MiniBoard Technical Report

YUXINAG LIU(151180087)
Nanjing University
School of Electronic Science and Engineering
Department of Electronic Engineering
lyx970124@gmail.com

Abstract

This is the technical report for my own graphics system, MiniBoard. All copy rights goes to Yuxiang LIU. For those who made contributions to help me finish this project, I present a big thank you to all of you, and your websites, blogs, books or other things will be listed in reference. MiniBoard is a simple graphics system for people to draw something they like. Almost all graphics algorithms in MiniBoard is implemented by myself, with a little help of some high level libraries. This documentation is about all graphics algorithms, process of design, results and experiments of this project. For more details about how to use MiniBoard, please refer to MiniBoard Manual.

I. Introduction

The assignment of computer graphics course is to build a small graphics system based on existing graphics libraries like OpenGL or Qt, etc.I chose Qt to complete the assignment and named my system 'MiniBoard'. Technically, MiniBoard is a painting system that allows you to paint basic shapes and edit them. Detailed description of MiniBoard's functionality and usage is available in the system manual. This documentation, instead, is mainly about the design, construction and experiments of the system. The sections that will be described in detail later are:

- Graphics Algorithms
- Software Structure
- Encountered Problems
- Build Environment
- Summary

Although there are many bugs and deficiencies, I consider my job well done in consideration of my lack of programming background. Future improvements upgrades will be made and hopefully a more convenient and robust version of 'MiniBoard' will appear.

II. Graphics Algorithms

There are a number of high level libraries for us to draw geometry primitives and do something very cool. But the goal of our CG course is to understand and implement them by ourselves. So here are some most crucial ones implemented in MiniBoard.

i. Geometry Primitives Generation

i.1 Line

For line generation, Miniboard adopts Bresenham algorithm. It is more precise than DDA and more capable of interpreting lines in exrtreme circumstances. The steps of Bresnham line generation algorithm are as follows:

- Take $|k|<1$ as an example.
- Input start and end position of the line.
- Load start point into frame buffer and paint it to screen.
- Compute constants like dx and dy, also the first perceptron, i.e. k=0, and it's recursion formula.

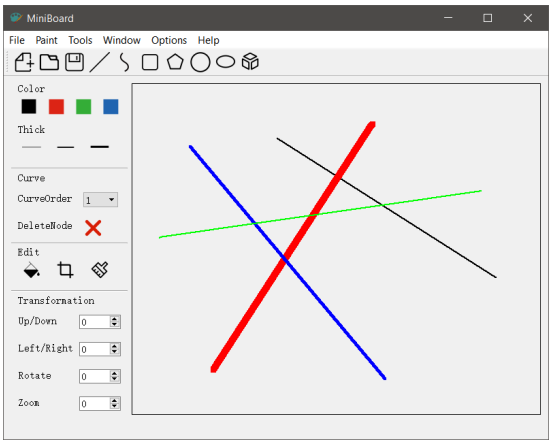


Figure 1: *Line.*

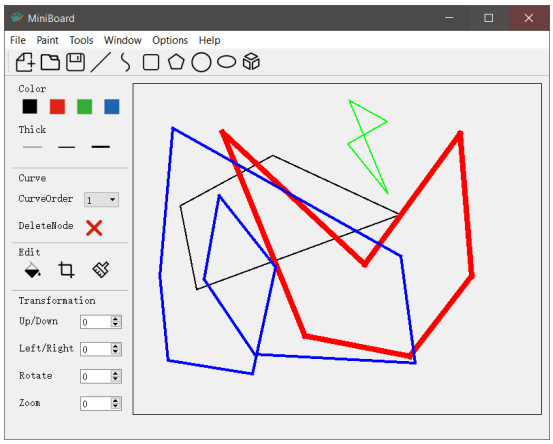


Figure 2: *Polygon.*

- Start painting points from k=0, and determine the next point based on perceptron's value. More specifically, we determine the next y based on x's increment.
- Until we finish painting the end.
- If we want to draw lines of more slopes, let's say $|k|>1$, we only need to determine next x based on y's increment.

The experiment result is demonstrated in Figure 1. As the figure shown, in general, Bresenham algorithm does a pretty good job finishing the task. More details about how to draw lines in different colors and thicknesses will be discussed in Software Structure and how to interactively draw them in MiniBoard Manual.

i.2 Rectangle

This is the easiest shape we can imagine. We only have to input the diagonal points and two pairs of parallel lines to form a rectangle.

i.3 Polygon

Although polygon should be as easy as rectangle, in that we only have to connect all vertices, in practice however, how to draw them interactively proves to be a troublesome problem. This part will be discussed thoroughly in Software Structure and MiniBoard Manual.

i.4 Circle

Miniboard adopts middle-point circle generation algorithm to draw circles. The steps of this algorithm are as follows:

- Input two endpoints of circle's diameter and determine the center position and radius of the circle.
- Compute the first perceptron.
- Translate the circle to the origin and we only need to draw half of the part in the first quadrant because we can use circle's symmetry to derive other parts.
- Start painting from (0, r), and determine the next point based on perceptron's value.
- Until we hit x=y.

i.5 Ellipse

As a matter of fact, ellipse is very similar to circle, only that it's a more generic form. The only difference between them is that we have to draw the whole part of first quadrant and when doing so, we must divide this part into to subparts. The detail is shown below:

- Before we hit x=y, we determine the next y based on x's increment.
- When it hits x=y, we use the last point painted as the first point in the second region and do it the opposite way, i.e. determine the next x based on y's increment, exactly the same as line generation.

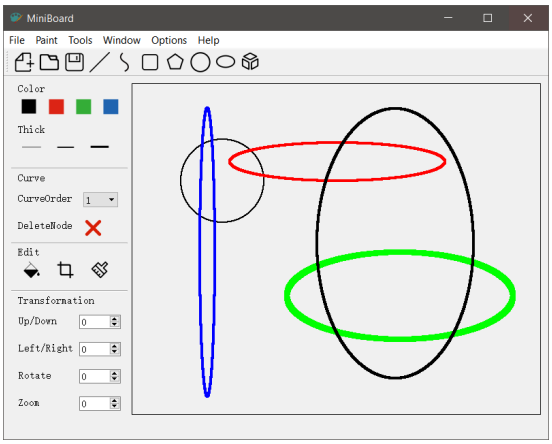


Figure 3: *Circle and Ellipse.*

The results of circle and ellipse generation is demonstrated in Figure 2.

i.6 Curve

In my opinion, besides filling a polygon and displaying 3D objects, curve generation is one of the most difficult algorithms in our course. Here I use B-spline in MiniBoard. In the mathematical subfield of numerical analysis, a B-spline, or basis spline, is a spline function that has minimal support with respect to a given degree, smoothness, and domain partition. It's mathematical representation is:

$$N_i^n(x) = \frac{x - u_i}{u_{i+n} - u_i} N_i^{n-1}(x) + \frac{u_{i+n+1} - x}{u_{i+n+1} - u_{i+1}} N_{i+1}^{n-1}(x)$$
$$N_i^0(x) = \begin{cases} 1 & ,x \in [u_i, u_{i+1}) \\ 0 & ,otherwise \end{cases} \tag{1}$$

Actually this formula is not the big deal. The most tricky thing is how to draw and edit curves interactively. Users could use MiniBoard to easily create and edit curves via mouses, just as we do in most common graphics softwares like Maya or 3dsMax. This part will be discussed in Software Structure and MiniBoard Manual. The test results, including first, second and third order curves, are shown in Figure 4.

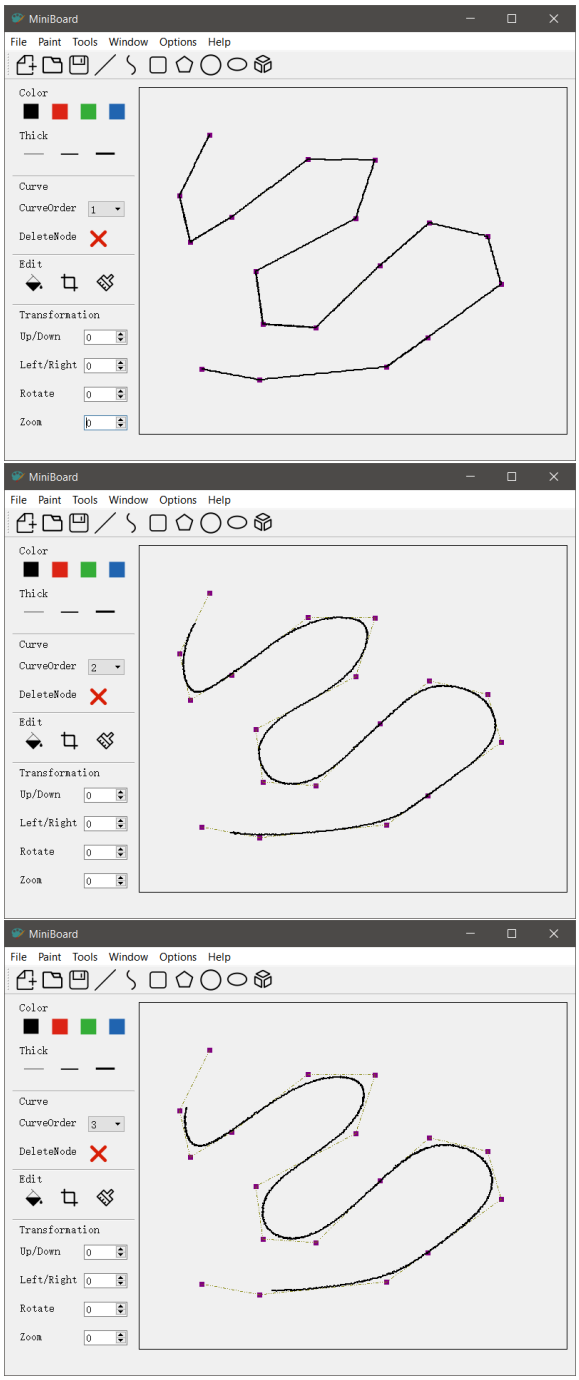


Figure 4: *Curve.*

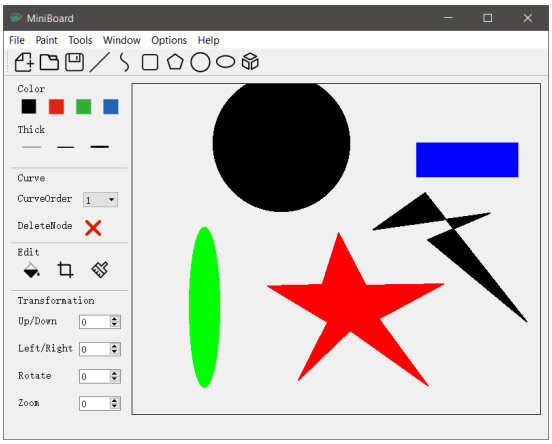


Figure 5: *Fill.*

ii. Geometry Primitives Edit

ii.1 Area Filling

When it comes to fill, we often think of scan line and neighborhood fill. For MinBoard I choose scan line algorithm to fill polygon. But I use a little trick to fill rectangle, circle and ellipse. The methods are listed below:

- Rectangle: Just do it the brutal way. I use two nested loops to fill a rectangle.
- Circle: When drawing circles, I use Bresenham algorithm. Here we just do it again, but in a different way. Every time we do not paint a point, instead we paint a line. That means we generate a circle that has the same skeleton as the original one. But this circle is solid. So it gives us an illusion that we have filled it. I do not want to use neighborhood method because it's too slow, despite it's simplicity.
- Ellipse: Same as circle.
- Polygon: Scan line method. It is a little bit difficult to explain this algorithm here, so I just skip this.

The test results are in Figure 5.

ii.2 Shape Clipping

Filling and clipping in MiniBoard are all based on shape objects. This means we can only edit the fresh drawn shapes. We cannot edit those

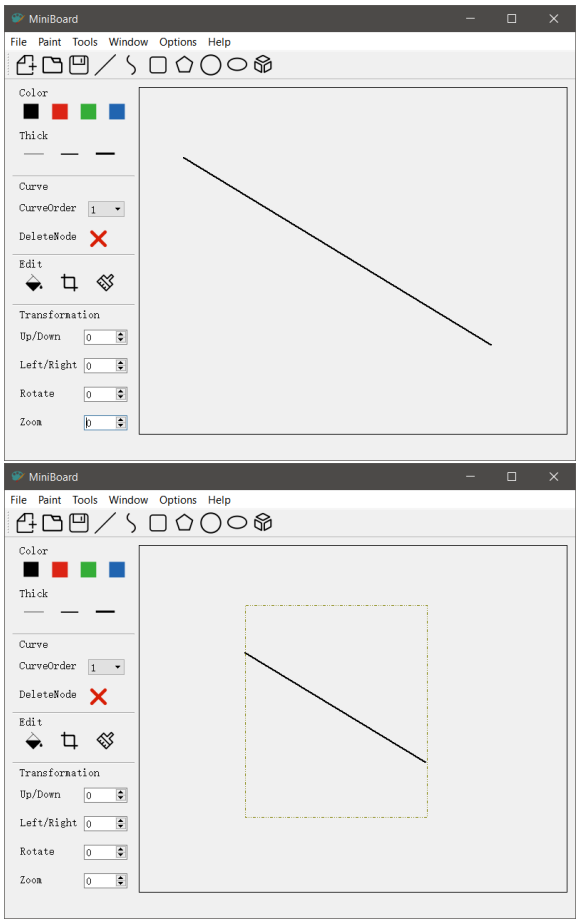


Figure 6: *Clip.*

shapes that have been drawn before. Because clipping for polygons is pretty hard to implement in my graphics system so I only finish clipping for lines. I adopt Cohen-Sutherland algorithm to complete the task. The core of this algorithm is to encode area into binary code to make it easier to compute intersection point. Test results are shown in Figure 6.

ii.3 Transformation

Theoretically it should be pretty easy to do transformation to geometry primitives. Transformation is just applying some matrices. All linear transformations, no matter how many of them, can be represented by one matrix.

However, in practice, all the geometry primitives are generated by algorithms, not by some

high-level APIs. This makes it a lot more difficult to apply transformation matrix to these primitives. What I do here is to write my own function to apply matrix to a specific point. Also, it is inefficient to apply transformation to all points of a shape. So we store different information of different shapes. And we transform these critical features of specific shapes and then repaint the screen. Another problem about rotation and zoom is that we have to specify a center to perform these operations. So I decide to first move primitives to the origin, then apply the transformation matrix, and finally move them back to where they belong. Thus it gives us the illusion that we are actually doing transformation with respect to their centers.

The details of different types of transformation of different shapes are listed below:

- Line: Store two endpoints. When perform transformation, we perform it on these two points. We do not need rotation or zoom for lines because we have a more effective way to achieve that, which will be discussed in MiniBoard Manual.
- Rectangle: Store four vertices. Perform transformation on them.
- Polygon: Store all vertices. Perform transformation on them. For rotation and zoom we have to specify a center. Here I use xc=(xmin+xmax)/2, yc=(ymin+ymax)/2, where xmin is the minimum x value of all vertices.
- Circle: Store two endpoints of the diameter. Perform transformation on them. It is a bit similar to rectangle because we can determine a circle by these two points. Also we actually do not need to rotate a circle because it is meaningless.
- Ellipse: Here we have to do it the brutal way. I store all points and apply matrix to them. The reason is that ellipse is generated based on the fact that its axis is parallel with coordinate axis. So when repainting, we have to repaint all points, otherwise we cannot perform rotation properly.

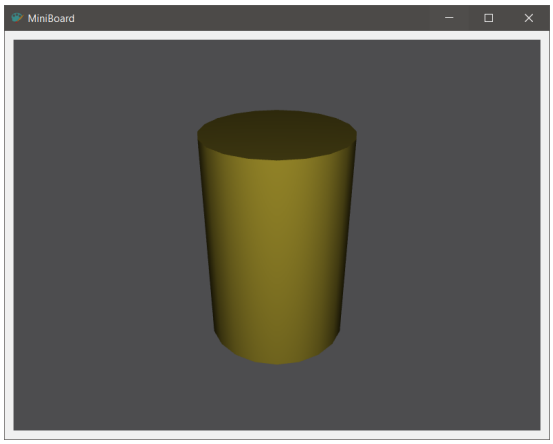


Figure 7: *Three-D Display.*

iii. 3D Display

Realistic computer graphics is about displaying 3D objects as they are in the real world. One of the most important trick to achieve this is to determine the spacial order of these objects and hide those in the back. Here z-buffer, also known as depth buffer, seems to be an ideal option. When an object is rendered, the depth of a generated pixel (z coordinate) is stored in a buffer (the z-buffer or depth buffer). This buffer is usually arranged as a two-dimensional array (x-y) with one element for each screen pixel. If another object of the scene must be rendered in the same pixel, the method compares the two depths and overrides the current pixel if the object is closer to the observer. The chosen depth is then saved to the z-buffer, replacing the old one. In the end, the z-buffer will allow the method to correctly reproduce the usual depth perception: a close object hides a farther one. In this way, we can display 3D objects rather realistically.

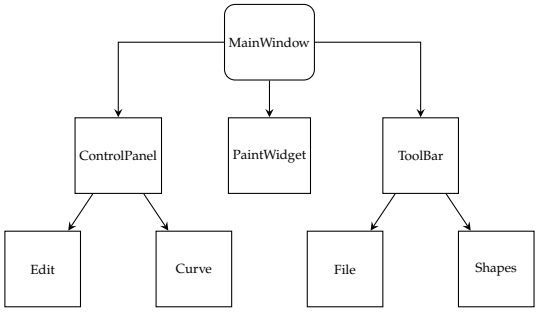
The three-D is demonstrated in Figure 7.

III. Software Structure

i. General Structure

The essential idea is to design a software with certain level of scalability, i.e. we are programming the APIs, not specific implementations.

Although I am not quite proficient with programming, which means this goal is not entirely achieved, and with all due respect I think this task is very hard for undergraduates to finish it alone, I managed to figure out a compromise. I designed a abstract class for all shapes, providing the APIs. And in paintwidget and mainwindow files, I don't have to use specific shape class, instead, I just leave the task to compiler let it dynamically bind a specific shape. This means that if we want to add a new shape some day, we don't need to change paintwidget and mainwindow files. More adjustments have to be made in the future to polish MiniBoard, but for now, it is good enough for me(although not good enough for others). The general structure of MiniBoard's software is demonstrated in the flow chart. Details will be discussed below.



ii. Abstract Class: Shape

This is the base class of all shapes. Here we provide all APIs that are needed in MiniBoard. There is one little problem that bothers me. Polygon and curve are very different. So I have to design many APIs specificly for polygon and curve, which makes shape class a little bit prolix. Anyway, the job is done. Also, making shapes into objects has a big advantage, which is we can assign different attributes to different shapes. So we could draw colorful shapes, which is fun.

iii. Main Window

Main Window handles signal transmission between controlpanel, paintwidget and mainwindow itself. I use Qt creator to design the general appearance of MiniBoard. Then use codes to complete all functionalities.

iv. Paint Widget

Paint widget is the main area for drawing. I add this widget to limit our drawing actions to a specific area because we absolutely don't want to draw something onto the toolbar or the control panel. And of course, the most crucial functions are all here, e.g. paintEvent for all painting events, mouseEvent for mouse interaction with users.

v. Control Panel

This is another widget that I added. Control panel is for setting drawing options, editing primitives and transforming them. All the buttons are designed in Qt creator UI. And the functionality is implemented by codes.

IV. Encountered Problems

i. Rotation Interpolation

When rotating shapes, images or anything digital, we must do a interpolation due to the discreteness of raster display. Let's say we rotate an image by 60 degrees, in most circumstances we use nearest neighbor interpolation or bilinear interpolation to fill in the blanks left by rotation.

However in MiniBoard, I did not find a proper way to do interpolation to geometry primitives. Also, because we use raster display, there are a lot of approxiamtions when doing rotation. As a result, the outcome of rotation of primitives generated by algorithms discussed before is somehow a little unsatisfying. When we rotate these shapes, they will deform slightly, especially ellipse, which is shown in Figure 8.

But after all the functionality is implemented anyway. We can do rotation, it's just it does not have a perfect outcome. So perhaps in

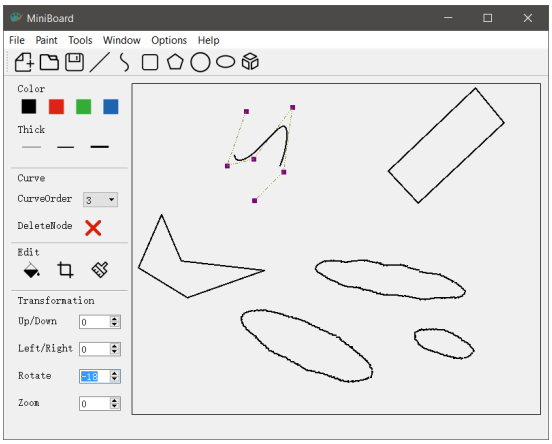


Figure 8: *Interpolation problem.*

the future I will look into this and solve the problem.

ii. Clip Window

MiniBoard, to some extent, already provide users with a friendly interface. But there is a huge problem about clipping. When dragging to form a clip window, theoretically speaking, we should be able to do it freely. But here we have to drag from the top left to the bottom right, even if I already take this into consideration. For now it remains this way and I haven't find a solution to make it more friendly to clip. Also, clipping has some bugs sometimes.

iii. Data Structure

I myself is not a CS student so I am not quite familiar with programming and I lack some basic CS knowledge, such as data structure. I have never learned data structure before. As a result, the two specific tasks in our assignment, i.e. filling polygons and 3D display, are rather difficult for me. So I did not finish filling polygons and displaying 3D objects all by myself. But maybe after next semester's Data Structure course, I will come back and reimplement these algorithms.

iv. Note

There are countless other problems which are not listed here. Apparently those are already solved, consuming me a huge amount of time. These listed ones remains untackled. I have not come up with a proper solution so they are for future study.

V. Build Environment

- System: Windows 10 Education
- IDE: Qt Creator 4.5.0
- Compiler: MinGW 5.3.0 32-bit
- Debugger: GNU GDB 7.10.1 for MinGW
- Qt Version: Qt 5.10.0

VI. Summary

This assignment of our CG course is very challenging, even for a CS student. And as a EE student I am more than confused at the beginning. Because I am not quite familiar with coding, I completely lost my head for some periods. But gradually I got the hang of how to design a graphics system. There are countless setbacks and unsolved bugs here, and there are countless nights and days poured into completing this task. No matter what the outcome is, I am pleased by what I have accomplished. MiniBoard certainly has many bugs and deficiencies, which may be tackled in the future, but generally I consider myself successful in completing the task. A big thank you here for Prof. Yan ZHANG, Prof. Zhengxing SUN and all T.A.s!

Citations: [1] [2] [3] [4] [5] [6] [7] [8]

References

[1] Z. SUN, *Computer Graphics*. China Machine Press, 2009.

[2] Q. Company. https://doc.qt.io/.

[3] Q. Company. https://en.wikipedia.org/.

[4] FinderCheng. http://blog.51cto.com/devbean/243546, Dec 2009.

[5] RyuZhihao123. http://blog.csdn.net/mahabharata_/article/details/71856907, May 2017.

[6] xiaoyaoyouzou. http://blog.csdn.net/u014004602/article/details/49619377, Nov 2015.

[7] brightming. http://blog.csdn.net/brightming/article/details/53953926, Dec 2016.

[8] tzb592825420. http://blog.csdn.net/u013044116/article/details/49737585, Nov 2015.