# Display RST Application Review

> **Author:** Tony Soong (2012011271)
>
> **Time:** 8/28/2013

This is the document for my project for the Qt class. It is used to display Rectilinear Steiner Trees using a Qt Application.

## Usage

- **Help(Ctrl+H):** Displays help for the project.
- **File->Read(Ctrl+R):** Reads some point data from a file, and displays them.
- **File->Refresh(Ctrl+E):** Clears all the lines on the window, and resets the view point.
- **File->Clean(Ctrl+C):** Clears everything on the window.
- **Algorithm->SMST(Ctrl+S):** Displays the Minimum Spanning Tree of the points.
- **Algorithm->LMST(Ctrl+L):** Displays the RST obtained by algorithm of LMST.
- **Algorithm->ZMST(Ctrl+Z):** Displays the RST obtained by algorithm of ZMST.
- **Add Point:** Use **Add Point(Ctrl+N)** or **Right Click** on a place without points.
- **Delete Point: Right Click** on a point to delete that point.
- **Drag Point: Left Click** on a point and drag the point to replace the point.
- **Move Viewpoint:** Use **Arrow Keys** or **Right Click & Drag** to move the whole canvas.
- **Zoom In/Zoom Out:** Use **Zoom In(Ctrl+>)**, **Zoom Out(Ctrl+<)**, or the **mouse wheel**.

## Classes

- **MainWindow:** the main window for the application.
- **SketchPad:** the class for painting and stores the data of points and lines.
- **AddPoint:** (inherited from QDialog) the class used to manually add a point.

## Functions

### MainWindow

The `MainWindow` class mainly creates the overall environment of the class, such as the menu bars, tool bars, and status bars. The first two contains the actions connected to the `SketchPad` class pointer, namely `MainWindow.sketchpad`, which further operates the data and displays the points.

### SketchPad

The `SketchPad` class gets input from files or manually, deals with the algorithms(SMST, LMST, ZMST) and paints the points by using `QPainter` and `paintEvent()`. Whenever we change the points or lines in `SketchPad`, it will call `update()` and renews the picture.

> Note that the MainWindow also deals with keyborad inputs as the `MainWindow.sketchpad` cannot be selected by keyborad input, while mouse inputs are dealt in `SketchPad` class, since we need to further information on the exact positions of the points, and doing it in `MainWindow` will cause deviations(for about (0px, -40px) on my computer).

### AddPoint

`AddPoint` is a simple QDialog class which uses hard-coded layouts, and deals with input points.

## Implementations

### Event Handling

Event handling is largely used. In this project, `wheelEvent()`, `mouseMoveEvent()`, `mousePressEvent()`, `mouseReleaseEvent()`, `keyPressEvent()` and `paintEvent()` are used. They mainly serve to change the data points and repaint the picture. Private variables are used for them to temporarily store the data generated by previous events, so that posterior events can use them for calculation, such as:

```
void SketchPad::mousePressEvent(QMouseEvent *event) {
    if (event->button() == Qt::RightButton) {
        rightStartPos = event->pos();
        rightDragged = false;
        rightSelected = -1;
        //...
    }
    //...
}
```

### Painting

Painting is very essential for correct event handling. This implementation uses three variables to calculate the display position of points, namely (`xView`, `yView`), (`xFocus`, `yFocus`), and `scale`. `xView` and `yView` adjusts the view point of the canvas and is largely used in transition; `scale` adjusts the size of the canvas which is used in zooming; `xFocus` and `yFocus` adjusts the central point for zooming. The function to derive the painting point from the original point(and vice versa) are as follows:

```
Point SketchPad::calculatePaintPoint(int x, int y) {
    int xx = (x - xFocus) * scale + xFocus + xView;
    int yy = (y - yFocus) * scale + yFocus + yView;
    return Point(xx, yy);
}
```

```
Point SketchPad::calculateOriginalPoint(int x, int y) {
    int xx = (x - xFocus - xView) / scale + xFocus;
    int yy = (y - yFocus - yView) / scale + yFocus;
    return Point(xx, yy);
}
```

## Slots and Signals

These are very simple actions for us to connect with Qt predefined classes, and we don't need them between our classes.

```
readAction = new QAction(tr("&Read"), this);
readAction -> setShortcut(tr("Ctrl+R"));
connect(readAction, SIGNAL(triggered()), this, SLOT(read()));
```