UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

NUMERICAL
CONTROL PROGRAM

Gyarmathy Csaba

Technical University of Cluj-Napoca

2022-2023

Contents

Introduction

Project specification

The purpose of this project is to design, implement and simulate a program which controls a CNC (**C**omputer **N**umerical **C**ontrol) machine.

The program should read the instructions from a file which contains a path (sequence of segments and arcs) and generate the commands corresponding to the code to move a cutting head in two directions (X and Y axis) across the working surface. The traced path should be displayed on the screen.

What is a CNC?

Numerical control, often known as CNC or computer numerical control, is the automated computer-based control of machining equipment, such as drills, lathes, mills, grinders, routers, and 3D printers. Without a manual operator directly overseeing the machining operation, a CNC machine follows coded programming instructions to process a piece of material (metal, plastic, wood, ceramic, or composite) to match the given specifications.

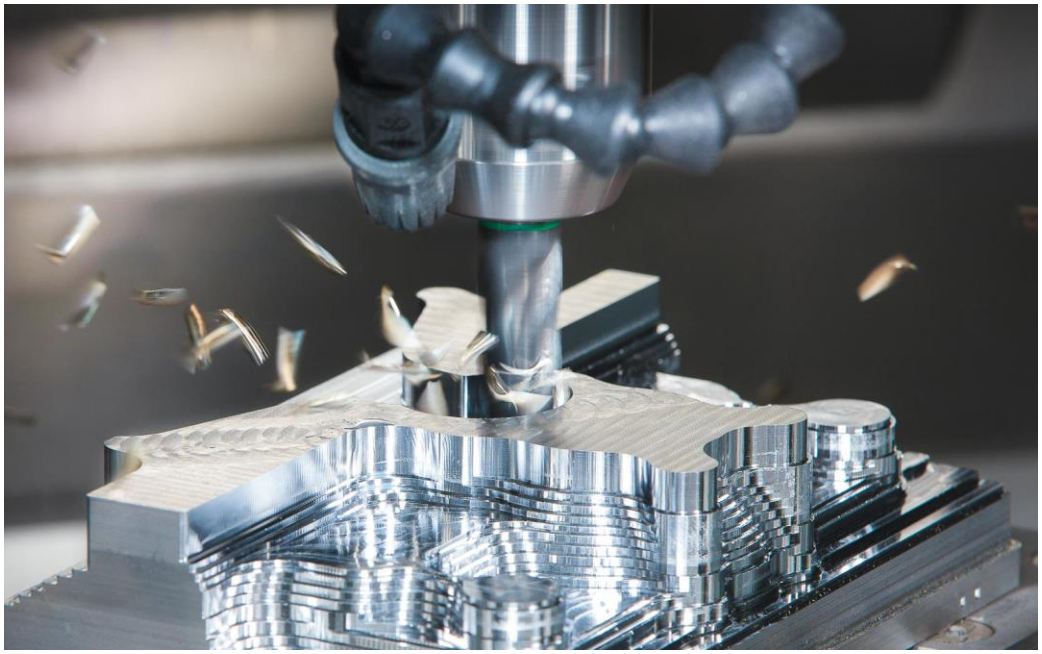

Figure 1.1. A CNC mill in action

A CNC machine is a motorized controllable tool or occasionally, a motorized manipulable platform, that is operated by a computer in accordance with precise input instructions. A CNC machine receives instructions in the form of a sequential program of machine control instructions like G-code (see Fig. 1.2) and M-code, which are then carried out. The program can be written manually by a human or, much more frequently, by CAD (Computer Aided Design) or CAM (Computer Aided Manufacturing) software that generates graphical designs.

G-code interpretation

Our program will read G-code from a file and interpret it. The following is an example of a simplified code to help understand how it works.

```
G00 X0 Y0;              //moves cutting head to initial position
G01 Y160 F100;          //linear interpolation, 100mm/min feed rate
G02 X20 Y180 I20 J0;    //clockwise radial move
                        //I and J are coordinates of the center
                        //current position of the head is the reference
G01 X160;
G02 X180 Y160 I0 J-20;
G01 Y40;
G02 X140 Y0 I-40 J0;
G01 X0;
```

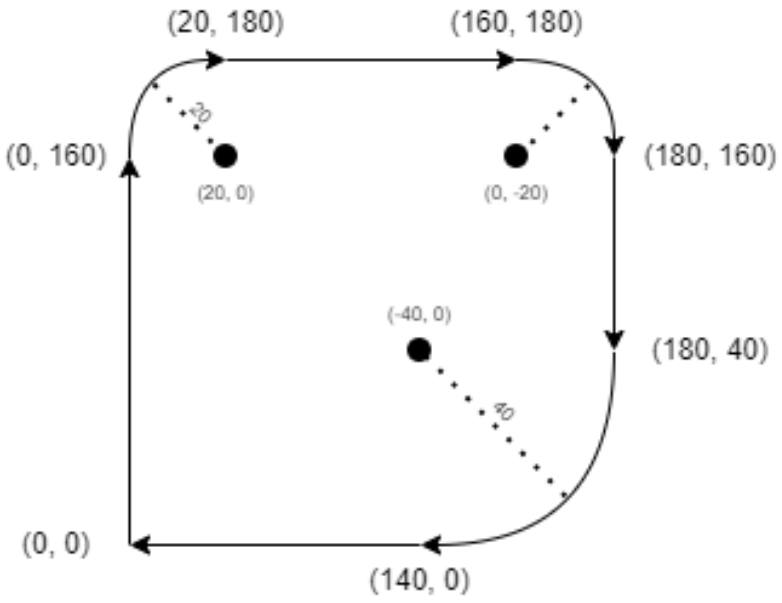

Figure 1.2. Traced example of G-code

Objectives

The program should be able to open and read information from a chosen file. This means that it must interpret what each command line means and be able to trace a line between the given coordinates or, in the case of a circle, an arc. The path the cutting head takes should be displayed on the user's screen.

Bibliographic study

CNC classification

The machine tool is called "mother machine" in the sense that it is a machine that makes machines. As machine tools have advanced from manual machine tools to NC machines, these have become perfect in the role of mother machines with the improvement of accuracy and machining speed.

NC machine tools can be classified as **cutting machines** and **non-cutting machines**. A cutting machine means a machine that performs a removal process to make a finished part, milling machines, turning machines and EDM machines being good examples. In our case, which is a plasma cutter, we are talking about a cutting machine, quite literally, as it cuts through sheet metal to create the finished product. Non-cutting machine tools change the shape of the blank material by applying force. Press machines are good examples of this. In addition, robot systems for welding, cutting, and painting can be included in a broad sense.

About the code interpreter

For this project, I will design and implement a code interpreter. The code interpreter is a software module, which translates the part program into internal commands for moving tools and executing auxiliary functions in a CNC system. And to do so, we need to understand the basics of the G-code the program will interpret. Some G-code commands are the following:

```
[G00 Rapid Motion Positioning]
[G01 Linear Interpolation Motion]
[G02 Circular Interpolation Motion-Clockwise]
[G03 Circular Interpolation Motion-Counterclockwise]
[G04 Dwell (Group 00) Mill]
[G10 Set offsets (Group 00) Mill]
[G12 Circular Pocketing-Clockwise]
[G13 Circular Pocketing-Counterclockwise]
```

To keep our project simple and relatively easy to understand, the interpreter will only recognize and execute the first 4 type of commands, as it can be seen in Fig. 1.2. We are limited to only a basic window that shows the trajectory of the cutting head, so this will benefit the interpreter greatly.

The interpreter could be considered as a simple task for the conversion of G/M codes to the CNC-understandable internal data structures. However, the design and implementation of the interpreter is a large and comprehensive task because programming rules or grammar described in a programming manual and an operating concept shown in an operation manual should be considered when developing the interpreter. Therefore, the interpreter is the representative indicator that shows the design concept and the functional aspect of a CNC and is a big part of CNC as it generally spends more than 50% of the total development time to develop the interpreter.

Avoiding CNC crashing

One of the great benefits of not having an actual, physical, CNC machine is that we can easily avoid CNC **crashing**. In CNC, a "crash" occurs when the machine moves in such a way that is harmful to the machine, tools, or parts being machined, sometimes resulting in bending or breakage of cutting tools, accessory clamps, vises, and fixtures, or causing damage to the machine itself by bending guide rails, breaking drive screws, or causing structural components to crack or deform under strain.

Coordinate systems

In CNC systems, a machine coordinate system, a workpiece coordinate system, and local coordinate systems are defined for convenience when editing a part program and handling machine tools.

A machine coordinate system is defined by setting a particular point of the machine tool as the origin of a coordinate system. A workpiece coordinate system is defined by setting a particular point on the workpiece as the origin to make editing a part program easier. That is, when editing a part program using one workpiece coordinate system, we can edit the part program by defining another coordinate system based on the workpiece coordinate system. We call this secondary coordinate system a "local coordinate system".

For this project, we will use the window as the workpiece coordinate system. Since probably most of our test cases will begin from the origin, or will work only in the first quadrant, it would be wise to place the origin somewhere near the bottom left corner of the window.

Design

Why Java?

The project will be implemented in the **Java** programming language. Java is a high-level, class-based, object-oriented programming language that is designed to have as few implementation dependencies as possible. It is a general-purpose programming language intended to let programmers *write once, run anywhere* (WORA), meaning that compiled Java code can run on all platforms that support Java without the need to recompile. Java applications are typically compiled to bytecode that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++, but has fewer low-level facilities than either of them. The Java runtime provides dynamic capabilities (such as reflection and runtime code modification) that are typically not available in traditional compiled languages.

The syntax of Java is largely influenced by C++ and C. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built almost exclusively as an object-oriented language. All code is written inside classes, and every data item is an object, with the exception of the primitive data types, (i.e. integers, floating-point numbers, boolean values, and characters), which are not objects for performance reasons.

The Graphics library

The project will utilize Java's **Graphics** library, which contains the Graphics class. It is the abstract base class for all graphics contexts that allow an application to draw onto components that are realized on various devices, as well as onto off-screen images.

A Graphics object encapsulates state information needed for the basic rendering operations that Java supports. This state information includes the following properties:

- The Component object on which to draw.
- A translation origin for rendering and clipping coordinates.
- The current clip.
- The current color.
- The current font.
- The current logical pixel operation function (XOR or Paint).
- The current XOR alternation color

Coordinates are infinitely thin and lie between the pixels of the output device. Operations that draw the outline of a figure operate by traversing an infinitely thin path between pixels with a pixel-sized pen that hangs down and to the right of the anchor point on the path. Operations that fill a figure operate by filling the interior of that infinitely thin path. Operations that render horizontal text render the ascending portion of character glyphs entirely above the baseline

coordinate. All coordinates that appear as arguments to the methods of this Graphics object are considered relative to the translation origin of this Graphics object prior to the invocation of the method.

From this library the program will use the **`drawLine`** and **`drawArc`** methods, which do what you expect them to do.

- `drawLine(int x1, int y1, int x2, int y2)` - Draws a line, using the current color, between the points (x1, y1) and (x2, y2) in this graphics context's coordinate system.
- `drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)` - Draws the outline of a circular or elliptical arc covering the specified rectangle.

These two methods will be a crucial part of the program, as they are the main commands the user can write in G-code. As mentioned before, the G01 command draws a line, and G02 or G03 commands draw arcs.

MVC Architecture

To make the program more comprehensive and the code easy to understand, it will be build using the MVC architecture. **Model–view–controller** (**MVC**) is a software architectural pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.

Traditionally used for desktop graphical user interfaces (GUIs), this pattern also became popular for designing web applications. Popular programming languages have MVC frameworks that facilitate the implementation of the pattern. The role of each package is the following:

- **Model -** The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- **View –** This package contains any representation of information such as a window, button, textbox, chart, diagram or table. Multiple views of the same information are possible, like a bar chart for management and a tabular view for accountants.
- **Controller** - Accepts input and converts it to commands for the model or view.

In addition to dividing the application into these components, the model–view–controller design defines the interactions between them. The model is responsible for managing the data of the application. It receives user input from the controller. The view renders presentation of the model in a particular format. The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

Program design

The user interface of the program will look somewhat, if not exactly, like the next concept:

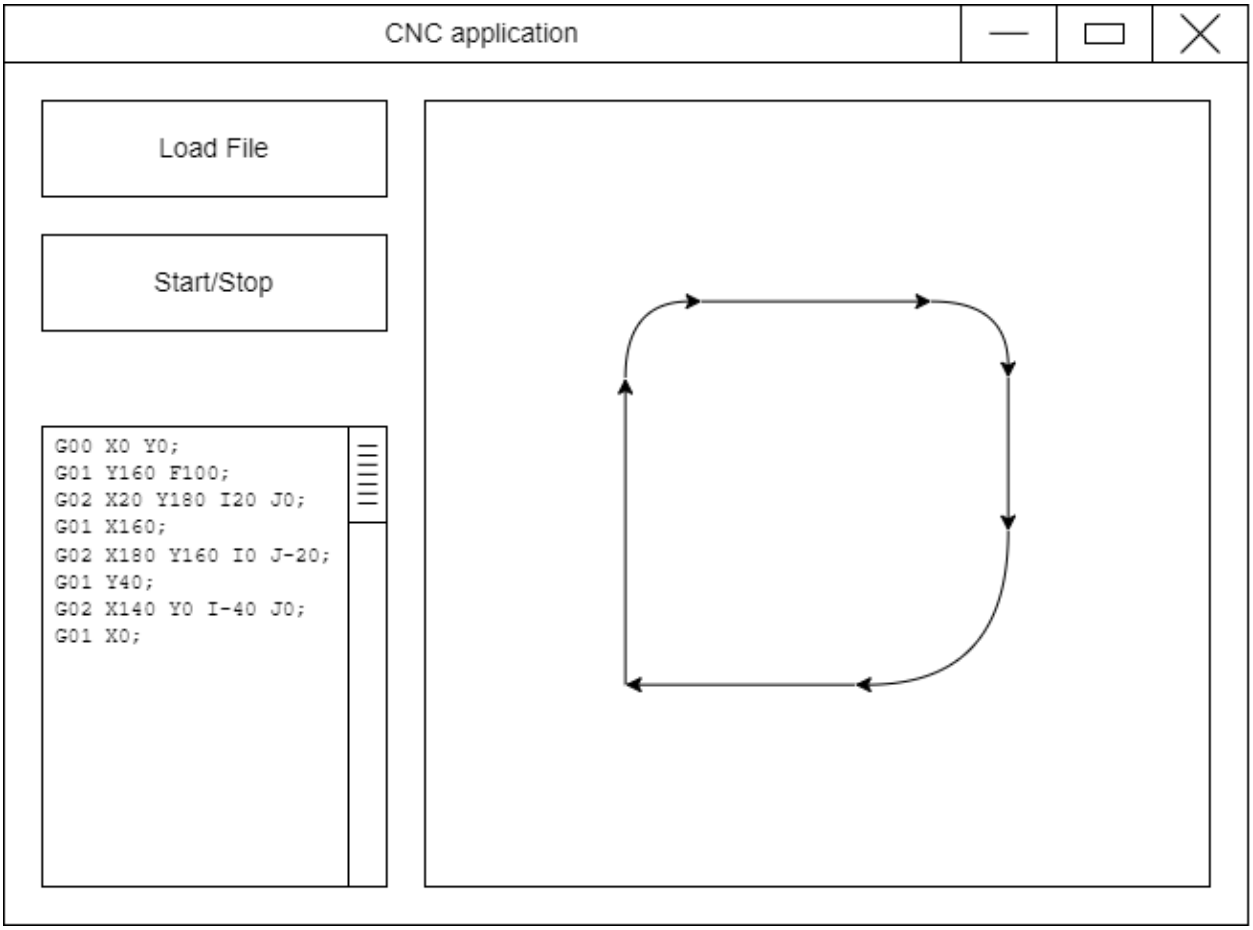

Figure 3.1. GUI concept of the CNC application

The components of the GUI will be:

- **Application Window** – The window of the application that contains all the other components. Can be minimized, maximized and closed any time. However, maximizing is not recommended, as the other components have fixed size and may not scale accordingly.
- **Load File Button** – Pressing this button will bring up the File Explorer for you to select the desired **text file** you wish the application to interpret. The file explorer should let you only select a **.txt** file. Upon successful loading a message "*file_name successfully loaded!"* will be displayed.
- **Start/Stop Button –** Button to start the simulation once the selected file is loaded. Pressing the button without a loaded file will result in an error message "*File not*

leaded!" being displayed. When the simulation starts, the text on the button should change to *"Stop"* and pressing it will stop the simulation. The text on the button will change back to *"Start"* again afterwards.

- **G-code Scroll Pane –** This pane will contain the G-code from the file so that the user can easily access and see what the code is.
- **Simulation area –** In this box you will be able to visualize the animated tracing of the given G-code commands, with a slight pause between the commands, and possibly displaying which is the current command and translating the command into human understandable language.

Implementation

Class diagram

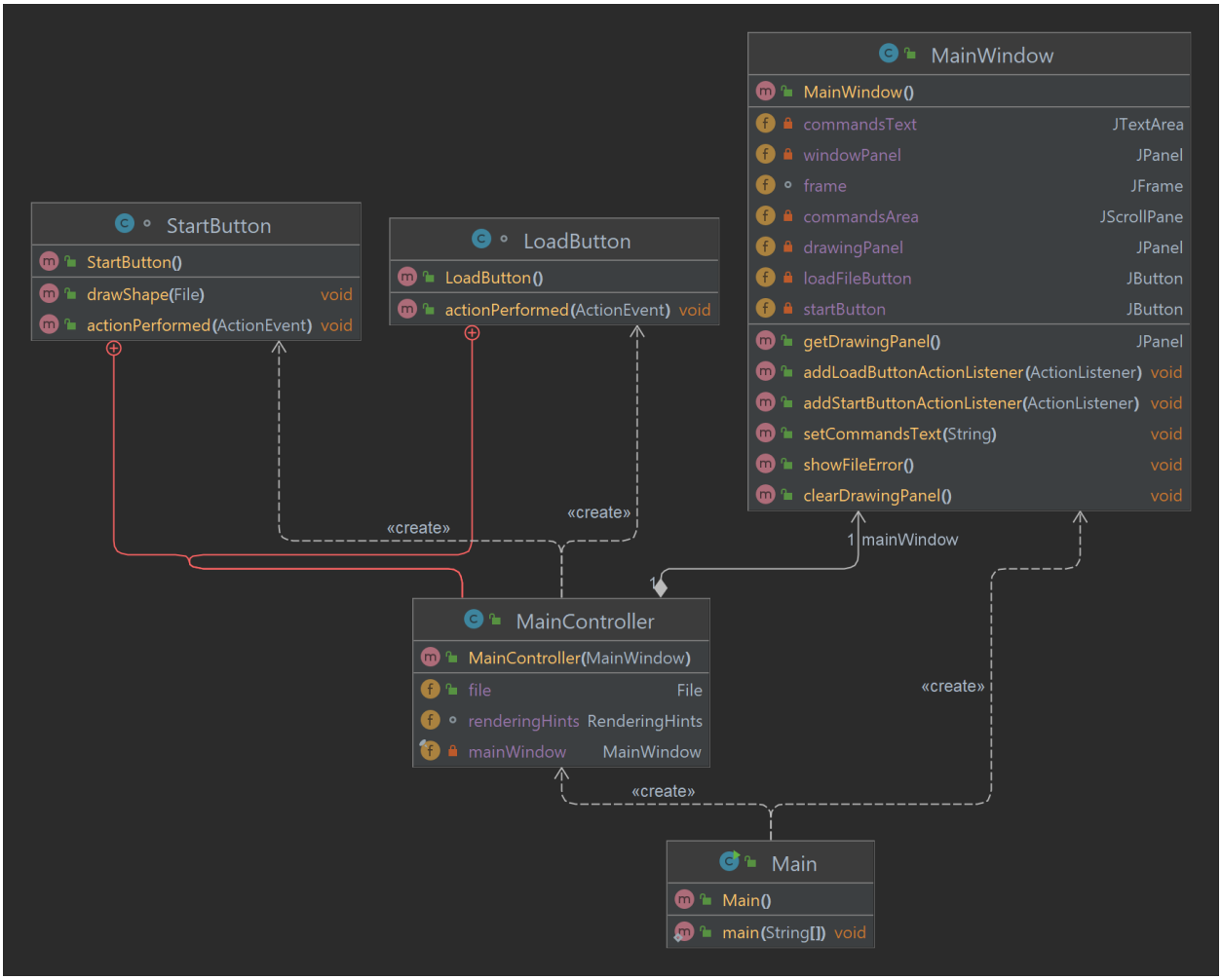

Figure 4.1. Class diagram

Code samples and explanation of the workflow

All the elements of the GUI can be found in the **MainWindow** class. The fields in this class are automatically generated by the Swing UI designer. The methods were written by me. The GUI contains two buttons, whose ActionListener implementation can be found in the **MainController.**

The **MainController** class contains all the logic of the program.

When the **LOAD** button is pressed, the following code is executed and the file is saves in the variable declared at the beginning of the class, as this file will be used later to draw the shape encoded within.

```
class LoadButton implements ActionListener {

    @Override
    public void actionPerformed(ActionEvent e) {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setCurrentDirectory(new File("."));
        int response = fileChooser.showSaveDialog(null);
        if (response == JFileChooser.APPROVE_OPTION) {
            file = new File(fileChooser.getSelectedFile().getAbsolutePath());
        }

        String allText = "";
        try {
            allText = new
String(Files.readAllBytes(Paths.get(fileChooser.getSelectedFile().getAbsolute
Path())));
        } catch (IOException ex) {
            mainWindow.showFileError();
        }

        BufferedReader reader;

        try {
            reader = new BufferedReader(new
FileReader(fileChooser.getSelectedFile().getAbsolutePath()));
            String line = reader.readLine();
            mainWindow.setCommandsText(allText);
            while (line != null) {
                line = reader.readLine();
            }

            reader.close();
        } catch (IOException ex) {
            mainWindow.showFileError();
        }
    }
}
```

This class is responsible for handling the action performed when the "Load" button is clicked. It creates a JFileChooser object and sets the current directory to the current working directory. It then opens a dialog box to allow the user to select a file. Once a file is selected, it reads the contents of the file and sets the contents of the file to the commands text area in the main window. The class also has a try-catch block to handle any IOExceptions that may be thrown when reading the file and displays an error message in the main window if an exception is thrown.

Also, in the **MainController** class is the code that handles what happens when the **START** button is clicked. When the button is clicked, the method *drawShape(File file)* is called:

```
public void drawShape(File file) {
            try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
                String line;

                Graphics2D graphics2D = (Graphics2D)
mainWindow.getDrawingPanel().getGraphics();
                graphics2D.setRenderingHints(renderingHints);
                Path2D.Double path = new Path2D.Double();
//                Arc2D.Double arc = new Arc2D.Double();

                while ((line = reader.readLine()) != null) {
                    String[] coordinates = line.split(" ");

                    if (line.startsWith("G00")) {
                        // parse the G00 command and move the cutting head to the
specified position

                        double x = Double.parseDouble(coordinates[1].substring(1));
                        double y = Double.parseDouble(coordinates[2].substring(1));
                        path.moveTo(x, y);

                    } else if (line.startsWith("G01")) {
                        // parse the G01 command and move the cutting head to the
specified position
                        double x = Double.parseDouble(coordinates[1].substring(1));
                        double y = Double.parseDouble(coordinates[2].substring(1));
                        path.lineTo(x, y);

                    } else if (line.startsWith("G02")) {
                        // parse the G02 command and draw a clockwise arc
                        double controlPointOneX = Double.parseDouble(coordinates[1]);
                        double controlPointOneY = Double.parseDouble(coordinates[2]);
                        double controlPointTwoX = Double.parseDouble(coordinates[3]);
                        double controlPointTwoY = Double.parseDouble(coordinates[4]);
                        double endX = Double.parseDouble(coordinates[5]);
                        double endY = Double.parseDouble(coordinates[6]);
                        path.curveTo(controlPointOneX, controlPointOneY,
controlPointTwoX, controlPointTwoY, endX, endY);
                    }
                }
                graphics2D.draw(path);
            } catch (IOException e) {
                mainWindow.showFileError();
            }
        }
```

This method reads the contents of the file passed to it and uses a BufferedReader to read through each line of the file. The method starts by initializing a Graphics2D object, which is used

to draw on the drawing panel of the application's main window. It also initializes a Path2D.Double object, which is used to store the path of the shape being drawn.

The method then enters a while loop, reading each line of the file one at a time. The first thing it checks is whether the line starts with "G00", which indicates a move command. If it does, the code parses the x and y coordinates from the line, and uses the *moveTo()* method of the Path2D object to move the "virtual pen" to that position without drawing. If the line starts with "G01", this indicates a draw command. The code parses the x and y coordinates from the line, and uses the *lineTo()* method of the Path2D object to draw a line from the current position to the new position specified in the line. If the line starts with "G02", this indicates an arc command. The code parses the x and y coordinates of the control points and the end point of the arc. The *curveTo()* method of the Path2D object is used to draw the arc.

Once the while loop completes and all lines of the file have been read, the *draw()* method of the Graphics2D object is called, passing in the Path2D object as an argument. This causes the shape stored in the Path2D object to be drawn on the drawing panel of the application's main window.

In the *actionPerformed* method of the **StartButton** class, the code first clears the drawing panel, before calling the *drawShape* method, passing the file that was loaded. It also checks whether any file was loaded and show appropriate message.

NOTE: I was unable to implement the traditional G02 and G03 commands. Therefore, this program uses a modified version of the G02 command *only*, which draws a curve, rather than an arc. This has as its first two arguments the coordinates of the first Bézier control point, the next two arguments are the coordinates of the second Bézier control point, and the final two arguments are the coordinates of the endpoint of the curve.

Testing

For the testing of the program, I made 3 different shapes, one combining curves and straight lines, one being only straight lines, and the last one only containing curves. All of these shapes were drawn on the canvas of the GUI according to the G-code commands.

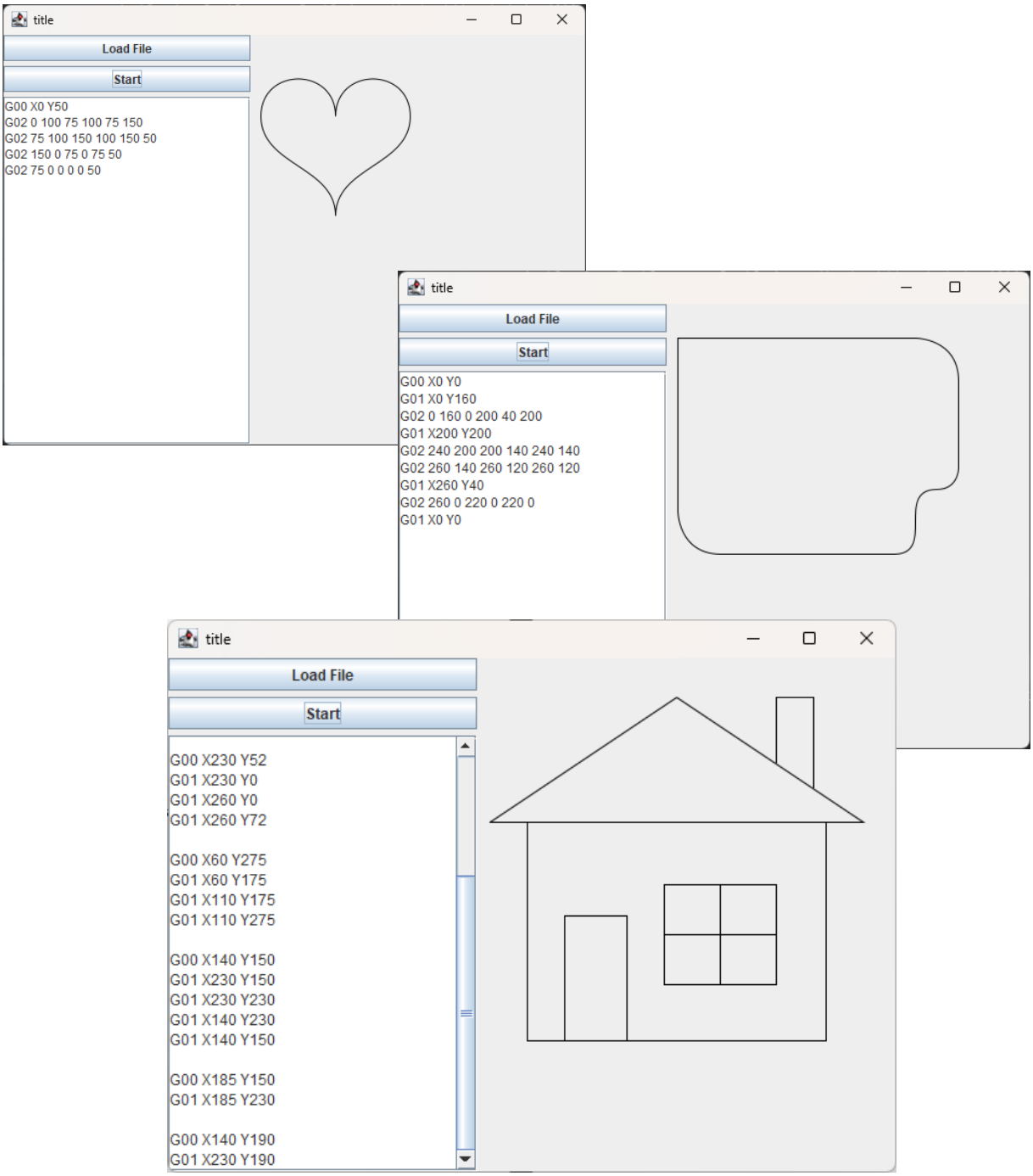

Figure 5.1. The shapes used for testing the program being drawn

Conclusions

This project serves as a good example of how you can draw shapes in Java with G-code. It helped develop the understanding of the elementary G-code commands and how they translate into shapes.

However, the program is *far from perfect*. Here are some important improvements that can be done:

- Implementation of the traditional G02 and G03 commands
- Moving the center of the coordinate system to the bottom left of the canvas
- Error checking if the provided G-code is written in the correct syntax and if the provided coordinates exceed the size of the drawing canvas
- Add animation for the drawing

Overall, I am very pleased with how it turned out and I am excited to develop it further.

Bibliography

https://en.wikipedia.org/wiki/Numerical_control

https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

https://docs.oracle.com/javase/7/docs/api/java/awt/Graphics.html

Suk-Hwan Suh, Seong Kyoon Kang, Dae-Hyuk Chung, Ian Stroud, **Theory and Design of CNC Systems**, 2008, Springer Science & Business Media, Theory and Design of CNC Systems - Suk-Hwan Suh, Seong Kyoon Kang, Dae-Hyuk Chung, Ian Stroud - Google Books