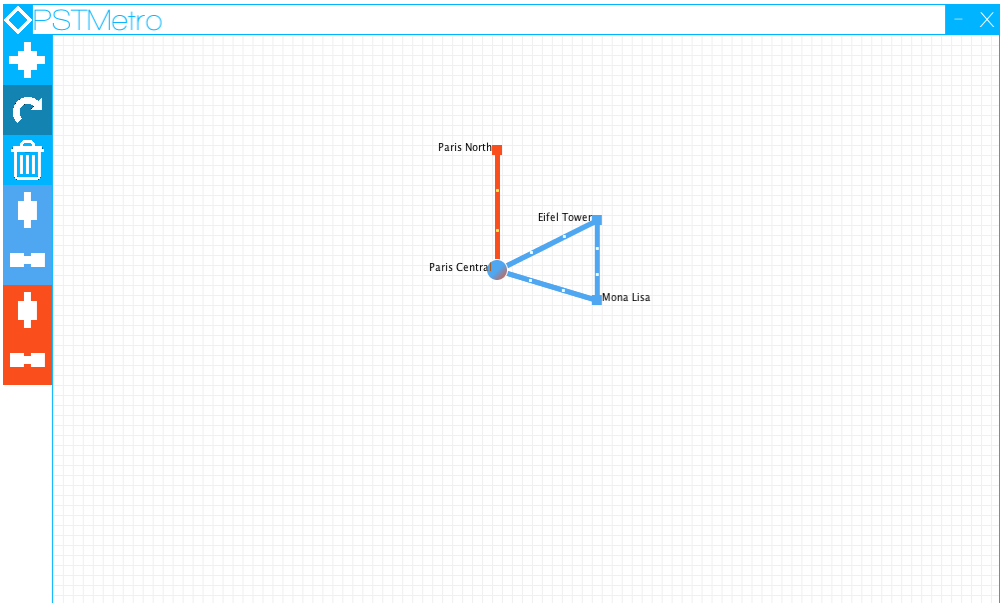
Screenshot showing the toggle delete button.

Appendix 2.10



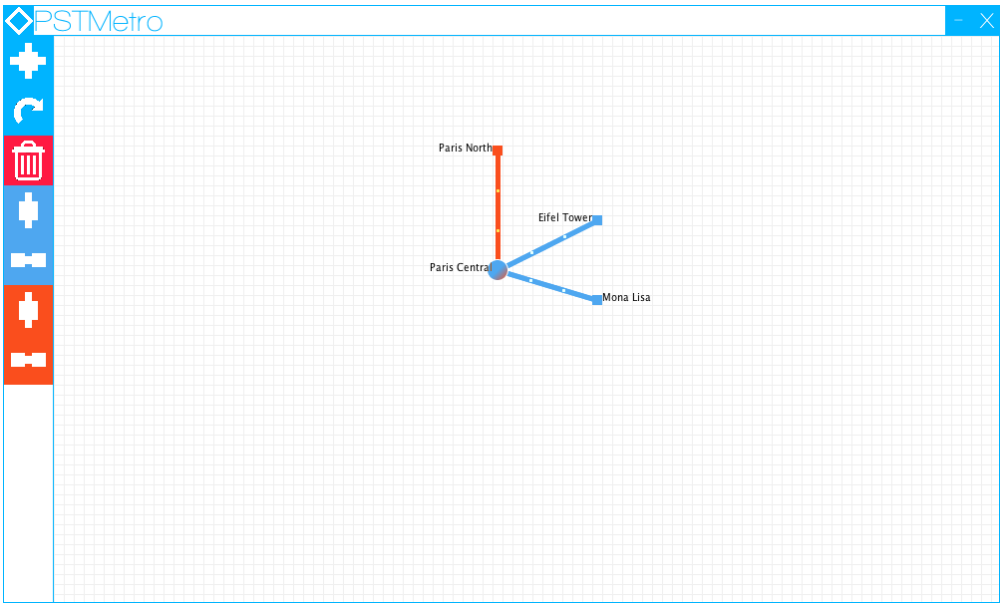
Screenshot showing the toggle rotate button.

Appendix 2.11



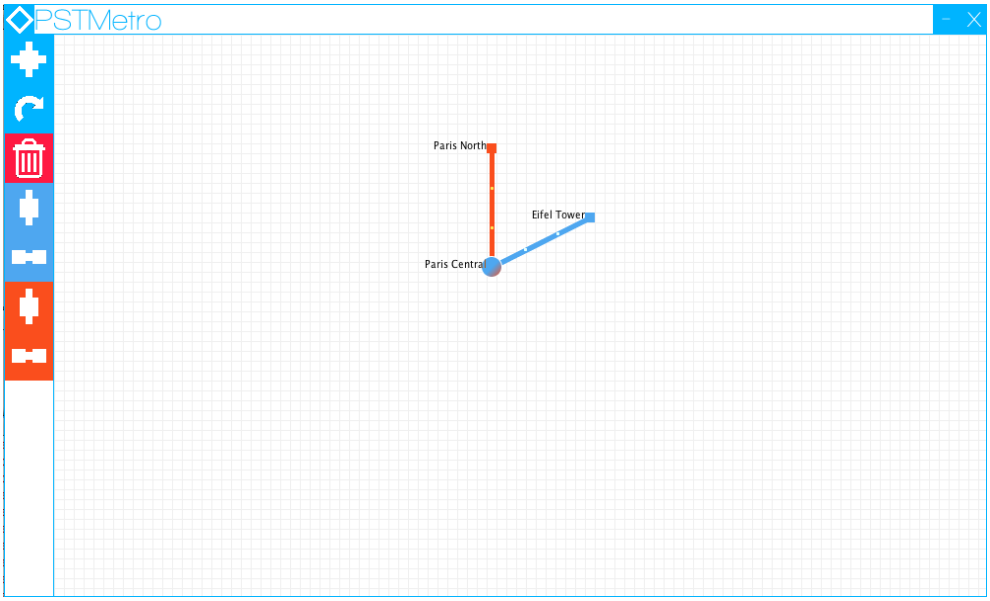
Screenshot showing the station name “Mona Lisa” after rotation.

Appendix 2.12



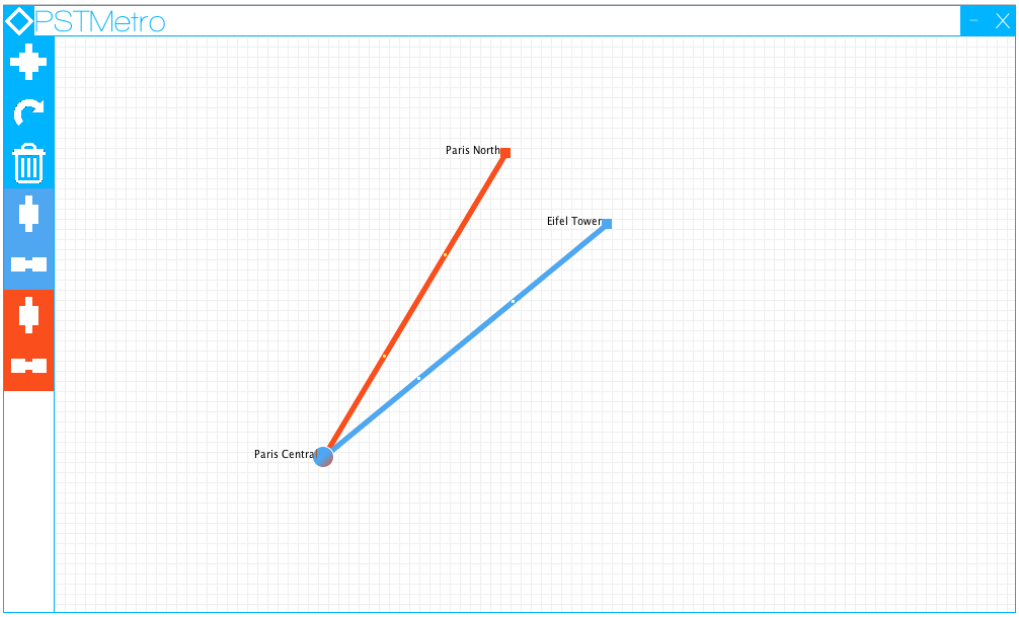
Screenshot showing the connection between “Eiffel Tower” & “Mona Lisa” has been deleted.

Appendix 2.13



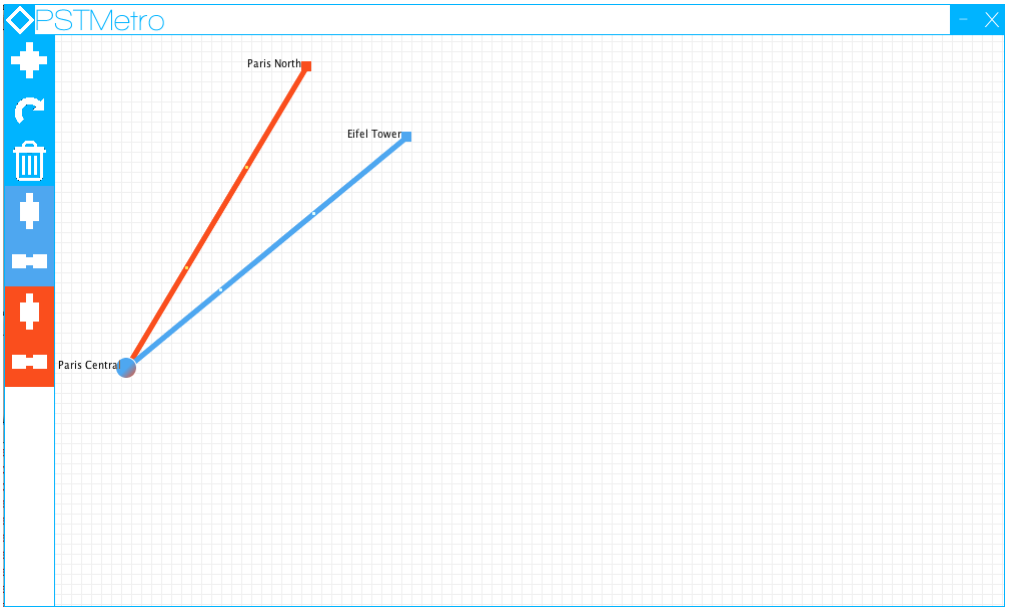
Screenshot showing the station “Mona Lisa” after deletion.

Appendix 2.14



Screenshot showing that “Paris Central’s” position has been moved on the map/grid.

Appendix 2.15

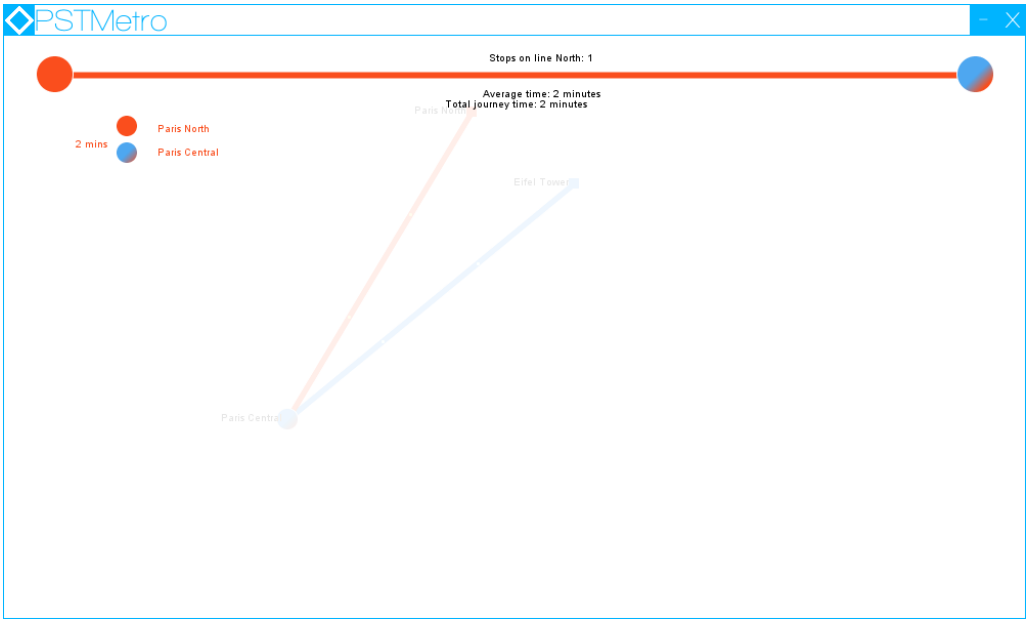


Screenshot showing the whole system being moved to the left.

Appendix 3

Screenshot showing the Journey Planner window. It also shows that the source station has been selected.

Appendix 3.1



Screenshot showing that the destination station has been selected. The journey planner window will now appear, displaying stations, changes and timings between stations.

Bibliography

Oracle 2013. The List Interface . Oracle . Available from: http://docs.oracle.com/javase/tutorial/collections/interfaces/list.html [Accessed 26/11/2013]

Oracle 2013. The Map Interface. Oracle. Available from: http://docs.oracle.com/javase/tutorial/collections/interfaces/map.html [Accessed 26/11/2013]

Oracle 2013. The Set Interface. Oracle. Available from: http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html [Accessed 26/11/2013]