Synthetic Geometry Toolbox | H02

CHENG, Tien-chun
20537176, tchengac@connect.ust.hk

PACHECO RODRIGUEZ, Wemp Santiago
20543280, wspr@connect.ust.hk

PATUPAT, Albert John Lalim
20544416, ajlpatupat@connect.ust.hk

December 16, 2019

Project Objectives

The main purpose of this project was to create an environment for synthetic geometry, more widely known as straightedge and compass constructions. More specifically, this project aimed to design and implement a computer program with the following capabilities:

1. Imitates a straightedge and a compass, and automatically constructs more advanced objects based on these classical tools, such as midpoints, parallel lines, perpendicular lines, and circles passing through three selected points.
2. Supports addition, mutation, and deletion of points and certain constructions, which leads to the adjustment of preexisting geometric objects, such as lines, circles, and triangles.
3. Visualizes the graphs of user-defined geometric constructions.
4. Supports construction of important triangle centers of selected triangles.

Features Incorporated

After a significant amount of time planning, programming, and debugging, we managed to successfully implement the following features in our final program:

1. Automatic creation of advanced constructions given certain parameters:
   (a) Creation of *Points* of the following types:
      - Independent point, given Cartesian coordinates (x, y).
      - Point on line/circle, given Cartesian coordinates (x, y) and *Line*/*Circle*.
      - Midpoint, given two *Points*.
      - Intersection, given any pair of *Lines* and/or *Circles*.

   (b) Creation of *Lines* of the following types:
      - Line passing through points, given two *Points*.
      - Parallel line through point, given *Line* and *Point*.
      - Perpendicular bisector, given two *Points*.
      - Tangents through point, given *Circle* and *Point*.

   (c) Creation of *Circles* of the following types:
      - Passing through points, given three *Points*.
      - Centered on point and passing through point, given two *Points*.
      - Centered on point with line segment for radius, given three *Points*.
2. Mutation of *Points* and deletion of *Constructions*, which triggers adjustments of all figures derived from the edited/removed object. In particular, *Points* with at least one degree of freedom can be moved through drag-and-drop.
3. Visualization of graphs of all user-defined *Constructions* and information display of clicked and selected object.
4. Identification of *Triangles* and immediate construction of important triangle centers for any given *Triangle*. Supported triangle centers are centroid, incenter, circumcenter, orthocenter, nine-point center, and Lemoine point.

Figure 1: Illustrating feature 1. Adding a new point.

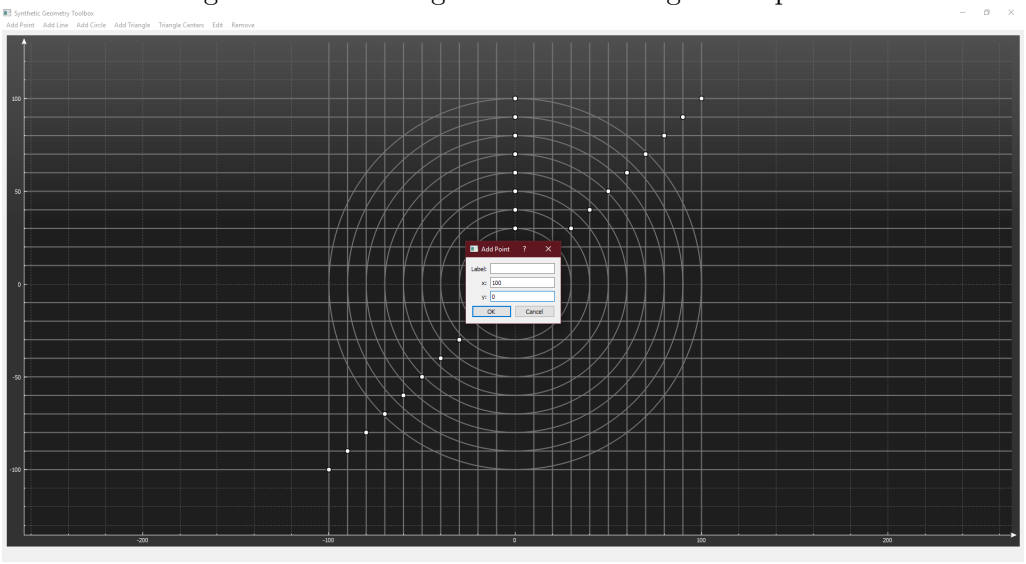


Figure 2: Illustrating feature 1/3. Point has been added and the graph is visualized.

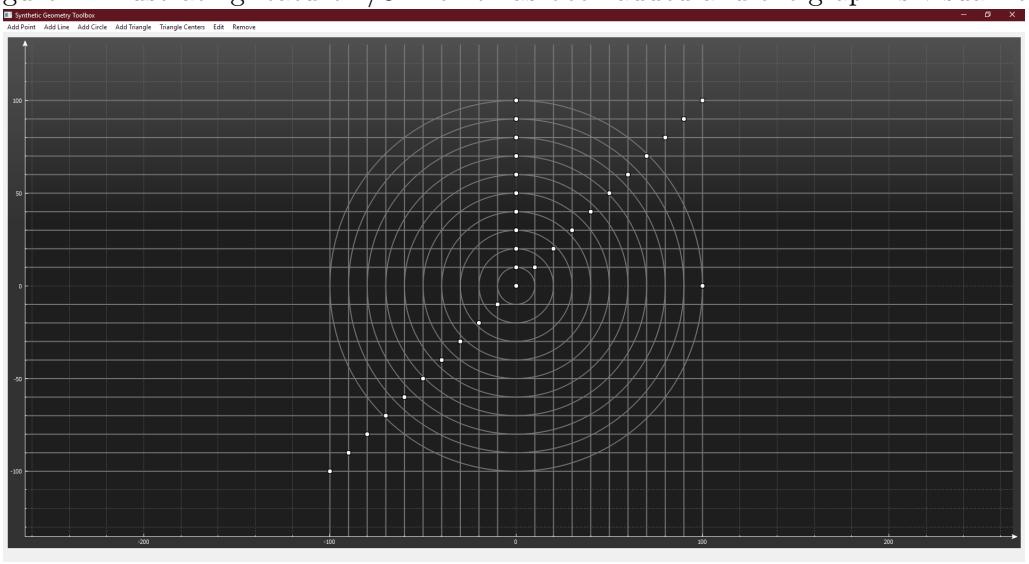


Figure 3: Illustrating feature 2. Dragging the center of a series of concentric circles.

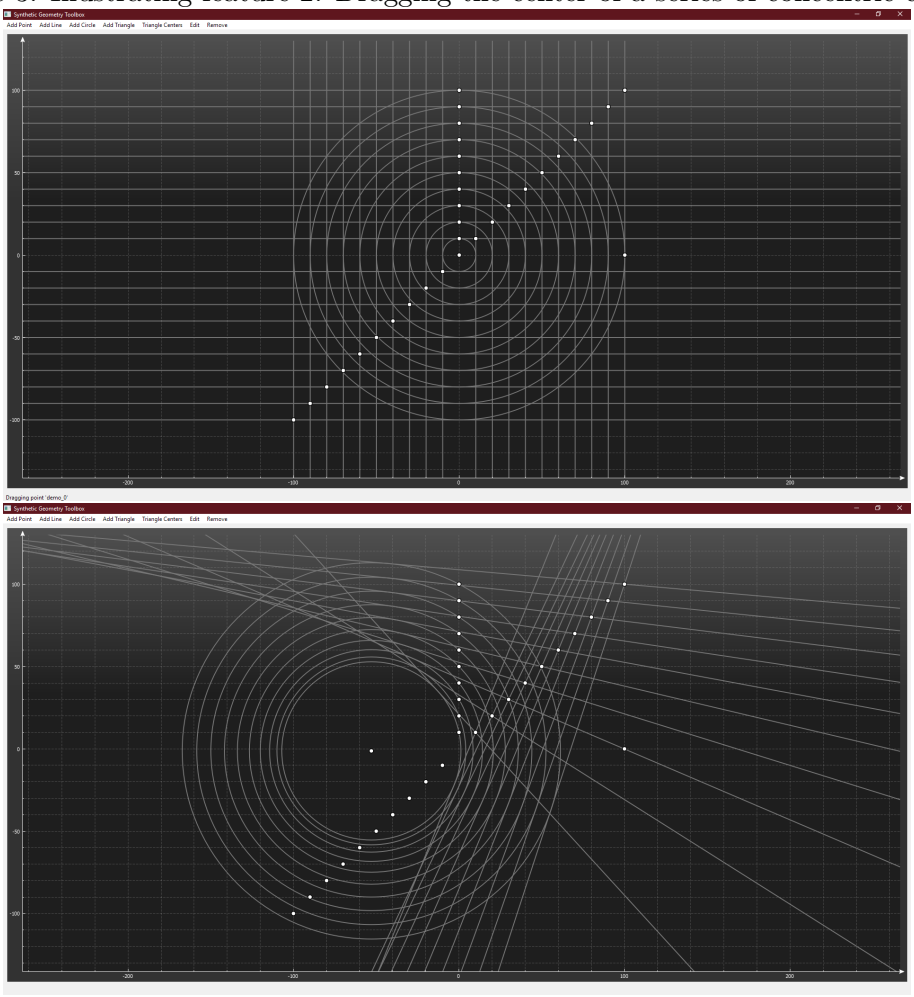
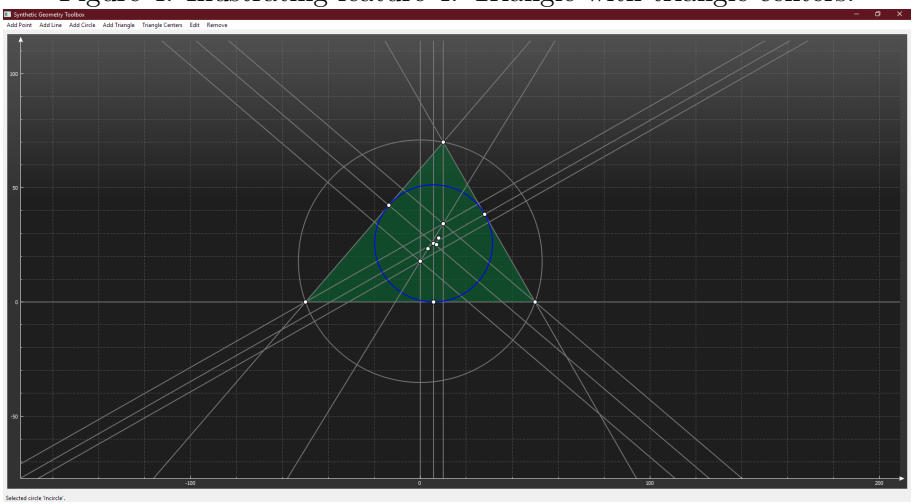


Figure 4: Illustrating feature 4. Triangle with triangle centers.



OOP Design

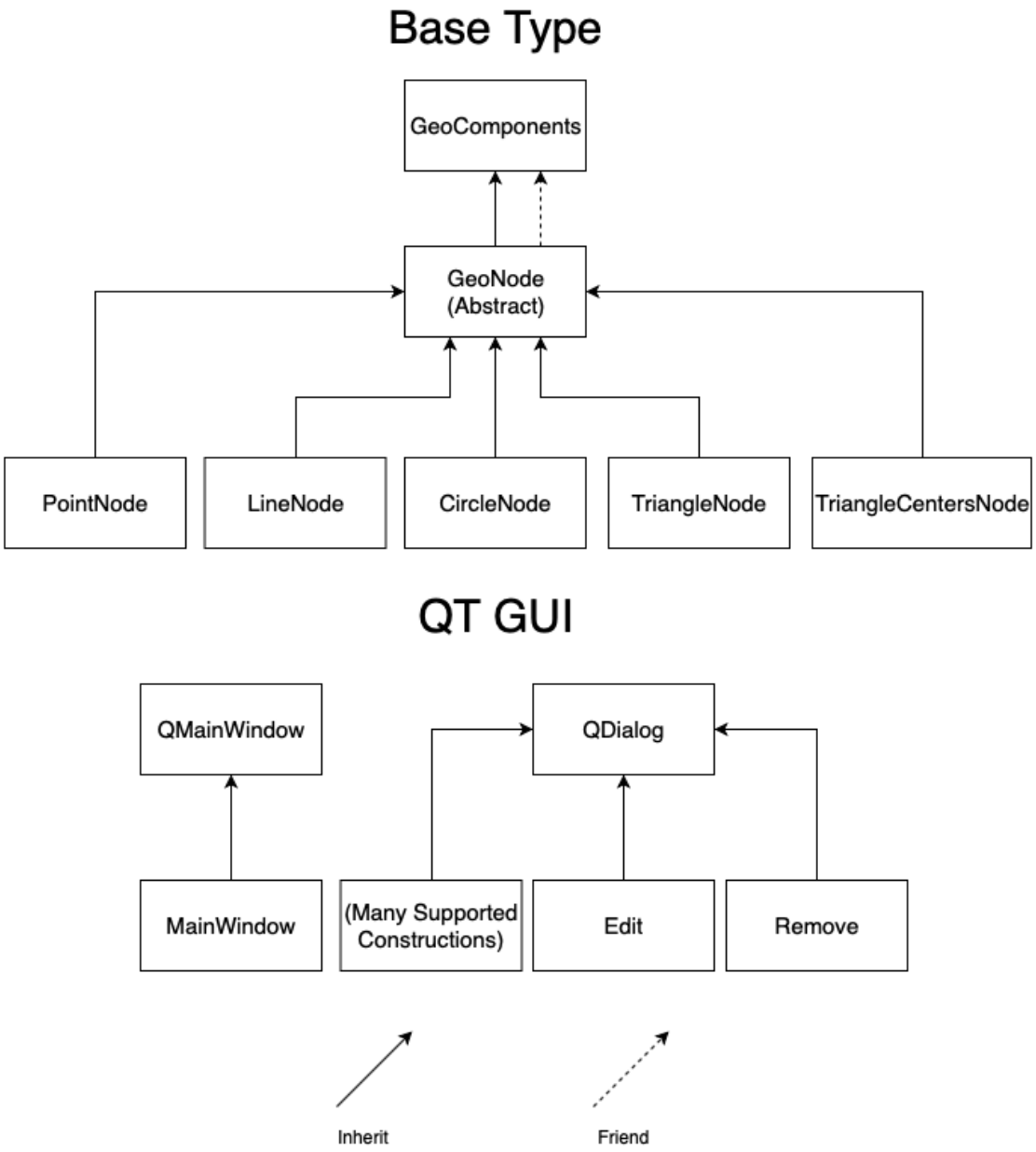
The main challenge for our OOP Design was to be able to contain and manipulate drastically different types of geometric constructions. In order to tackle this problem, two levels of exhaustive classification were used.

For the lower level, we noticed that all points constructed in different ways interact with other geometric constructions primarily through their Cartesian coordinates. Hence, it was reasonable to classify these different points into a single *PointNode* class. To differentiate various types of points, a *PointType* enum type and function overloading were used to specify the appropriate constructor, mathematical definition and calculations. For other geometric constructions, a similar treatment was applied, thus creating *LineNode*, *CircleNode*, *TriangleNode*, and *TriangleCentersNode* classes.

For the higher level, we observed that there were several combinations of the *Node* types of the *parents*, which are preexisting geometric constructions an object is dependent on. For example, a *PointNode* may have a single *CircleNode* parent or two *LineNode* parents. Hence, to approach this problem, all *Node* classes were to inherit from an abstract *GeoNode* class. Applying Liskov's substitution principle, *GeoNode* pointers can reference to different *Node* objects, which allows storage of different constructions in a single container. Furthermore, virtual functions were used to simplify access, update, and display of the different *Node* objects, which proved to be useful for the numerous constructors. In order to store and manipulate the *GeoNode* objects as a whole, a friend class *GeoComponents* was defined to serve as the base of our project.

From this back-end implementation of the geometric constructions, a simple UI component was designed using Qt GUI Programming. The OOP Design of this component has three main parts: base, input, and output. The base is a *MainWindow* class which inherits from the *QMainWindow* class, while the input element is composed of multiple classes inherited from the *QDialog* class. For ease, the forms of both of these classes were manually edited using Qt Creator's Design Mode. Lastly, the output element makes use of the external library *QCustomPlot*. It consists of an object of class *QCustomPlot* where the graphs of the different geometric constructions are represented by different derived classes from *QAbstractItem* and *QAbstractPlottable* according to the type of the geometric construction. Identification of a figure in the *QCustomPlot* to its specific construction in *GeoComponents* proceeds from shared labels whereas updating of the *QCustomPlot* makes use of pointers to the respective figure in each *Node*.

Figure 5: Design of classes and inheritance.



Data Structures Used

1. Directed Acyclic Graph (DAG): Geometric components are generally constructed from preexisting ones. Thus, in order to keep track of the dependencies, each child *GeoNode* has a dynamic array of pointers to its parent *GeoNodes*. Note that the order of construction is a topological order in the dependency graph, which implies that the graph is acyclic.

2. STL Vector (Dynamic Resizable Array): Storing *GeoNodes*. O(1) access is preferable for tracking ancestry in the dependency graph.

Considered Alternative Solutions:

1. Use STL map to maintain mapping between user-defined labels to *GeoNodes*: There will be too much copies in operations; the memory required of an unnecessary data structure is larger than array; there is an extra log in worst case time complexity.

2. Use STL set for faster single node insertion and deletion: Due to the dependencies in the graph, the worst case complexity is also a log worse than vector.

3. Use STL list: The operations will be similar to the vector, but the code is cleaner with vector.

4. Store bidirectional edge, keep track of child *GeoNodes*: No worst case time saved. Storing dependencies is a requirement of updating, but we can omit the reverse edge to save the memory. (Note: The condition that this approach saves time is if the graph is a tree, and thus, apply tree decomposition. Unfortunately, it's not.)

5. Adjacency matrix to store the graph: It is likely that the graph is a sparse graph, so this approach wastes memory.

External Libraries Used

- **Standard C++ Library:** iostream, cmath, vector.
- **QT:** QMainWindow, QApplication, QDialog.
- **QCustomPlot.**

Conclusion

The use of a vector to store our geometric constructions, in combination with our Directed Acyclic Graph of dependencies, allows us to edit and remove constructions from the vector while updating all their dependants effectively by making use of our abstract data class. This synergy between all the components allows us to do real-time updating of the components under edition, and in particular, allows for a very smooth display of the constructions when under constant progressive mutation of a point. In addition to all this, the code structure has a very high maintainability and re-usability due to the strict organization. In particular, the GUI is being kept as a separate component from the main data structure to exemplify how easy it is for our classes to adapt and be used by another component.