NORTHWEST NAZARENE UNIVERSITY

Cyclops
An Open Source Educational Tool for Modeling Digital Logic

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Andrew Kurtz Fillmore
2017

THESIS
Submitted to the Department of Mathematics and Computer Science
in partial fulfillment of the requirements
for the degree of
BACHELOR OF SCIENCE

Andrew Kurtz Fillmore
2017

Cyclops
An Open Source Educational Tool for Modeling Digital Logic

Author: [illegible]

Approved: [illegible]
Barry L. Myers, Ph.D., Chair,
Department of Mathematics & Computer Science
Faculty Advisor

Approved: Barbara Howard
Barbara Howard, M.A., Associate Professor
Center for Academic Success and Advising
Second Reader

Abstract

An application was created to model digital logic gates for use in an introductory course in computer science. This application was written in Java using the Swing graphics toolkit. The design of the application followed principles of object-oriented including abstract classes, interfaces, and serialization. The use of several object-oriented design patterns compounded these design principles and include the model-view-controller pattern, the observer pattern, and the abstract factory pattern. The application's features include the ability to zoom, the ability to insert new objects using onscreen buttons, the ability to drag objects around the work space arbitrarily, and the ability to save and load projects.

Contents

List of Figures

1 Introduction

Cyclops was created to fill a need in computer science education. Students and professors needed a way to explore and model digital logic with an easily accessible program. Incorporating elements of object-oriented design, Cyclops provides a simple and accessible way to teach and learn about how logic gates work. The design of Cyclops is modular and will allow changes to be made more easily in the future.

2 Design

Cyclops is a modeling tool which is primarily comprised of a graphic interface with which the user interacts. When the users executes Cyclops they will see a window with a menu bar on the top, a row of buttons on the left, a slider in the bottom right corner, and a large empty work space occupying the rest of the space. Figure 1 demonstrates this layout.

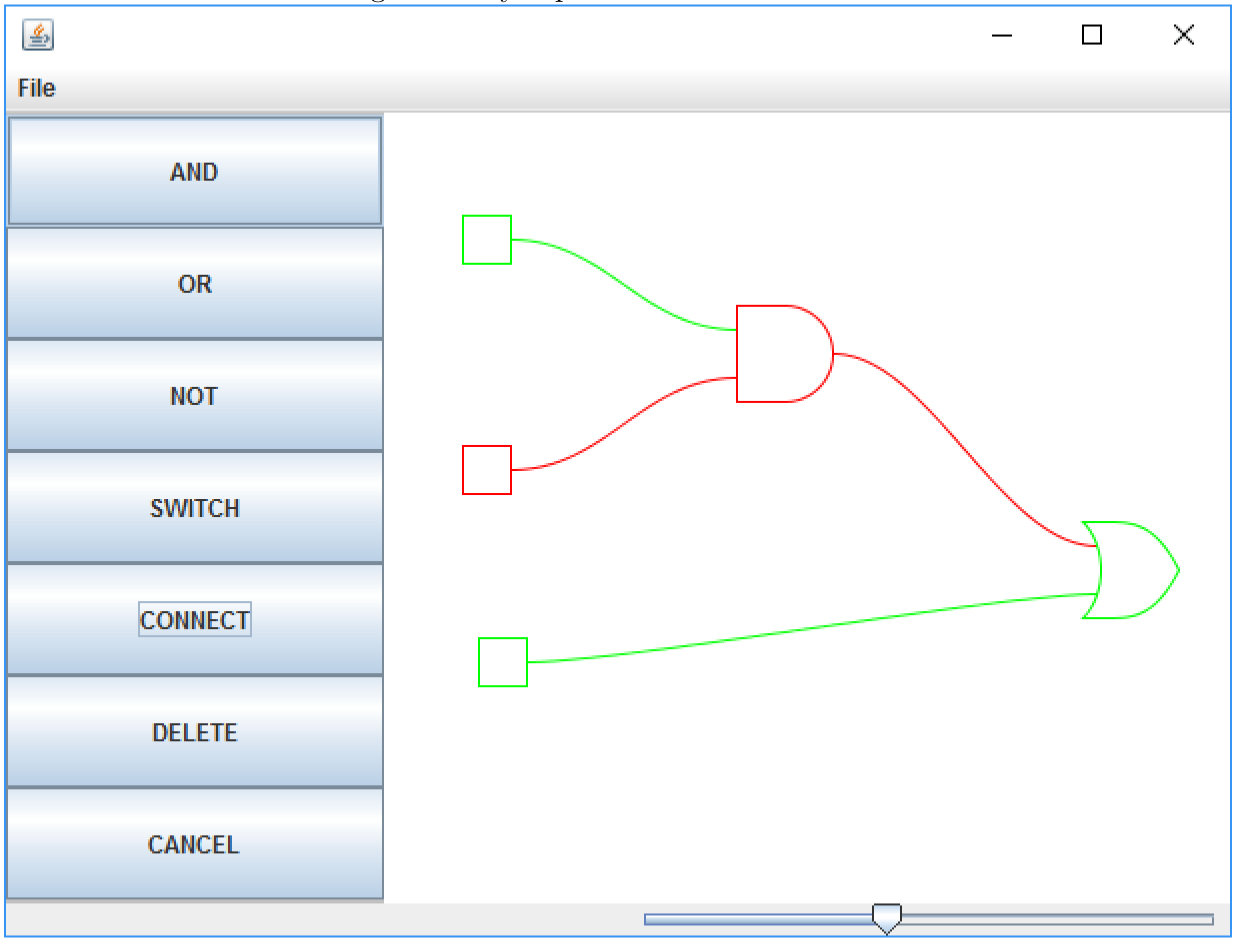
Users create gates by clicking one of the gate buttons and then placing a gate onto the work area. The slider controls how far the work space is zoomed in or out. The users can also scroll in and out with their mouse wheel. Clicking and dragging with the right mouse button causes the work space to move around. Clicking and dragging with the left mouse button on a gate will cause only that gate to be moved.

In the code, each gate is represented as its own object. The different types of gates inherit from a single Gate super-class. This provides the common functionality that all the gates share. Included are members for a gate's current state and position and methods for determining if the gate contains a given point in its area.

2.1 Design Patterns

The majority of Cyclops' design involves design patterns. Cyclops uses several design patterns including the Model-View-Controller pattern, the Subject-Observer

Figure 1: Cyclops' Visual Interface



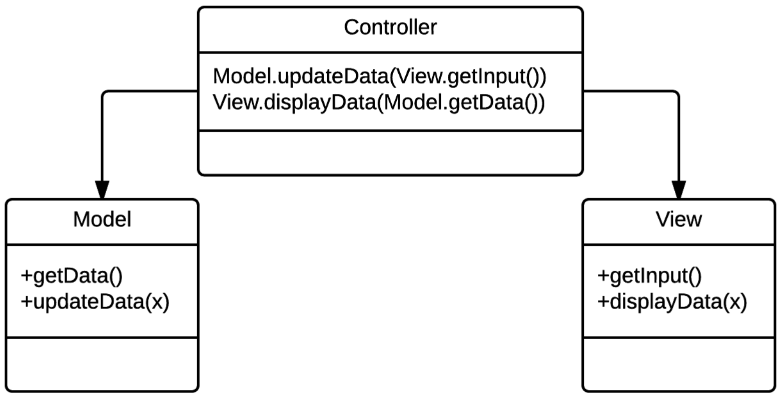
pattern, and the Abstract Factory pattern.

2.1.1 Model-View-Controller

The Model-View-Controller pattern is used to decouple the user interactions from the way data is stored and manipulated. The two parts of the program are called the view and the model respectively. The pattern's main benefit is to allow changes to be made to either the view or the model with out drastically effecting the other. For instance, if it is decided that the layout of the main window should use smaller buttons, that change can easily be made to the view without effecting how the model deals with the action of a button press.

Essentially, the view can be thought of as a dumb screen. The view should have no logic or smarts. It should only give input from the user to the model and display what the model gives back. The model, on the other hand, can be thought of as a black box. It takes input and produces output without the user being able to see into it. It takes and stores data from the view and produces data to be displayed by the view. The controller is what ties the model and the view together. It controls how data and events flow between the two parts. Figure 2 models how these three objects fit together and interact. Notice that the controller acts to pass data between the model and the view in both directions.

Figure 2: UML Diagram of Model-View-Controller Pattern



2.1.2 Subject-Observer

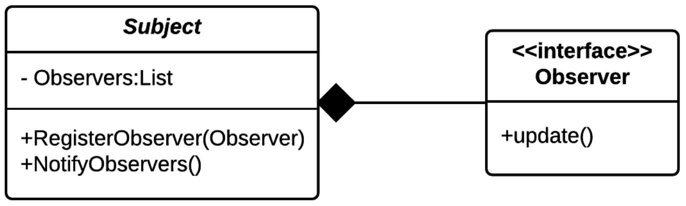
The Subject-Observer pattern in used to propagate events from a subject to the observers. In Cyclops, this is used to transfer the changing state from one gate to the next. The Subject class is an abstract class and the Observer class is an interface. Any class can inherit from these two classes.

The Subject class provides a list of Observers, a method to register Observers, and an abstract method to notify Observers. The Observer interface provides a handle to a method named update. Registering an observer simply adds that observer to the observer list. To notify all the observers, the subject will iterate through that observer list and call the update method. Thus the update method acts as a way for a subject to pass a signal to an observer. Figure 3 shows how these two classes act together.

In practice, a class will implement the Observer interface and subscribe to a subject it wants to observe. This means that the object will tell the subject to register itself. When ever an action happens that the subscribed observers need to know about the subject will call a method to notify its Observers. The update method acts as an event. When the observers are updated they will do other work based on the signal they are given.

In Cyclops, the Gate super-class implements and extends both the Observer interface and the Subject abstract class, respectively. This means that every gate acts as both a subject and an observer. A signal starts when a single gate is updated. The updated gate acts as a subject and notifies all of its observers. All of these observers will decide their state according to their own internal logic. If the gate's state changes, it then acts as a subject and passes the signal along. If it does not change then there is no reason to update its observers. Thus, the signal propagates through the network of gates. The signal stops when it reaches either a gate that does not change its state or a gate that has no observers registered.

Figure 3: UML Diagram of Subject-Observer Pattern



2.1.3 Abstract Factory

The Abstract Factory pattern is used to create new objects that all inherit from the same super class. A factory will produce new object based on the parameters it is given that can be referenced as the super-class. In Cyclops a factory is used to create new gates based on the type specified by the user. The factory class has a static get method that contains the logic to return specific objects based on the parameters it is given. This method is used in place of the `new` operator to create new objects.

2.2 Graphics

Java's Swing package provides a robust means of creating drawings on windows. Displaying an object is as simple as telling a canvas to draw it. Cyclops utilizes this functionality to display objects like gates and connections for the user to see.

2.2.1 Rendering

In order to draw an object, the view needs instructions on how to draw that particular object. But every object is drawn differently. To make it simpler for the view to do its job, every object that can be drawn implements the `IRenderable` interface. This interface provides a handle to the render method. This method will then have a different implementation for each object. This means that calling the render method on an and-gate will draw something different compared to calling the render method on an or-gate.

To draw all the different gates and connections, the model passes a list of these objects (all of them implementing the `IRenderable` interface) to the view. The view then simply iterates through the list and calls `render()` on all of the object. The view doesn't need to know any of the specifics of how these objects are drawn. It simply lets each individual object do its own drawing.

2.2.2 Transformation

In order for the user to be able to drag objects around on the screen and use the zoom feature, each object must be able to be geometrically transformed. The geometric name for these transformations are translations and dilation.

These transformations work to position objects on the screen as well as allowing them to be dragged around and zoomed in or out. Each object keeps track of where it is in space. The model also keeps track of how far zoomed-in the user is. During the rendering process the object starts out at the origin with a default scale. A transformation is applied to move it to its correct location and match its scale with the current level of zoom. A translation represented by a matrix equation where the old point A is multiplied by the transforming matrix to give A'. Equation 1 shows how this process works using homogeneous coordinates[1].

$$A' = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} A \tag{1}$$

A direct dilation or scaling can also be represented using a matrix equation in homogeneous coordinates as shown in Equation 2. Here λ represents the scale factor or the amount that the point is being dilated.

$$A' = \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} A \tag{2}$$

When an object is dragged on the screen a bit its location must be updated

[1]Homogeneous coordinates are used in graphics programming to allow complex transformations using matrices. A two-dimensional point (x, y) is represented by the three-dimensional vector $\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$.

continuously in discrete increments so that it moves along with the mouse. As the mouse moves its location is tracked. The difference between the current position and previous position of the mouse gives the current change. Adding this difference to the object's position will translate it in such a way that it follows the mouse.

When the user zooms in or out all the object must be dilated based on where the mouse is positioned. This allows for the user to zoom in on a particular location by positioning the mouse where they want to zoom in. A simple dilation would only require scaling each objects coordinates by the amount zoomed in or out. This would cause each object to zoom in and out relative to the origin in the upper left hand corner. To zoom in on the cursor, Cyclops uses a homothetic transformation. This transformation translates the object so that the cursor is effectively at the origin, performs a simple dilation, and then translates the object back to where it was. In practice, this transformation is done without matrices. The transformation is applied directly to the points to save on processing costly matrix calculations. Equations 3 and 4 show this application on a point (x, y) transforming to (x', y'). The center of transformation is a point (a, b) and the scaling factor is λ.

$$x' = a + \lambda(x - a) \tag{3}$$

$$y' = b + \lambda(y - b) \tag{4}$$

2.3 Saving and Loading

Cyclops allows users to save and load their project on the hard disk. To reduce space and complexity, the actual files saved to the disk only contain enough information to reproduce the saved project. A Data object contains all the relevant data that represents a project. The model uses this object to store the current project.

The Data class implements the Serializable interface. Saving is accomplished by serializing the Data object to the hard disk. Similarly, loading a previously saved project just de-serializes a Data object into the model.

3 Expected Use

Cyclops is intended to be used as a teaching tool. Students normally learn about digital logic in an introductory computer science course. This may be a college level course or advanced placement high school class. Cyclops will allow students to explore how these gates interact and fit together. Students can also explore the relationship between how the networks of gates they create relate to other areas of computer science. These areas may include Boolean algebra and computer architecture among others.

4 Future Work

While Cyclops is currently in a stable production state there are still features that can be added in the future. One of these features is the ability to input a expression written in Boolean algebra that Cyclops will translate into a logic gate network. This feature could be implemented with a pseudo-compiler. Most of the changes would be made to the model as the bulk of the feature involves data processing. A small text box would be added to the view in order to receive user input.

Another potential feature to add is the ability for the users to create their own custom gates with multiple inputs and outputs. The user would be able to define these custom gates based on truth tables or through previously defined gate networks. The goal is to be able let the users be more creative by giving them a way to reuse ideas.

There are also some small changes that should be made. Currently, Cyclops only implements And, Or, and Not gates. Future versions should also implement Nand, Nor, and Xor gates. There is also a known issue involving cycles within gate networks.

If a network of gates creates a cycle where a signal propagates without reaching a terminating state, an exception will be thrown. This puts the application into a broken state that is only recoverable from by deleting all the gates from the work area. A solution to this issue would involve algorithms that find cycles in directed graphs. It was determined that implementing this solution would be beyond the reasonable scope of the project.

References

[1] Deitel, P. J., & Deitel, H. M. (2015). *Java, How to Program - Early Objects* (10th ed.). Boston, Mass: Prentice Hall.

[2] Design Patterns and Refactoring. (2017). Retrieved from `https://sourcemaking.com/design_patterns`

A Appendix

A.1 Model.java

```java
import javax.swing.SwingUtilities;
import java.awt.Cursor;
import java.awt.event.MouseEvent;
import java.awt.event.MouseWheelEvent;
import java.io.File;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.nio.file.Files;
import java.util.ArrayList;
import java.util.List;

class Model {
    private Data data;
    private Gate beingDragged;
    private Gate connectionStart;
    private Gate connectionEnd;
    private double mousePrevX;
    private double mousePrevY;
    private ClickState clickState;
    private double MIN_SCALE = 5;
    private double MAX_SCALE = 50;

    Model() {
        data = new Data();
        data.setScale(10);
        clickState = ClickState.DEFAULT;
    }

    private void makeConnection(Gate from, Node to) {
        from.registerObserver(to);
        to.setInUse(true);
        data.getConnections().add(new Connection(from, to));
        from.notifyObservers();
    }

    double getScale() {
        return data.getScale();
    }

    void setScale(double scale, double focusX, double focusY) {
```

```
    for (Gate gate : data.getModelGates()) {
        gate.updateTransform(scale,
                focusX + ((scale)/ data.getScale()) *
                         (gate.currentXPos - focusX),
                focusY + ((scale)/ data.getScale()) *
                         (gate.currentYPos - focusY));
    }
    data.setScale(scale);
}

private void addGate(GateType gateType, double xPos, double yPos) {
    Gate newGate = GateFactory.getGate(gateType, xPos, yPos,
                                       data.getScale());
    data.getModelGates().add(newGate);
    newGate.checkNodes();
}

private void deleteGate(Gate gate) {
    ArrayList<Connection> connectionsToBeDeleted = new ArrayList<>();
    for (Connection connection: data.getConnections()
         ) {
        if (connection.getInput() == gate ||
            connection.getOutputGate() == gate)
        {
            connectionsToBeDeleted.add(connection);
        }
    }

    for (Connection connection:connectionsToBeDeleted
         ) {
        deleteConnection(connection);
        connection.setInput(null);
        connection.setOutput(null);
    }
    gate.setState(false);
    gate.notifyObservers();
    gate.removeAllObservers();
    data.getModelGates().remove(gate);
}

private void deleteConnection(Connection connection) {
    connection.getOutput().setInUse(false);
    data.getConnections().remove(connection);
}
```

```
void mouseClick(MouseEvent event) {
    boolean selected;
    switch (clickState)
    {
        case DEFAULT:
            for (Gate gate : data.getModelGates()) {
                gate.clicked(event.getX(), event.getY());
            }
            setClickState(ClickState.DEFAULT);
            break;
        case DELETE:
            boolean deleted = false;
            Gate gateToBeDeleted = null;
            for (Gate gate : data.getModelGates()) {
                if (gate.contains(event.getX(), event.getY()) &&
                    !deleted)
                {
                    gateToBeDeleted = gate;
                    deleted = true;
                }
            }
            if(gateToBeDeleted != null) {
                deleteGate(gateToBeDeleted);
            }
            setClickState(ClickState.DELETE);
            break;
        case CONNECTION_START:
            selected = false;
            for (Gate gate : data.getModelGates()) {
                if (gate.contains(event.getX(), event.getY()) &&
                    !selected) {
                    connectionStart = gate;
                    selected = true;
                }
            }
            setClickState(ClickState.CONNECTION_END);
            break;
        case CONNECTION_END:
            selected = false;
            for (Gate gate : data.getModelGates()) {
                if (gate.contains(event.getX(), event.getY()) &&
                    !selected &&
                    gate != connectionStart)
                {
                    connectionEnd = gate;
                    makeConnection(connectionStart,
                                   connectionEnd.getNode());
                    selected = true;
                }
            }
            if(selected)
                setClickState(ClickState.DEFAULT);
            else
                setClickState(ClickState.CONNECTION_END);
            break;
        case PLACE_AND_GATE:
            addGate(GateType.AND_GATE, event.getX(), event.getY());
            setClickState(ClickState.DEFAULT);
            break;
        case PLACE_OR_GATE:
            addGate(GateType.OR_GATE, event.getX(), event.getY());
            setClickState(ClickState.DEFAULT);
            break;
        case PLACE_NOT_GATE:
            addGate(GateType.NOT_GATE, event.getX(), event.getY());
            setClickState(ClickState.DEFAULT);
            break;
        case PLACE_SWITCH:
            addGate(GateType.SWITCH, event.getX(), event.getY());
            setClickState(ClickState.DEFAULT);
            break;
        default:
            System.err.println("Something went wrong in the
                                clickState switch in
                                Model.mouseClick");
            System.err.println("Invalid clickState");
    }
}

void mouseDown(MouseEvent event) {
    if (SwingUtilities.isLeftMouseButton(event)) {
        boolean selected;
        switch (clickState)
        {
            case DEFAULT:
                selected = false;
                for (Gate gate : data.getModelGates()) {
                    if (gate.contains(event.getX(), event.getY()) &&
                        !selected)
                    {
                        beingDragged = gate;
                        selected = true;
                    }
                    mousePrevX = event.getX();
                    mousePrevY = event.getY();
                }
                setClickState(clickState.DEFAULT);
                break;
            case DELETE:
                boolean deleted = false;
                Gate gateToBeDeleted = null;
                for (Gate gate : data.getModelGates()) {
                    if (gate.contains(event.getX(), event.getY()) &&
                        !deleted)
                    {
                        gateToBeDeleted = gate;
                        deleted = true;
                    }
                }
                if(gateToBeDeleted != null) {
                    deleteGate(gateToBeDeleted);
                }
                setClickState(ClickState.DELETE);
                break;
            case CONNECTION_START:
                selected = false;
                for (Gate gate : data.getModelGates()) {
                    if (gate.contains(event.getX(), event.getY()) &&
                        !selected)
                    {
                        connectionStart = gate;
                        selected = true;
                    }
                }
                setClickState(ClickState.CONNECTION_END);
                break;
            case CONNECTION_END:
                selected = false;
                for (Gate gate : data.getModelGates()) {
                    if (gate.contains(event.getX(), event.getY()) &&
                        !selected &&
                        gate != connectionStart)
                    {
                        connectionEnd = gate;
```

```
                        makeConnection(connectionStart,
                                       connectionEnd.getNode());
                        selected = true;
                    }
                }
                if(selected)
                    setClickState(ClickState.DEFAULT);
                else
                    setClickState(ClickState.CONNECTION_END);
                break;
            case PLACE_AND_GATE:
                addGate(GateType.AND_GATE, event.getX(), event.getY());
                setClickState(ClickState.DEFAULT);
                break;
            case PLACE_OR_GATE:
                addGate(GateType.OR_GATE, event.getX(), event.getY());
                setClickState(ClickState.DEFAULT);
                break;
            case PLACE_NOT_GATE:
                addGate(GateType.NOT_GATE, event.getX(), event.getY());
                setClickState(ClickState.DEFAULT);
                break;
            case PLACE_SWITCH:
                addGate(GateType.SWITCH, event.getX(), event.getY());
                setClickState(ClickState.DEFAULT);
                break;
            default:
                System.err.println("Something went wrong in the
                                    clickState switch in
                                    Model.mouseClick");
                System.err.println("Invalid clickState");
        }
    } else if (SwingUtilities.isRightMouseButton(event)) {
        mousePrevX = event.getX();
        mousePrevY = event.getY();
    }

}

void mouseUp(MouseEvent event) {
    if (SwingUtilities.isLeftMouseButton(event)) {
        beingDragged = null;
    }
}
```

```
void mouseDrag(MouseEvent event) {
    if (SwingUtilities.isLeftMouseButton(event)) {
        for (Gate gate : data.getModelGates()) {
            if (gate.equals(beingDragged)) {
                gate.updateTransform(data.getScale(),
                           event.getX() - mousePrevX + gate.currentXPos,
                           event.getY() - mousePrevY + gate.currentYPos);
            }
        }
    } else if (SwingUtilities.isRightMouseButton(event)) {
        for (Gate gate : data.getModelGates()) {
            gate.updateTransform(data.getScale(),
                           event.getX() - mousePrevX + gate.currentXPos,
                           event.getY() - mousePrevY + gate.currentYPos);
        }
    }
    mousePrevX = event.getX();
    mousePrevY = event.getY();
}

List<? extends IRenderable> getVectorGraphics() {
    List<IRenderable> renderables = new ArrayList<>();
    renderables.addAll(data.getConnections());
    List<Gate> ReverseModelGates = new ArrayList<>();
    for (int i = data.getModelGates().size() - 1; i >= 0; i--) {
        ReverseModelGates.add(data.getModelGates().get(i));
    }
    renderables.addAll(ReverseModelGates);
    return renderables;
}

void MouseWheel(MouseWheelEvent event) {
    double scaleChange = -event.getPreciseWheelRotation();
    if (data.getScale() + scaleChange >= MIN_SCALE &&
        data.getScale() + scaleChange <= MAX_SCALE)
    {
        for (Gate gate : data.getModelGates()) {
            gate.updateTransform(data.getScale() + scaleChange,
                    event.getX() +
                    ((data.getScale() + scaleChange)/ data.getScale()) *
                                (gate.currentXPos - event.getX()),
                    event.getY() +
                    ((data.getScale() + scaleChange)/ data.getScale()) *
                                (gate.currentYPos - event.getY()));
        }
        data.setScale(data.getScale() +scaleChange);
    }
}

void setClickState(ClickState clickState) {
    this.clickState = clickState;
}

Cursor getCurrentCursor() {
    switch (clickState) {
        case DEFAULT:
            return new Cursor(Cursor.DEFAULT_CURSOR);
        case PLACE_AND_GATE:
        case PLACE_OR_GATE:
        case PLACE_NOT_GATE:
        case PLACE_SWITCH:
        case CONNECTION_START:
        case CONNECTION_END:
            return new Cursor(Cursor.CROSSHAIR_CURSOR);
        case DELETE:
            return new Cursor(Cursor.HAND_CURSOR);
        default:
            return new Cursor(Cursor.DEFAULT_CURSOR);
    }

}

boolean save(File file)
{

    ObjectOutputStream objectOutputStream;
    try {
         objectOutputStream =
             new ObjectOutputStream(
                 Files.newOutputStream(file.toPath()));
         objectOutputStream.writeObject(data);
         objectOutputStream.close();
    }
    catch (IOException ioException) {
        System.err.println("Error writing to file");
        System.err.println(file.toString());
        System.err.println(ioException.toString());
        System.err.println(ioException.getMessage());
        return false;
    }
```

```
        return true;
    }

    boolean open(File file)
    {
        ObjectInputStream objectInputStream;
        try
        {
            objectInputStream =
                new ObjectInputStream(
                    Files.newInputStream(file.toPath()));
            data = (Data) objectInputStream.readObject();
        }
        catch (IOException ioException) {
            System.err.println("Error reading file");
            System.err.println(file.toString());
            System.err.println(ioException.toString());
            System.err.println(ioException.getMessage());
            return false;
        }
        catch (ClassNotFoundException classNotFoundException)
        {
            System.err.println("Error reading file");
            System.err.println(file.toString());
            System.err.println(classNotFoundException.toString());
            System.err.println(classNotFoundException.getMessage());
            return false;
        }
        return true;
    }
}
```

A.2 View.java

```
import javax.swing.*;
import javax.swing.event.ChangeListener;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.Rectangle2D;
import java.io.File;
import java.util.ArrayList;
import java.util.List;

class View extends JFrame {
```

```
private Canvas drawingCanvas;
private Canvas ButtonCanvas;
private JButton b_PlaceAndGate;
private JButton b_PlaceOrGate;
private JButton b_PlaceNotGate;
private JButton b_PlaceSwitch;
private JButton b_MakeConnection;
private JButton b_Delete;
private JButton b_CancelPlace;
//private JMenuItem addGate;
private JSlider ZoomSlider;
private JMenuItem m_open;
private JMenuItem m_save;


View() {
    this.setName("Name");
    this.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
    this.setSize(300,200);
    this.setLayout(new GridBagLayout());
    this.setBackground(Color.black);

    ButtonCanvas = new Canvas();
    ButtonCanvas.setBackground(Color.LIGHT_GRAY);
    ButtonCanvas.setLayout(new GridLayout(7, 1));
    drawingCanvas = new Canvas();
    drawingCanvas.setBackground(Color.WHITE);

    ZoomSlider = new JSlider(5, 50);

    GridBagConstraints constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.BOTH;
    constraints.gridx = 0;
    constraints.gridy = 0;
    constraints.gridwidth = 1;
    constraints.gridheight = 1;
    constraints.weightx = 0;
    constraints.weighty = 0;
    constraints.ipadx = 100;
    this.add(ButtonCanvas, constraints);

    constraints = new GridBagConstraints();
    constraints.fill = GridBagConstraints.BOTH;
    constraints.gridx = 2;
    constraints.gridy = 0;
```

```
        constraints.gridwidth = 2;
        constraints.gridheight = 1;
        constraints.weightx = 400;
        constraints.weighty = 500;
        constraints.ipadx = 100;
        constraints.ipady = 100;
        this.add(drawingCanvas, constraints);

        constraints = new GridBagConstraints();
        constraints.fill = GridBagConstraints.VERTICAL;
        constraints.anchor = GridBagConstraints.EAST;
        constraints.gridx = 3;
        constraints.gridy = 1;
        constraints.gridwidth = 1;
        constraints.gridheight = 1;
        constraints.weightx = 0;
        constraints.weighty = 0;
        constraints.ipadx = 100;
        this.add(ZoomSlider, constraints);

        b_PlaceAndGate = new JButton("AND");
        b_PlaceOrGate = new JButton("OR");
        b_PlaceNotGate = new JButton("NOT");
        b_PlaceSwitch = new JButton("SWITCH");
        b_MakeConnection = new JButton("CONNECT");
        b_Delete = new JButton("DELETE");
        b_CancelPlace = new JButton("CANCEL");
        ButtonCanvas.add(b_PlaceAndGate);
        ButtonCanvas.add(b_PlaceOrGate);
        ButtonCanvas.add(b_PlaceNotGate);
        ButtonCanvas.add(b_PlaceSwitch);
        ButtonCanvas.add(b_MakeConnection);
        ButtonCanvas.add(b_Delete);
        ButtonCanvas.add(b_CancelPlace);

        JMenuBar menuBar = new JMenuBar();
        JMenu file = new JMenu("File");
        m_save = new JMenuItem("Save");
        file.add(m_save);
        m_open = new JMenuItem("Open");
        file.add(m_open);
        menuBar.add(file);

        this.setJMenuBar(menuBar);
    }
```

```
double getScale() {
    return ZoomSlider.getValue();
}

void setScale(double scale) {
    ZoomSlider.setValue((int) scale);
}

Rectangle2D getCanvasBounds() {
    return new Rectangle2D.Double(0, 0,
                                  drawingCanvas.getWidth(),
                                  drawingCanvas.getHeight());
}

void addMouseListenerDrawingCanvas(MouseListener mouseListener) {
    drawingCanvas.addMouseListener(mouseListener);
}

void addMouseMotionListenerDrawingCanvas(
                            MouseMotionListener mouseMotionListener) {
    drawingCanvas.addMouseMotionListener(mouseMotionListener);
}

void addMouseWheelListenerDrawingCanvas(
                            MouseWheelListener mouseWheelListener) {
    drawingCanvas.addMouseWheelListener(mouseWheelListener);
}

void addChangeListenerZoomSlider(ChangeListener changeListener) {
    ZoomSlider.addChangeListener(changeListener);
}

void addActionListenerPlaceAndGate(ActionListener actionListener) {
    b_PlaceAndGate.addActionListener(actionListener);
}

void addActionListenerPlaceOrGate(ActionListener actionListener) {
    b_PlaceOrGate.addActionListener(actionListener);
}

void addActionListenerPlaceNotGate(ActionListener actionListener) {
    b_PlaceNotGate.addActionListener(actionListener);
}
```

```
void addActionListenerPlaceSwitch(ActionListener actionListener) {
    b_PlaceSwitch.addActionListener(actionListener);
}

void addActionListenerMakeConnection(ActionListener actionListener) {
    b_MakeConnection.addActionListener(actionListener);
}

void addActionListenerDelete(ActionListener actionListener) {
    b_Delete.addActionListener(actionListener);
}

void addActionListenerCancelPlace(ActionListener actionListener) {
    b_CancelPlace.addActionListener(actionListener);
}

void addActionListenerOpen(ActionListener actionListener) {
    m_open.addActionListener(actionListener);
}

void addActionListenerSave(ActionListener actionListener) {
    m_save.addActionListener(actionListener);
}

void Update(List<? extends IRenderable> shapes, Cursor cursor) {
    Render(shapes);
    setCursor(cursor);
}

private void Render(List<? extends IRenderable> shapes) {
    drawingCanvas.Render(shapes);
}

public void setCursor(Cursor cursor) {
    drawingCanvas.setCursor(cursor);
    ButtonCanvas.setCursor(cursor);
}

public File chooseFile()
{
    JFileChooser fileChooser = new JFileChooser();
    int result = fileChooser.showOpenDialog(this);
    if (result == JFileChooser.CANCEL_OPTION)
    {
        return null;
        }
        return fileChooser.getSelectedFile();
    }

    private class Canvas extends JPanel {
        private List<? extends IRenderable> shapes = new ArrayList<>();

        @Override
        public void paintComponent(Graphics g) {
            Graphics2D g2 = (Graphics2D) g;
            RenderingHints rh =
                new RenderingHints(RenderingHints.KEY_ANTIALIASING,
                                   RenderingHints.VALUE_ANTIALIAS_ON);
            g2.setRenderingHints(rh);
            super.paintComponent(g2);
            for (IRenderable shape : shapes) {
                shape.render(g2);
            }
        }

        void Render(List<? extends IRenderable> shapes) {
            this.shapes = shapes;
            this.repaint();
        }
    }
}
```

A.3 Controller.java

```
import javax.swing.*;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import java.awt.event.*;
import java.io.File;

class Controller {
    private View view;
    private Model model;

    Controller(View view, Model model) {
        this.view = view;
        this.model = model;

        this.view.addMouseListenerDrawingCanvas(
            new CanvasMouseListener());
        this.view.addMouseMotionListenerDrawingCanvas(
            new CanvasMouseMotionListener());
        this.view.addMouseWheelListenerDrawingCanvas(
            new CanvasMouseWheelListener());
        this.view.Update(model.getVectorGraphics(),
                         model.getCurrentCursor());
        this.view.addChangeListenerZoomSlider(
            new ZoomSliderChangeListener());

        this.view.addActionListenerPlaceAndGate(
            new PlaceAndGateActionListener());
        this.view.addActionListenerPlaceOrGate(
            new PlaceOrGateActionListener());
        this.view.addActionListenerPlaceNotGate(
            new PlaceNotGateActionListener());
        this.view.addActionListenerPlaceSwitch(
            new PlaceSwitchActionListener());
        this.view.addActionListenerMakeConnection(
            new MakeConnectionActionListener());
        this.view.addActionListenerDelete(
            new DeleteActionListener());
        this.view.addActionListenerCancelPlace(
            new CancelPlaceActionListener());
        this.view.addActionListenerOpen(
            new OpenActionListener());
        this.view.addActionListenerSave(
            new SaveActionListener());

        this.view.setScale(model.getScale());
    }

    private class CanvasMouseListener implements MouseListener {
        @Override
        public void mouseClicked(MouseEvent event) {
            model.mouseClick(event);
            view.Update(model.getVectorGraphics(),
                        model.getCurrentCursor());
        }

        @Override
        public void mousePressed(MouseEvent event) {
            model.mouseDown(event);
            view.Update(model.getVectorGraphics(),
                        model.getCurrentCursor());
        }
    @Override
    public void mouseReleased(MouseEvent event) {
        model.mouseUp(event);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }

    @Override
    public void mouseEntered(MouseEvent event) {}

    @Override
    public void mouseExited(MouseEvent event) {}
}

private class CanvasMouseMotionListener implements MouseMotionListener {
    @Override
    public void mouseDragged(MouseEvent event) {
        model.mouseDrag(event);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }

    @Override
    public void mouseMoved(MouseEvent event) {}
}

private class CanvasMouseWheelListener implements MouseWheelListener {
    @Override
    public void mouseWheelMoved(MouseWheelEvent event) {
        model.MouseWheel(event);
        view.setScale(model.getScale());
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class PlaceAndGateActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        model.setClickState(ClickState.PLACE_AND_GATE);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());

    }
}

private class PlaceOrGateActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        model.setClickState(ClickState.PLACE_OR_GATE);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class PlaceNotGateActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        model.setClickState(ClickState.PLACE_NOT_GATE);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class PlaceSwitchActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        model.setClickState(ClickState.PLACE_SWITCH);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class MakeConnectionActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent)
    {
        model.setClickState(ClickState.CONNECTION_START);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class DeleteActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        model.setClickState(ClickState.DELETE);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class CancelPlaceActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        model.setClickState(ClickState.DEFAULT);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}

private class OpenActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        File file = view.chooseFile();
        if(file != null) {
            model.open(file);
            view.setScale(model.getScale());
            view.Update(model.getVectorGraphics(),
                        model.getCurrentCursor());
        }
    }
}

private class SaveActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent actionEvent) {
        File file = view.chooseFile();
        if (file != null) {
            model.save(file);
            view.Update(model.getVectorGraphics(),
                        model.getCurrentCursor());
        }
    }
}

private class ZoomSliderChangeListener implements ChangeListener {
    @Override
    public void stateChanged(ChangeEvent changeEvent) {
        double middleX = (view.getCanvasBounds().getMaxX() +
                          view.getCanvasBounds().getMinX()) / 2;
        double middleY = (view.getCanvasBounds().getMaxY() +
                          view.getCanvasBounds().getMinY()) / 2;
        model.setScale(view.getScale(), middleX, middleY);
        view.Update(model.getVectorGraphics(),
                    model.getCurrentCursor());
    }
}
}
```