

Analysis

Goal

This program will be an educational tool to help users understand how computers work. It will do this by allowing the user to create simulations of digital circuits by dragging and dropping virtual logic gates, the low-level building-blocks of computers. By combining many of these gates together, the program will be capable of simulating advanced circuits, such as an arithmetic logic unit (ALU) or even a primitive computer.

Target Audience

The target audience for this program is anyone who would like to know more about how computers work, but mainly students studying an A-Level in Computer Science. It will be helpful to these students because after using it they will understand logic gates better, which may lead to them performing better in their exams.

Stakeholders

The main stakeholder in the development of this program will be John Smith, a Computer Science A-Level student. John has difficulty with the boolean logic and logic gates section of the course and has said that if there was a way to experiment with logic gates and see for himself how they are able to be composed into complex circuits he would understand it better and achieve better grades.

During the development of this program, I will communicate with John; he will give me feedback and suggest ideas that he thinks should be implemented in the program.

Justification for a Computer-Based Solution

Someone could learn about digital circuitry by wiring together real logic gates, but this would not be suitable for a Computer Science A-Level student. The table below lists issues with learning about digital circuits by building a physical circuit, and how abstractions provided by the computer-based solution solves them.

Physical Circuit	Computer-Based Solution
The size of a circuit is limited by the number of available components, therefore cost.	There will be no limit on the number of components in the circuit – the only cost for more is increased processing time to run the simulation.
The user must ensure that they provide power to the circuit with the correct voltage and current, or they will damage the components.	The voltage and current are irrelevant to the actual logic, so the program abstracts them away by only simulating digital signals: 0 and 1.
It is very difficult to deduce what a circuit does just by looking at it because all the user would see is ICs wired together.	In this program, a circuit will be represented as a graph of logic gate symbols, connected by lines showing where the wires would be.
If a large circuit uses a common arrangement of components many times, it can become difficult to understand what the circuit is doing.	The program will allow the user to create virtual ICs that behave like the contained logic gates. For example, a computer built inside the program will

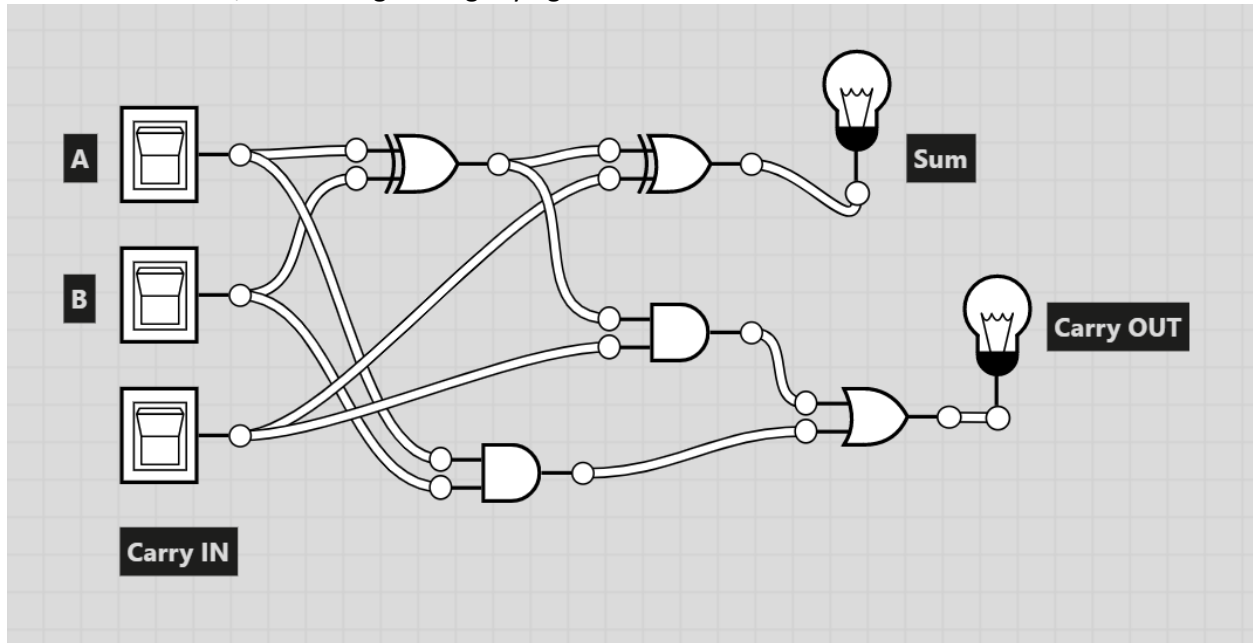
	have lots of copies of a RAM IC that can store one bit of information.
--	--

Research

Logic.ly

Logic.ly is a web-based digital logic simulator. One feature from Logic.ly that I want to include in my program is the ability to create integrated circuits.

Here is a full adder built in Logic.ly. A full adder is a circuit that adds 2 1-bit numbers giving the result as a 2-bit number, with the high bit signifying if there was an overflow.

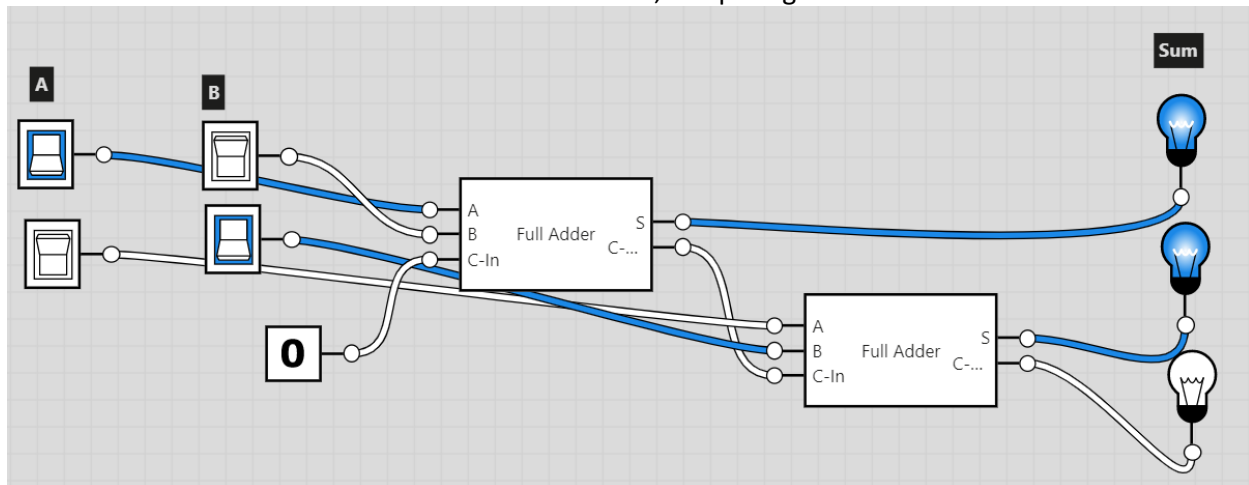


Full adders can be chained together to make circuits to add larger numbers by wiring the carry out of each adder to the carry in of the next one. This could be done by copying and pasting full adders, but that would be a bit like copying and pasting sections of code that are needed a lot: it is better to use a function. The logic gate equivalent of a function is an integrated circuit.

Using ICs in logic circuits has similar benefits to using functions in code:

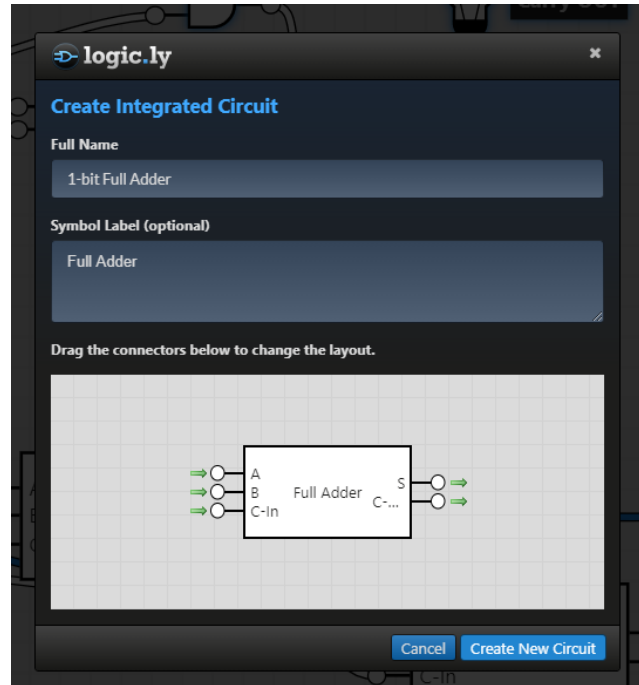
- Reduced gate duplication.
- If a section of the circuit needs to be changed, it is automatically changed everywhere it is used.
- The meaning of the whole circuit is clearer because of the labels on the ICs.

Shown below is a 2-bit adder built from full adder ICs, computing $1 + 2 = 3$.



In Logic.ly it is very easy to create an IC: select the desired components and press the “Create Integrated Circuit” button, then the UI on the right appears.

One problem with the IC creation process in Logic.ly is that you must set the labels of the inputs and outputs in the main circuit graph. In my program you will be able to change them in the IC creation UI.



Nandgame

Nandgame is an online game about building a CPU starting with only NAND gates. On each level, the user builds a circuit for a specific function using circuits from previous levels. Unfortunately for the curious student, the curriculum is enforced strictly and there is no way in Nandgame to get a blank canvas on which to experiment and build any circuit they please.

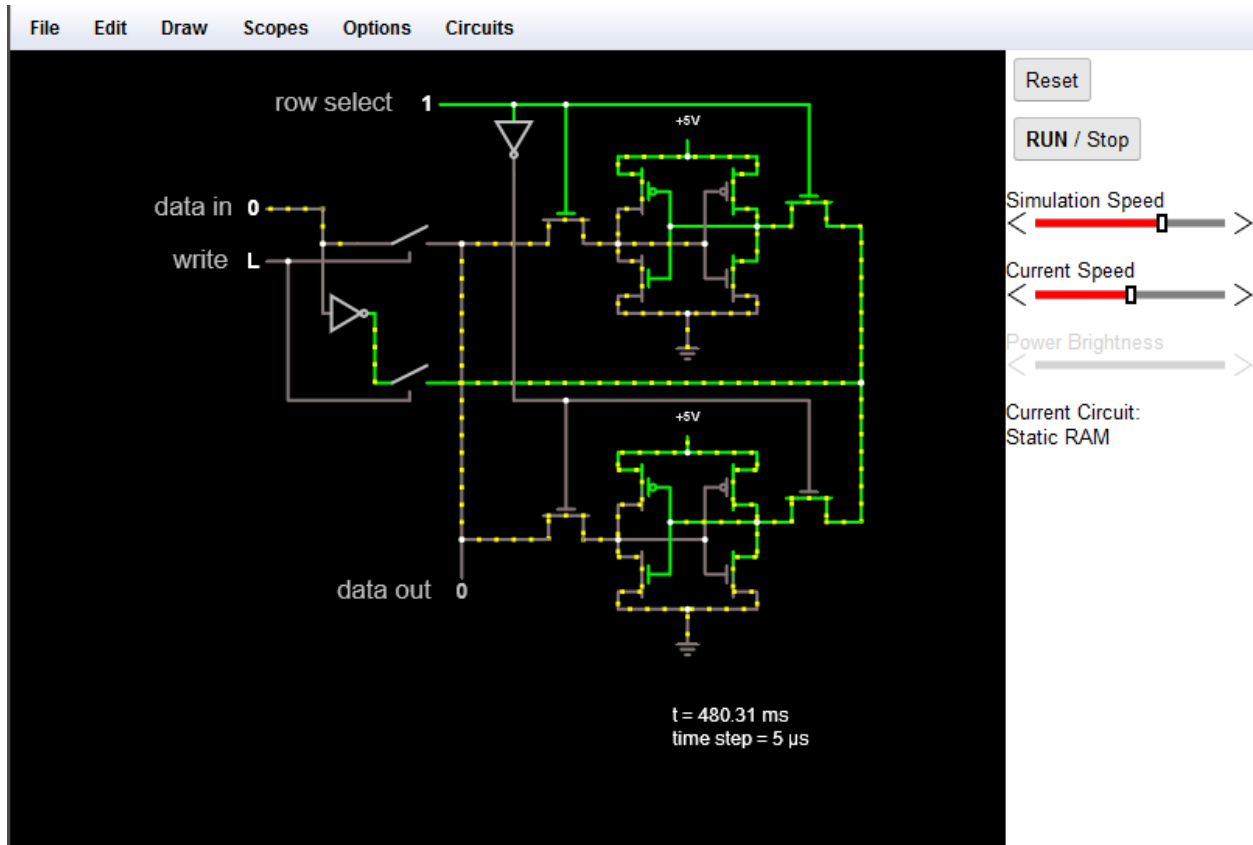
Considering that the purpose of my program is to be an educational tool, built-in lessons would be a good feature to include. However, I do not think this will be achievable in the timeframe available to develop the project.

Shown below is the level select screen of Nandgame.

Logic Gates	Arithmetics	Plumbing	Memory	Arithmetic Logic Unit	Processor
▶ Invert	🔒 Half Adder	🔒 Selector	🔒 Latch	🔒 Unary ALU	🔒 Combined Memory
🔒 And	🔒 Full Adder	🔒 Switch	🔒 Data Flip-Flop	🔒 ALU	🔒 Instruction Decoder
🔒 Or	🔒 Multi-bit Adder		🔒 Register	🔒 Opcodes	🔒 Control Unit
🔒 Xor	🔒 Increment		🔒 Counter	🔒 Condition	🔒 Program Engine
	🔒 Subtraction		🔒 RAM		🔒 Computer
	🔒 Equal to Zero				🔒 Input and Output
	🔒 Less than Zero				

Falstad

Falstad is a general-purpose electrical circuit simulator that also includes digital logic specific components. Falstad simulates voltage and current, which are beyond the scope of the program I will be creating, so I will not replicate this feature.



A screenshot of a user using Falstad, showing a RAM circuit that can store 2 bits of information.

Features of Proposed Solution

- The main interface of the program will be a canvas into which the user can drag and drop logic gates.
- Each output of the components will have a handle that can be dragged onto the input of another component. A line will then be drawn between the output and input to show that the connection was successful.
- The program will contain commands common in document editors:
 - Copy/Cut/Paste
 - Undo/Redo
 - Delete
 - Pan the view
 - Zoom in and out
 - Select multiple items and move/delete

Hardware Requirements

The program will be written in C#, so the user's computer must meet the .NET Framework system requirements.

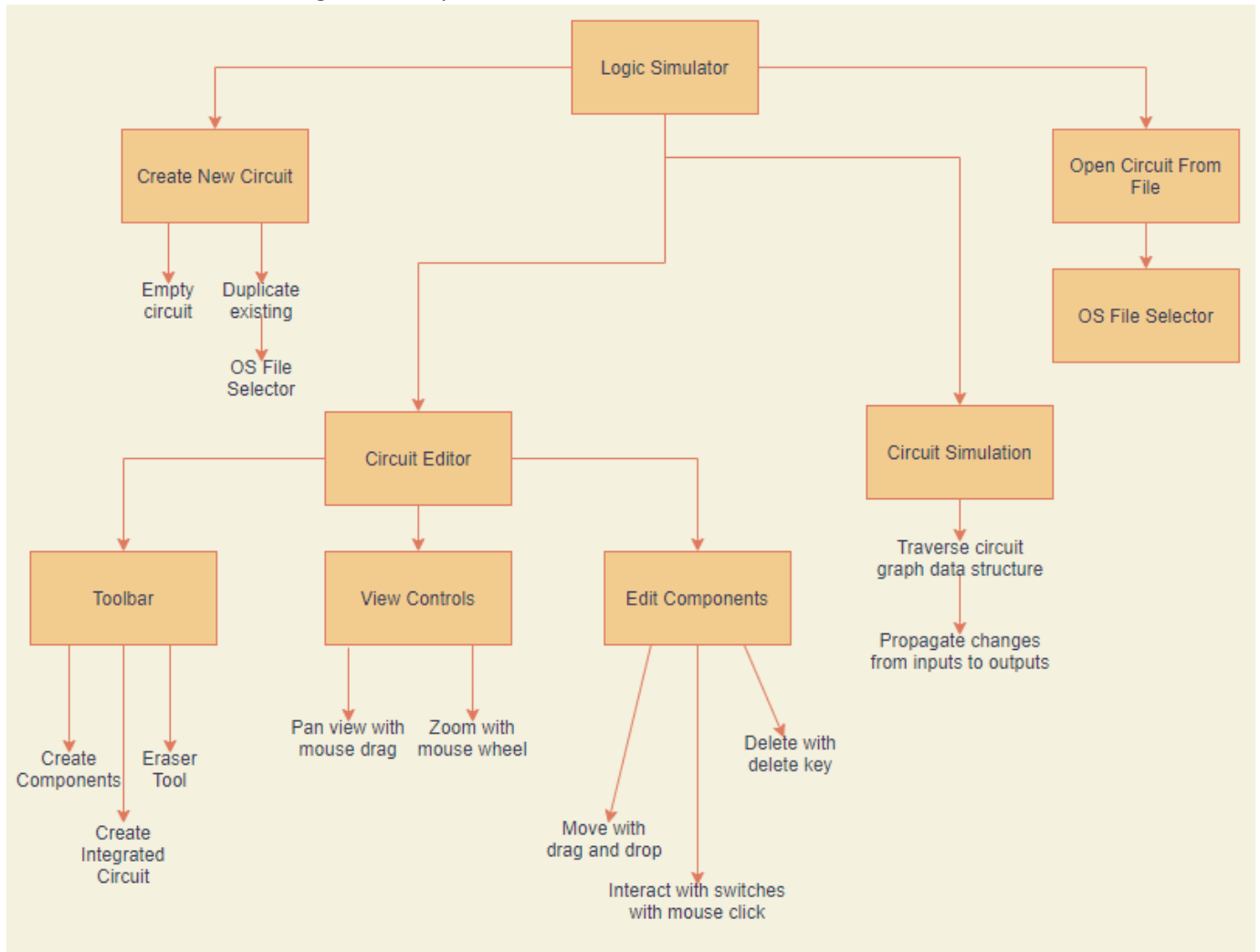
- Windows with .NET Framework
- 1 GHz CPU
- 512 MB RAM
- 4.5 GB free disk space

Success Criteria

- The user must be able to save and load circuits in files so that they can use them later.
- The program must be able to simulate circuits with deterministic results.
- The program should support sequential (stateful) logic as well as the simpler combinational logic.
- The program will include a clock component to periodically advance a stateful circuit.
- The program will include input components such as buttons, to allow the user to interact with the circuit.
- The program will include output components such as a light, to show the user the result of the circuit's calculation.
- The program must have the capability to create virtual integrated circuits.
- The program must be able to simulate complex circuits (such as a large adder, or memory) without crashing or locking up the UI.

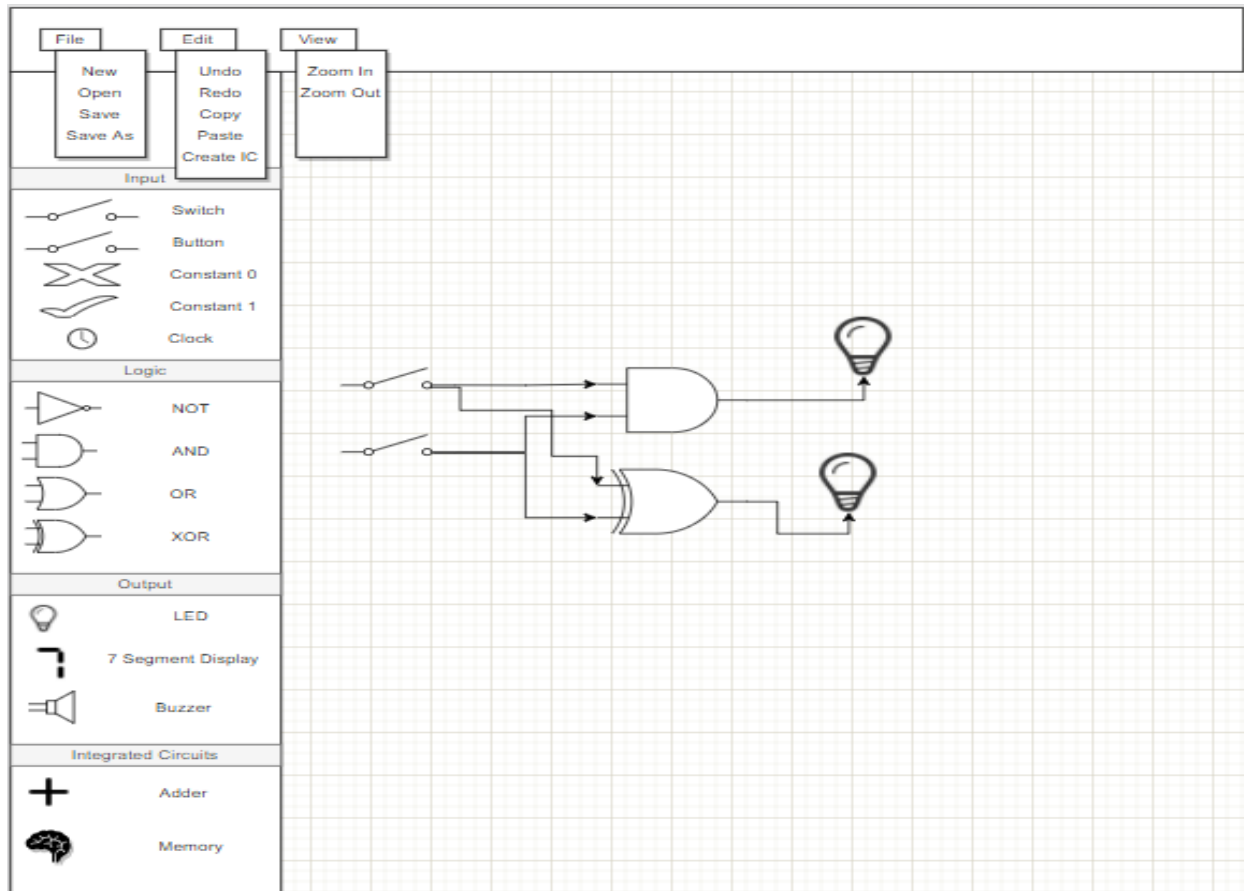
Design

In order to make the implementation of the program as easy as possible, the design will be planned out ahead of time. I will follow the principle of decomposition and abstraction by breaking down the project into smaller and more manageable components.



Circuit Viewer

The circuit viewer is the main interface of the program, allowing the user to edit the circuit and use interactive components like buttons to observe how the circuit reacts to input. The circuit is displayed as a logic diagram on an infinite plane, so the user can choose how to spatially arrange components to make it easier to understand the circuit.



The circuit will be stored in a directed graph (DG) data structure. The nodes of the graph are logic gates and other components. The edges of the graph are the wires between components.

Using a DG allows complex circuits with stored state (sequential logic) to be represented, which is needed for the user to be able to build advanced circuits such as memory, counters, or shift registers. Not including this functionality would greatly diminish the usefulness of the program, so a directed graph must be used.

The circuit viewer will contain an algorithm to evaluate the outputs of a circuit. The algorithm uses two stages of breadth first traversal. The algorithm runs every time an input node changes state, with that input as the root node. The first stage constructs the Potential Evaluation Set (PES). The PES is the set of all nodes that could change state in response to the input.

The second stage evaluates the nodes. When each node is updated, it checks if all of its inputs that are in the PES have already been updated. If this condition is not true, temporarily skip updating the node. The inputs will subsequently be updated by the algorithm and the original node will be returned to and updated.

```
function evaluateCircuit(Node root)
    // Stage 1: Construct potential evaluate set

    pes = new Set()
    queue = new Queue()
    for output in root.outputs
        queue.enqueue(output)
    end
    while !queue.empty()
        node = queue.dequeue()
        pes.add(node)
        for output in node.outputs
            if !pes.contains(output)
                queue.enqueue(output)
            end
        end
    end

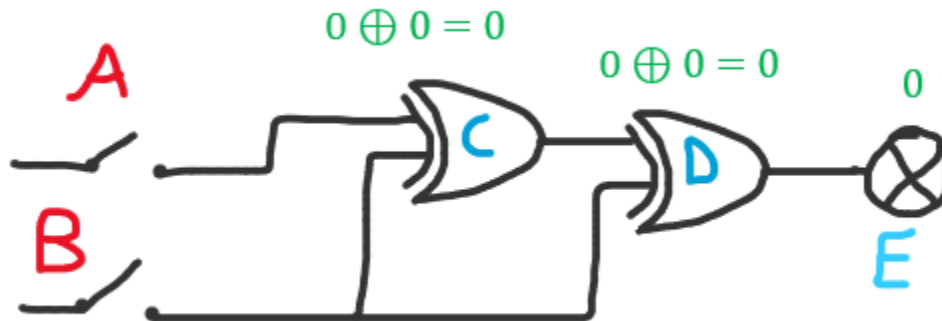
    // Stage 2: Evaluate
    queue = new Queue()
    visited = new Set()
    for output in root.outputs
        queue.enqueue(output)
    end
    while !queue.empty()
        node = queue.dequeue()
        doUpdate = true
        for input in node.inputs
            if pes.contains(input) && !visited.contains(input)
                doUpdate = false
                break
            end
        end
        if !doUpdate
            break
        end

        evaluateNode(node)
        visited.add(node)

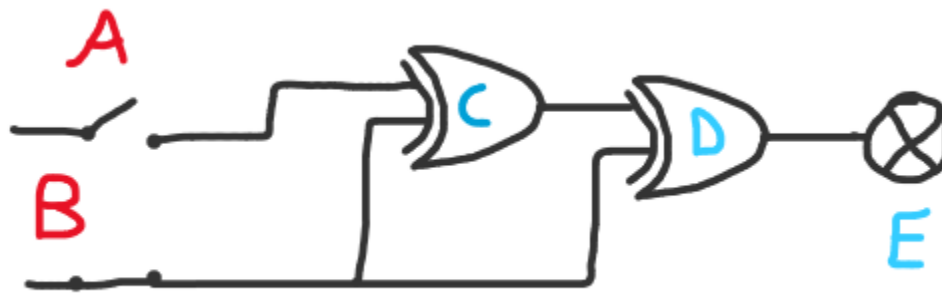
        for output in node.outputs
            if !visited.contains(output)
                queue.enqueue(output)
            end
        end
    end
end
end
10
```

Algorithm Example

The switches A and B start open, so through basic Boolean logic you can see that the light E is initially off.



The switch B is closed, so the root of the algorithm will be B.



Stage 1

This is a trace table of Stage 1: Construct Potential Evaluate Set. This stage shows that the only components that can be affected by B changing are C, D, and E. We can see this is true because obviously A cannot be affected by B, as it is an input element as well. When the queue is empty, the PES contains all possible nodes that can be reached from B, so Stage 1 is over.

root	queue	pes	node
B	C, D	Empty	
	D	C	C
	E	C, D	D
	Empty	C, D, E	E

Stage 2

This is a trace table of Stage 2: Evaluate. In this example D is the first node visited. However D cannot be updated yet because one of its inputs (C) is in the PES but has not been updated. The next node to visit it

C. C can be updated because neither of its inputs are in the PES. C then adds its output node (D) to the queue. D is visited again and this time it can be updated.

It is possible that if the program was evaluating this circuit C would be visited first, skipping the extra initial D visit. The order the nodes are visited in is dependent on the order they are stored in memory. However the order and result of the evaluation is always the same.

queue	visited	node	doUpdate
D, C	Empty		
C		D	false
D	C	C	true
E	C, D	D	true
Empty	C, D, E	E	true

Node Delete Algorithm

In a circuit, a node may be connected to other components in complicated ways involving many wires. This means that deleting a node from the circuit is not as simple as just removing it from the list of components. Every input of the node must be checked. If the input is connected, the reference to that input must be removed from the list of outputs of the node it is connected to. Then every output of the node must be checked. The node must be removed from the list of inputs of each node that the outputs are connected to.

```
foreach (inp in node.inputlist)
    if (inp.source == null)
        continue;
    end
    inp.source.inputlist.remove(inp);
end
foreach (outp in node.outputlist)
    foreach (inp in outp.inputlist)
        inp.source = null;
    end
end
odelist.remove(node);
```

Node Select Algorithm

The user should be able to use the mouse to select nodes in the circuit. First convert the position on screen of the mouse to a position in world coordinates. Then loop over every node, checking if the cursor is inside it's bounding rectangle. Then use which mouse button is pressed to determine what action to take.

```
function mouseDown(screenX, screenY, button)
    var x = screenX / cameraScale + cameraX
```

```
var y = screenY / cameraScale + cameraY

for each node in circuit
    var boundingBox = node.boundingBox
    if x >= boundingBox.left and x <= boundingBox.right and
        y >= boundingBox.top and y <= boundingBox.bottom
        // The user has clicked on a node

        if button == LeftMouseButton
            beginDrag(node)
        end
        else if button == RightMouseButton
            deleteNode(node)
        end
    end
end
end
```

Grid Lines

The program will feature a grid of lines in the background to help visualise panning and zooming the camera. Below is the algorithm for drawing these lines.

```
procedure drawGridLines()
    // The distance between the lines in world units.
    const lineDistance = 50

    var numLinesX = ((ScreenWidth / lineDistance) / CameraScale + 1) as integer
    var numLinesY = ((ScreenHeight / lineDistance) / CameraScale + 1) as integer

    // Draw vertical lines
    for i from 0 to numLinesX do
        var x = i * lineDistance + CameraPosition.X - CameraPosition.X % lineDistance

        DrawBlackLine(x, CameraPosition.Y, x, ScreenHeight / CameraScale +
CameraPosition.Y)
    end

    // Draw horizontal lines
    for i from 0 to numLinesY do
        var y = i * lineDistance + CameraPosition.Y - CameraPosition.Y % lineDistance

        DrawBlackLine(CameraPosition.X, y, ScreenWidth / CameraScale + CameraPosition.X,
y)
    end
end
```

Camera Zooming

When the user scrolls the mouse wheel the camera will zoom in and out. To make this animation satisfying and intuitive, the zooming should be centred on the cursor.

```
procedure MouseWheelEvent(float scrollAmount)
    // Change the camera scale by adding a fraction of the current scale times the scroll amount.
    // This causes (approximate) exponential scaling (the rate of change is proportional to the
    current value).

    float scaleChange = CameraScale * scrollAmount / 2000;
    float newScale = CameraScale + scaleChange;

    // Calculate the coordinates of the cursor in world coordinates using the old scale.
    var oldWorldMouseX = MousePosition.X / CameraScale + CameraPosition.X;
    var oldWorldMouseY = MousePosition.Y / CameraScale + CameraPosition.Y;

    // Calculate the coordinates of the cursor in world coordinates using the new scale.
    var worldMouseX = MousePosition.X / newScale + CameraPosition.X;
    var worldMouseY = MousePosition.Y / newScale + CameraPosition.Y;

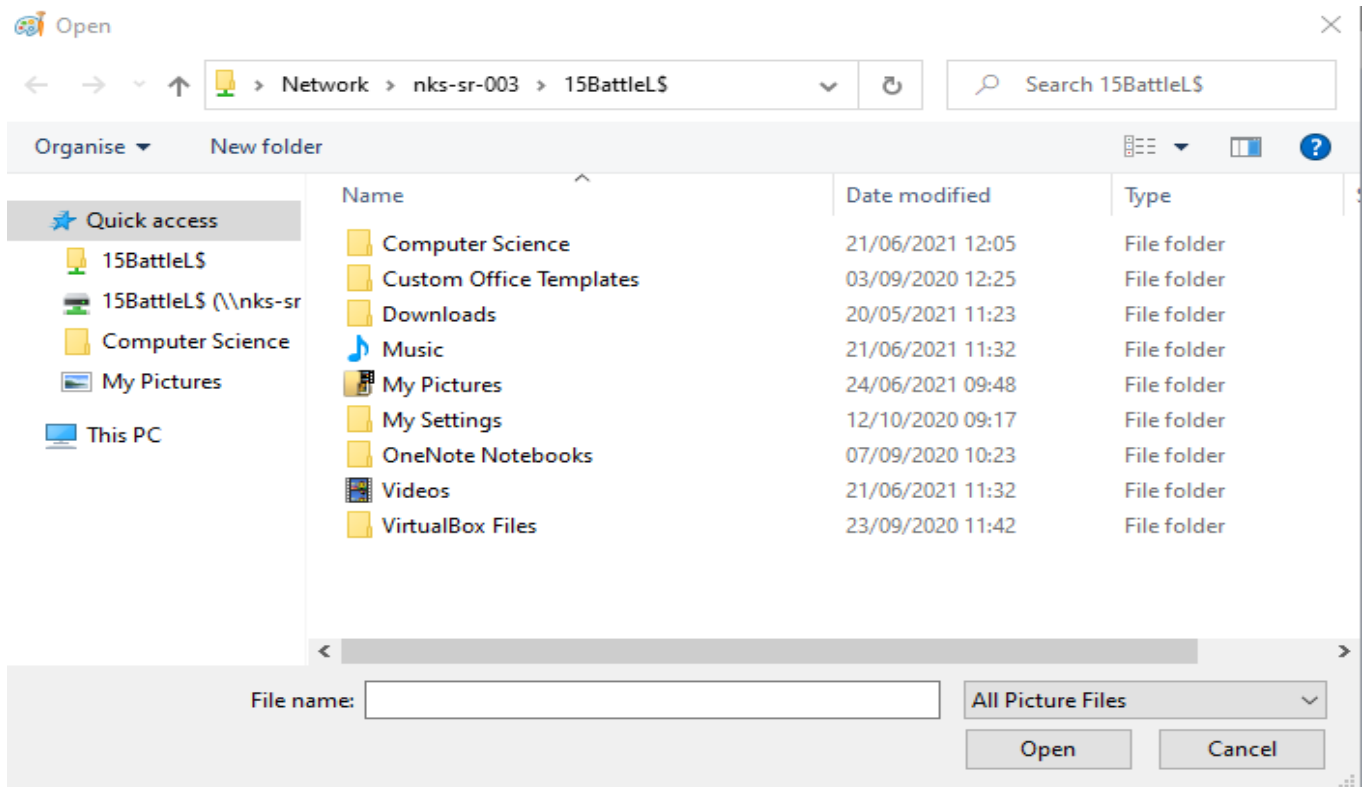
    // By just zooming without adjusting position the camera will appear to move relative to the
    cursor.
    // Subtract the difference between new and old positions to eliminate this movement and
    make the
    // camera zoom in and out centred on the cursor.
    CameraPosition.X -= worldMouseX - oldWorldMouseX;
    CameraPosition.Y -= worldMouseY - oldWorldMouseY;

    CameraScale = newScale;

    Invalidate();
end
```

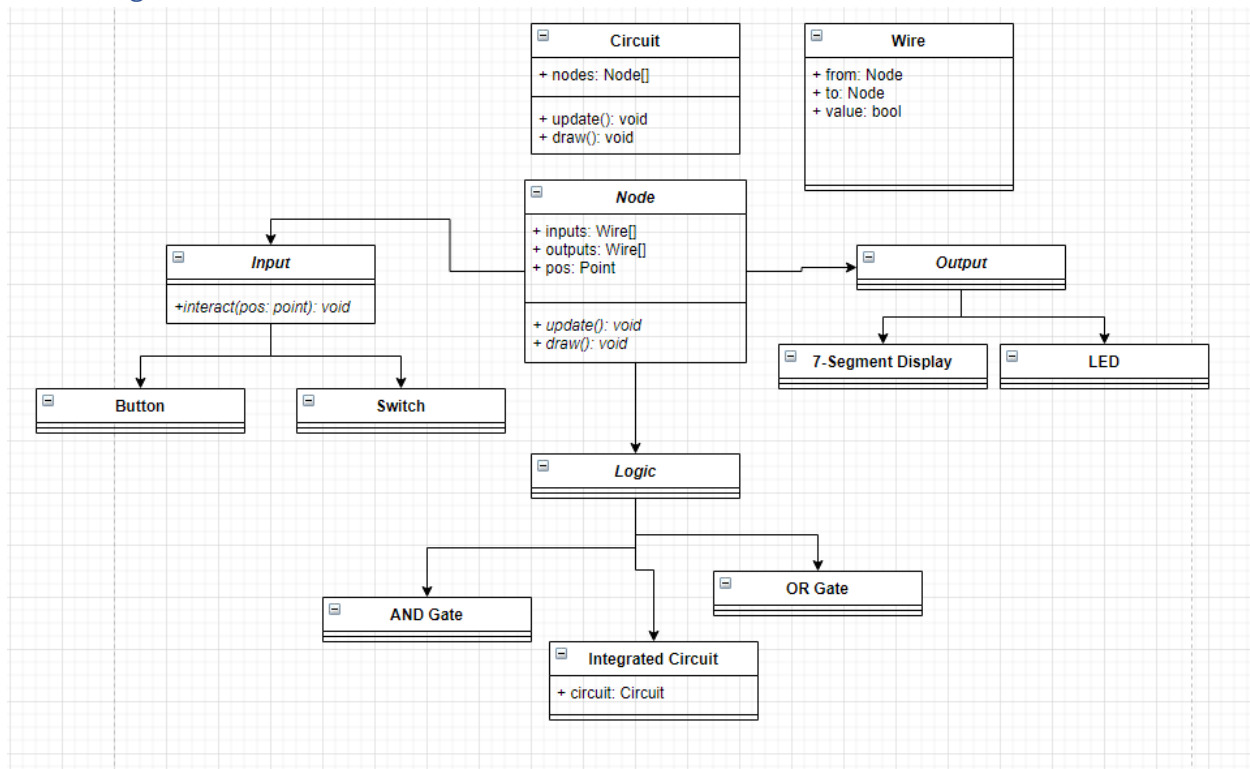
Open Circuit From File

The program will include an option to load a circuit from a file by using the operating system file open dialogue box. The circuit will then open the circuit in the circuit editor screen. The user can use this feature to do more work on previously created circuits, even if they are created by other people and sent to their computer.



In the program I will use Json.NET by Newtonsoft for saving and loading. This is a library that can write any C# object to a file as in JSON format.

Class Diagram



Circuit	
Wire	A wire represents the connection between two nodes. From is the node that outputs the Value. To is the node that uses Value as an input.
Node	Node is an abstract base class for logic gates, inputs, and outputs. When a node is updated, it reads inputs from the input array, and writes results of the evaluation to the outputs array.
Input	Nodes that inherit from Input can be interacted with by the user in the circuit editor using the mouse and keyboard.
Integrated Circuit	The user can encapsulate common arrangements of circuits in an integrated circuit and reuse them. When the user is selecting a group of nodes to become an integrated circuit (IC), any input nodes (e.g. buttons) will become input connections on the new IC, and any output nodes in the selection (e.g. LEDs) will become output connections.

Test Data

During the development of the program, I will use automated testing to ensure that the program functions correctly. When you make changes to the code with manual testing, you might only run new tests related to the code that you have just changed, not realizing that your changes broke another part of the program. Automated testing can *automatically* detect these side effects immediately.

My test data will consist of truth tables that the program must be able to correctly generate.

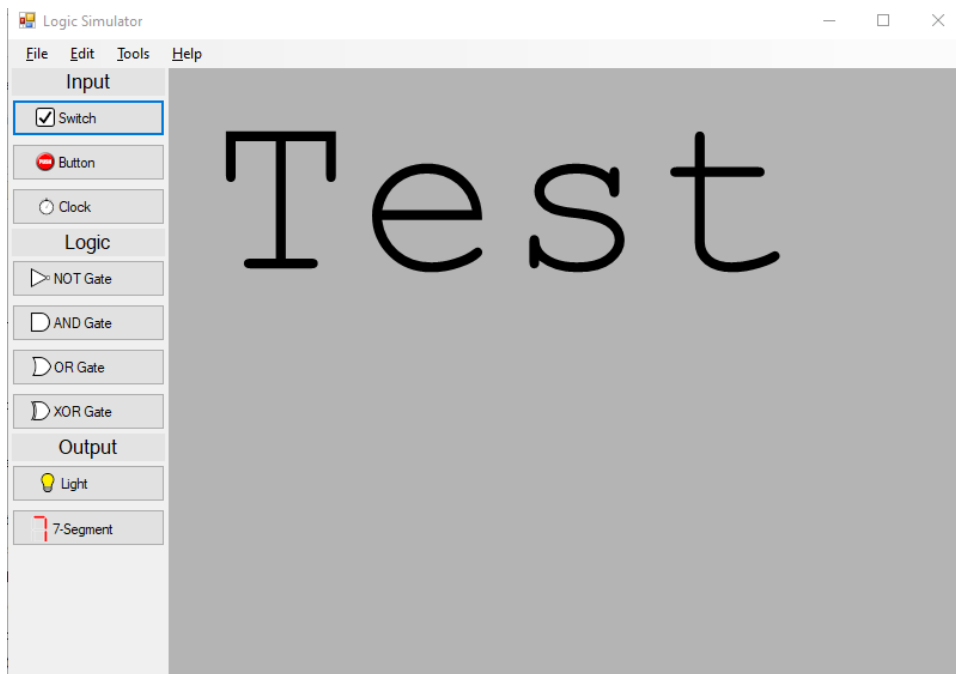
Circuit	Truth Table		
AND Gate	A	B	Output
	0	0	0
	0	1	0
	1	0	0
	1	1	1
OR Gate	A	B	Output
	0	0	0
	0	1	1
	1	0	1
	1	1	1
NOT Gate	A	Output	
	0	1	
	1	0	

XOR Gate	A	B	Output		
	0	0	0		
	0	1	1		
	1	0	1		
	1	1	0		
Full Adder	A	B	C _{in}	Sum	C _{out}
	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	0
	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	1	0	0	1
	1	1	1	1	1

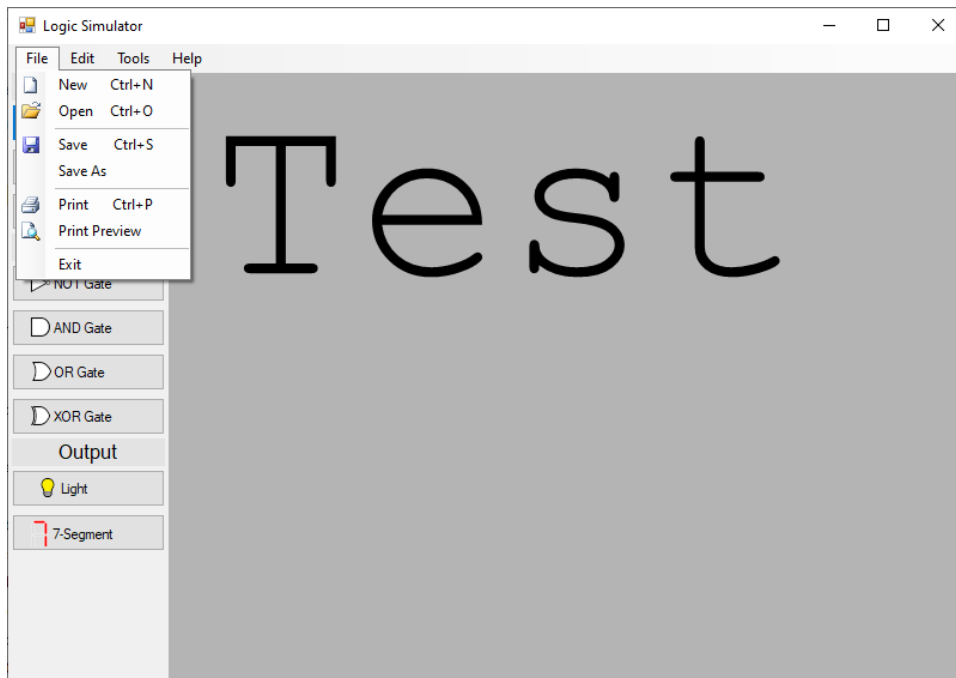
Development

Foundation

Before implementing the main functionality of the program, there must be a foundation for it to be contained in. The “node palette” on the left is created using the Windows Forms FlowLayoutPanel, which has built in scrolling functionality. This means that if the program is being used in a small window or more node types are added in the future the user will still be able to access them all by using the scrollbar.



The program will need a menu bar for buttons such as File/New, File/Save, Edit/Undo, and more. Windows Forms contains a MenuStrip class for this and has a button in the UI designer to automatically add in the standard buttons. The buttons do not do anything yet, as there are no files to save and no actions to undo, the functionality will be implemented later.

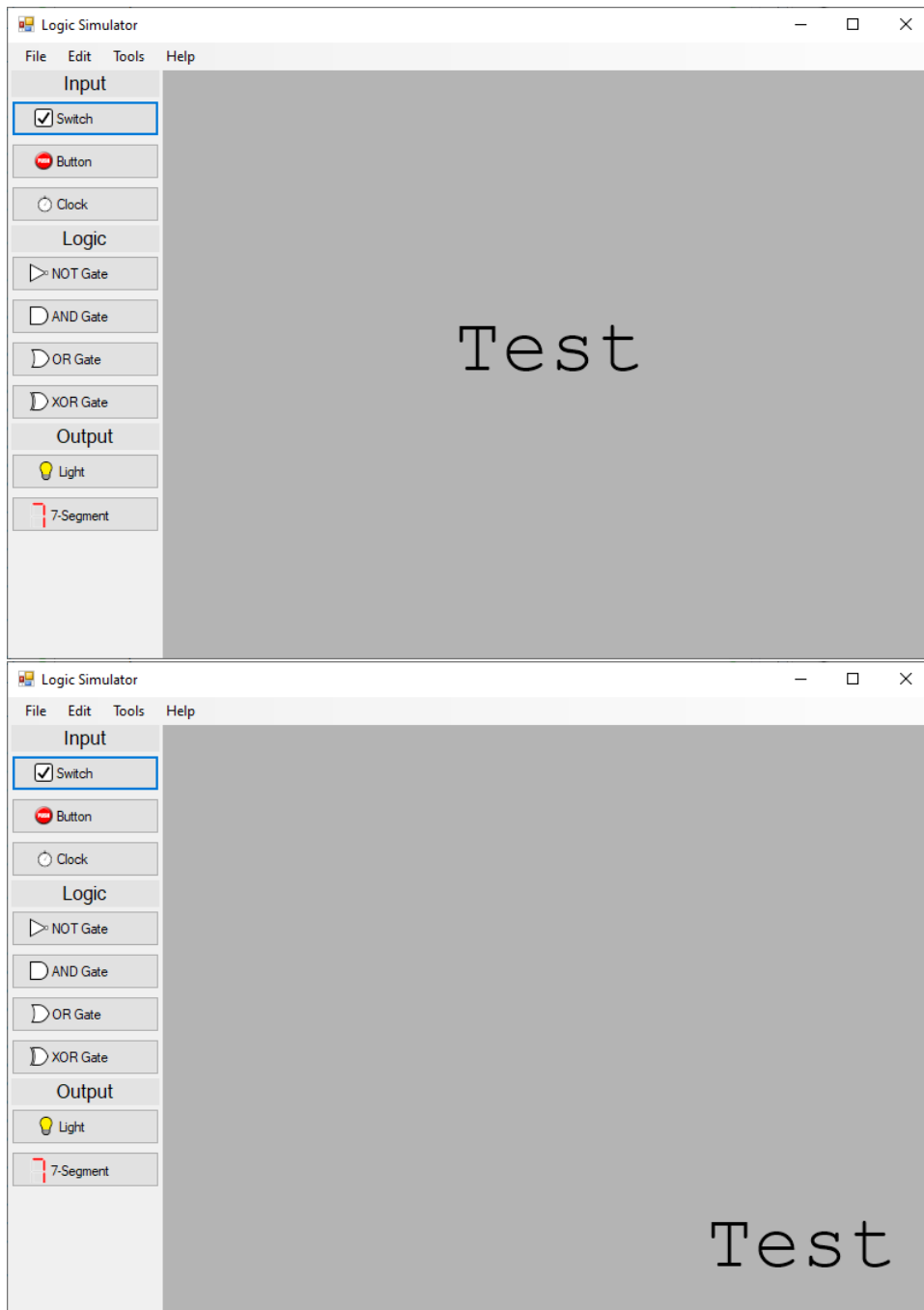


Camera

There are many interesting circuits the user may want to build that are too large to fit in a single window, so in order to not limit the user the program must support a camera or viewport that can pan and zoom. If the user clicks and drags the mouse on the empty background the camera should pan. When the user scrolls the mouse wheel the program should zoom in or out around the cursor. Many other programs use these same controls for the viewport, so it will be easy for people to learn to use this program.

Once the camera position and scale are calculated, the `Graphics.TranslateTransform` and `Graphics.ScaleTransform` functions can be used to transform all graphics operations. The Graphics transform functions do not apply to Windows Forms Controls as they might in a more advanced GUI framework, however that is okay because advanced controls won't be needed inside the viewport.

Below are two screenshots of the program, showing how the user can pan the camera.



```
bool mouseDown = false;
PointF lastMousePos = new PointF(0, 0);

1 reference
private void CircuitViewControl_MouseMove(object sender, MouseEventArgs e)
{
    if (mouseDown)
    {
        camPos.X -= (e.X - lastMousePos.X) / scale;
        camPos.Y -= (e.Y - lastMousePos.Y) / scale;

        Invalidate();
    }

    lastMousePos = e.Location;
}
```

When the mouse is down and moving, subtract the mouse movement from the camera position. For example, when the user clicks and drags left, the camera moves to the right.

```
PointF camPos = new PointF(0, 0);
float scale = 5;

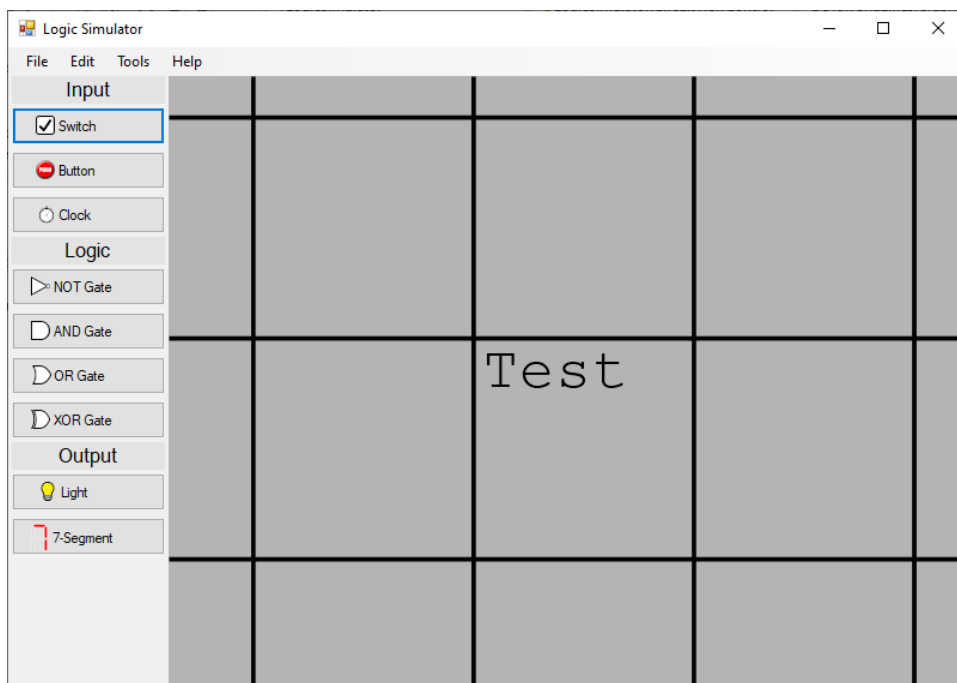
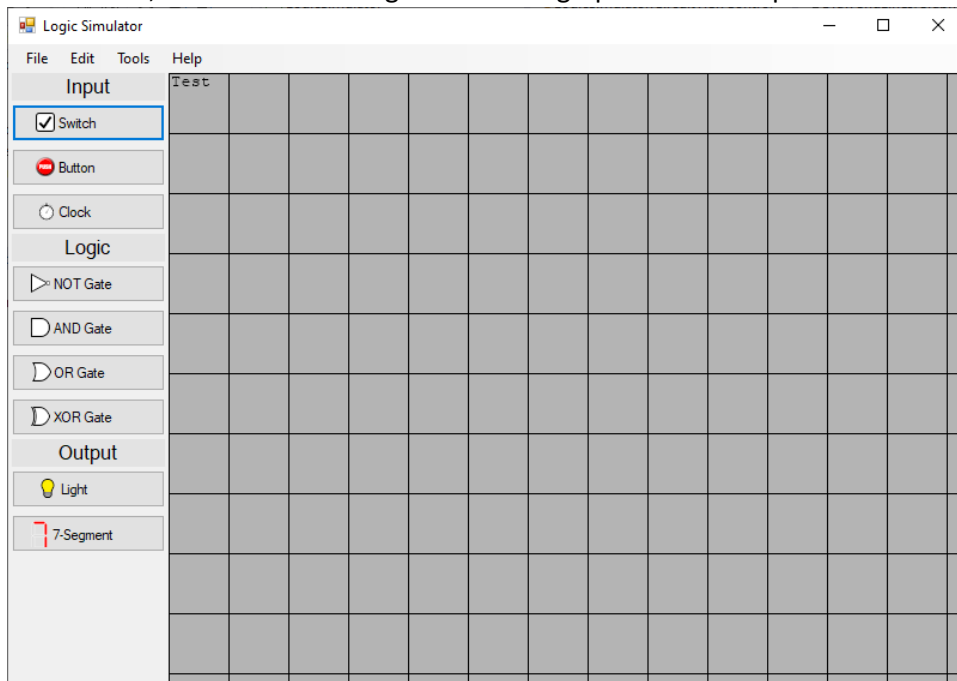
0 references
protected override void OnPaint(PaintEventArgs pe)
{
    base.OnPaint(pe);

    var g = pe.Graphics;
    g.SmoothingMode = SmoothingMode.AntiAlias;
    g.TextRenderingHint = TextRenderingHint.AntiAlias;

    g.ScaleTransform(scale, scale);
    g.TranslateTransform(-camPos.X, -camPos.Y);

    g.DrawString("Test", font, Brushes.Black, 0, 0);
}
```

In OnPaint, I enable antialiasing to make the graphics look less pixelated when moving the camera.



```
private void CircuitViewControl_MouseWheel(object sender, MouseEventArgs e)
{
    float scaleChange = scale * e.Delta / 2000f;
    float newScale = scale + scaleChange;

    var oldWorldMouseX = e.X / scale + camPos.X;
    var oldWorldMouseY = e.Y / scale + camPos.Y;

    var worldMouseX = e.X / newScale + camPos.X;
    var worldMouseY = e.Y / newScale + camPos.Y;

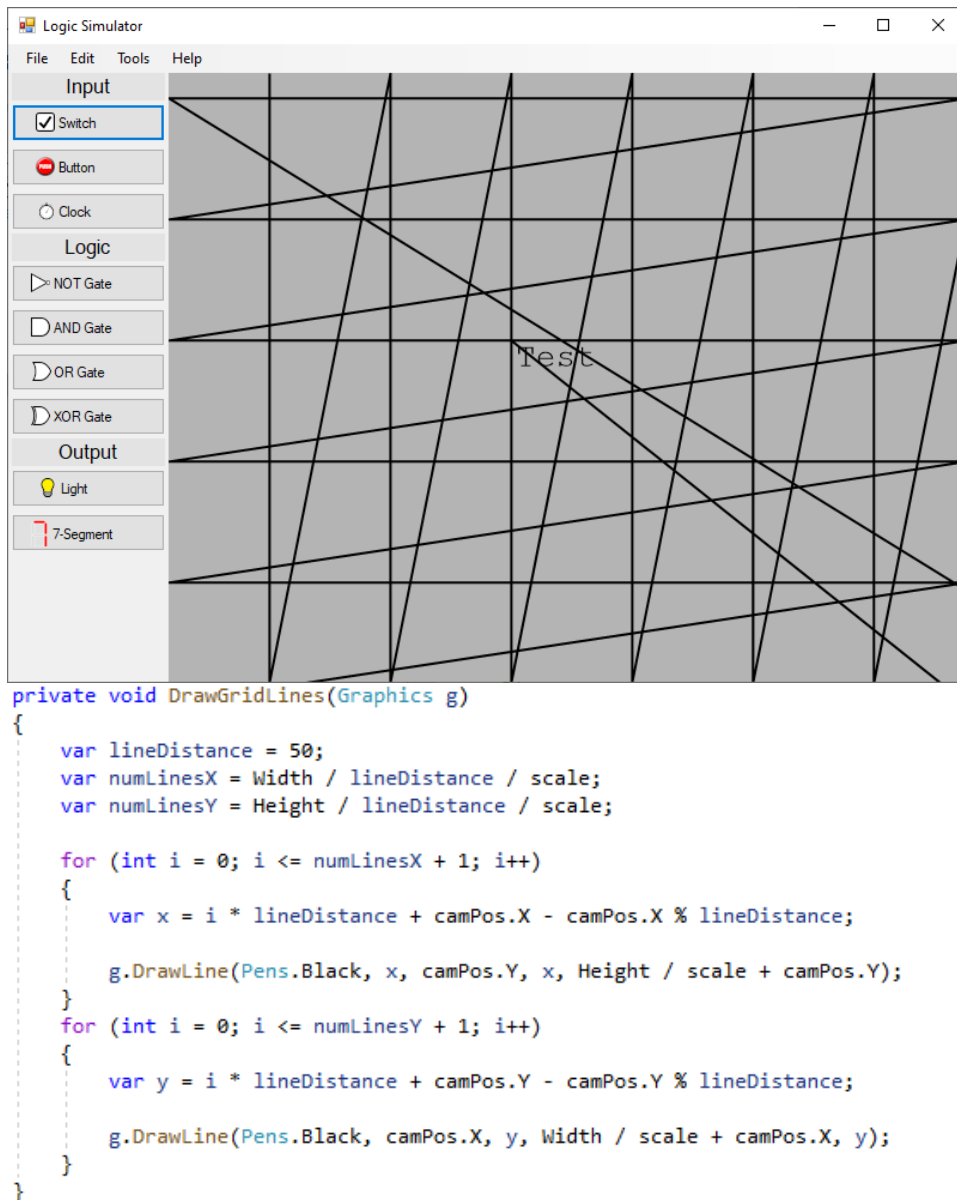
    camPos.X -= worldMouseX - oldWorldMouseX;
    camPos.Y -= worldMouseY - oldWorldMouseY;

    scale = newScale;

    Invalidate();
}
```

oldWorldMouseX/Y are the coordinates the cursor point to before the scale, transformed by the camera. worldMouseXY are the coordinates the cursor point to after the scale. I want the camera to zoom in around the cursor, so these coordinates should be the same. To correct this, translate the camera by the difference between these coordinates.

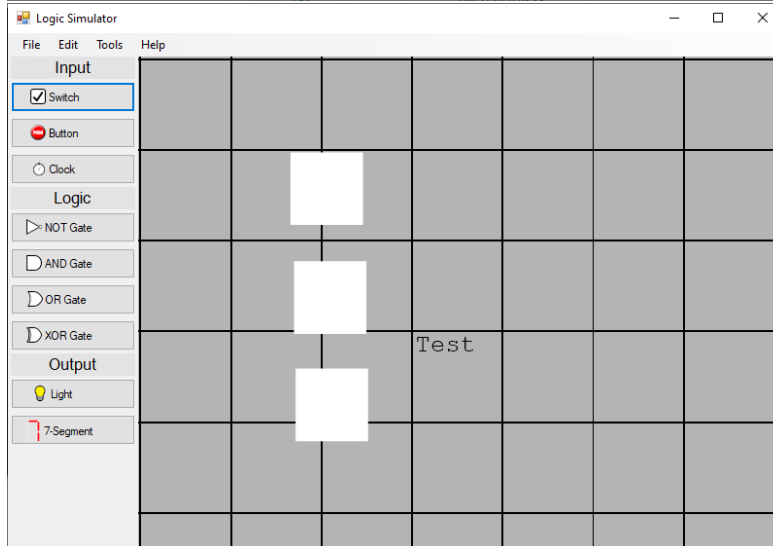
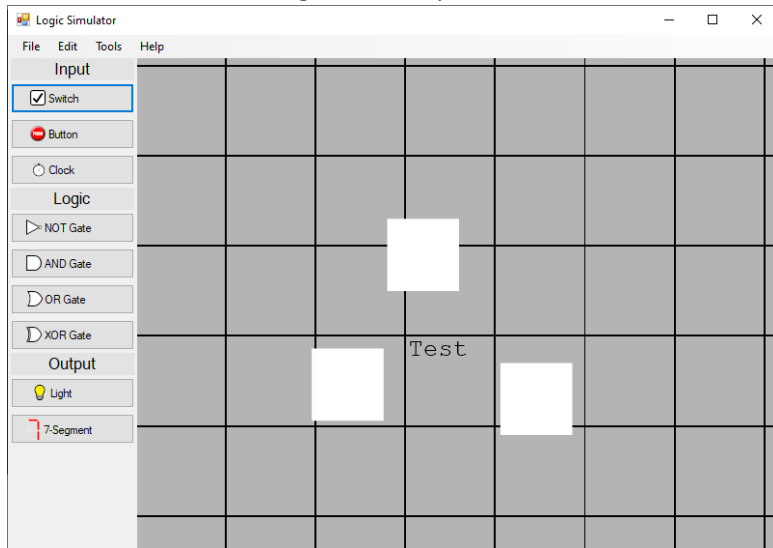
Windows Forms does not automatically redraw the screen at 60fps like a game would, for performance. A program must call `Invalidate` to tell Windows Forms to redraw the screen.



I implemented a line grid background to help visualise the camera transformations. When the camera is zoomed out far, a lot of lines are drawn, which leads to low performance. I tried to optimise this by storing the line coordinates in an array and using `g.DrawLine` to draw all of the lines at once. This did not make it any faster, and it also caused the interesting visual glitch shown above, where all lines are connected. Later in development I might try other methods to optimise the grid drawing, such as providing an option to turn the lines off or change the `lineDistance` depending on zoom level. However, that is not currently a high priority because it is rare that a user would zoom that far out.

Drag and Drop

If the user clicks and drags on a component of the circuit it should move around, following the cursor.



```
1 reference
private void CircuitViewControl_MouseDown(object sender, MouseEventArgs e)
{
    draggingNode = false;
    draggedNode = null;

    PointF worldCursor = ScreenToWorld(e.Location);

    foreach (var node in circuit.Nodes)
    {
        if (node.Rect.Contains(worldCursor))
        {
            draggingNode = true;
            draggedNode = node;
            return;
        }
    }

    panning = true;
}
```

I have changed the MouseDown function to check if the cursor is over any nodes. If it is, the user will start dragging a node, otherwise pan the background. It is not very efficient to do a linear search over every node in the circuit, it would be more efficient to use a space partitioning data structure such as a quadtree but that is outside the scope of this project. It is also not likely that there will be enough nodes that linear search will cause noticeable performance issues.

```
1 reference
public PointF ScreenToWorld(PointF screen)
{
    return new PointF(
        screen.X / scale + camPos.X,
        screen.Y / scale + camPos.Y);
}

0 references
public PointF WorldToScreen(PointF world)
{
    return new PointF(
        (world.X - camPos.X) * scale,
        (world.Y - camPos.Y) * scale);
}
```

The ScreenToWorld and WorldToScreen functions (seen in previous code snippet) are used for converting between coordinate systems. ScreenToWorld converts a point relative to the top-left of the screen ("screen coordinates") into a point relative to the origin ("world coordinates"). It is used to calculate which node the cursor is over. WorldToScreen converts from world coordinates to screen coordinates, which is used to calculate where on the screen a node should be drawn.

```
bool panning = false;
PointF lastMousePos = new PointF(0, 0);

bool draggingNode = false;
Node draggedNode = null;

1 reference
private void CircuitViewControl_MouseMove(object sender, MouseEventArgs e)
{
    float dx = (e.X - lastMousePos.X) / scale;
    float dy = (e.Y - lastMousePos.Y) / scale;

    if (draggingNode)
    {
        draggedNode.Rect.X += dx;
        draggedNode.Rect.Y += dy;

        Invalidate();
    }
    else if (panning)
    {
        camPos.X -= dx;
        camPos.Y -= dy;

        Invalidate();
    }

    lastMousePos = e.Location;
}
```

Now when the mouse moves, the program has to check if the user is dragging a node or the background. When the camera is being dragged it should look like every node is stationary relative to the cursor (the camera moves the opposite direction), which is why the dx and dy variables are subtracted.

Node

In this section I will begin work on the nodes.

```

8 references
public abstract class Node
{
    public RectangleF Rect;

    public WireConnector[] Inputs;
    public WireConnector[] Outputs;

    1 reference
    public Node(RectangleF rect, WireConnector[] inputs, WireConnector[] outputs)
    {
        Rect = rect;
        Inputs = inputs;
        Outputs = outputs;
    }

    1 reference
    public abstract void Update();

    3 references
    public virtual void OnPaint(Graphics g)
    {
        if (Program.DebugDraw)
        {
            g.DrawRectangle(Pens.Red, Rect.X, Rect.Y, Rect.Width, Rect.Height);
        }

        foreach (var input in Inputs)
        {
            DrawWireConnector(g, input);
        }

        foreach (var output in Outputs)
        {
            DrawWireConnector(g, output);
        }
    }

    2 references
    private void DrawWireConnector(Graphics g, WireConnector input)
    {
        float x = Rect.X + input.Pos.X - WireConnector.Radius;
        float y = Rect.Y + input.Pos.Y - WireConnector.Radius;

        Brush brush =
            input.Type == WireConnectorType.Input ? WireConnector.InputColour :
            WireConnector.OutputColour;

        g.FillEllipse(
            brush,
            x,
            y,
            WireConnector.Radius * 2,
            WireConnector.Radius * 2);
    }
}

```

This is the abstract Node class as shown in the class design diagram. All other nodes will inherit from this class, which will handle common tasks such as drag and drop and wire connections.

```
1 reference
public class Wire
{
    public Node From;
    public Node To;

    public bool Value;
}

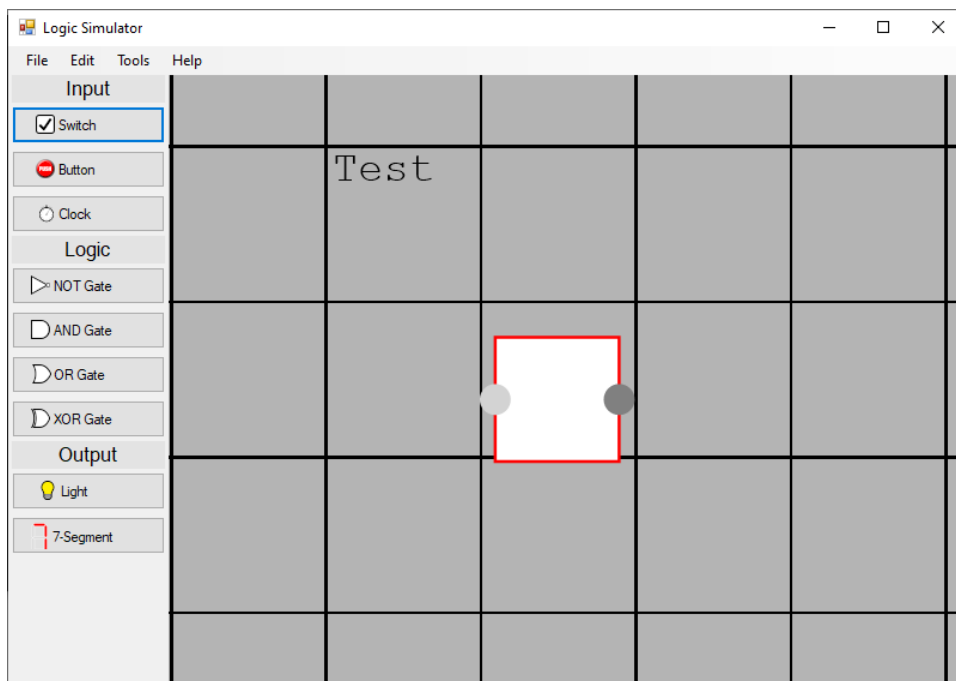
16 references
public class WireConnector
{
    public Wire Wire;
    public PointF Pos;
    public WireConnectorType Type;

    2 references
    public WireConnector(PointF pos, WireConnectorType type)
    {
        Pos = pos;
        Type = type;
    }

    public const float Radius = 5;
    public static Brush InputColour = Brushes.LightGray;
    public static Brush OutputColour = Brushes.Gray;
}

5 references
public enum WireConnectorType
{
    Input,
    Output
}
```

In this part of the code I couldn't follow the class diagram strictly. I realised that there needed to be a way to specify where on a node the connector is drawn so I introduced a new class "WireConnector". Every Node has two arrays of WireConnector, Inputs and Outputs.



Wires

In this section, I implement connecting nodes with wires.

```
public class Input
{
    public Node Node;
    public PointF Pos;

    // Where this input will read a value from
    public Output Source;

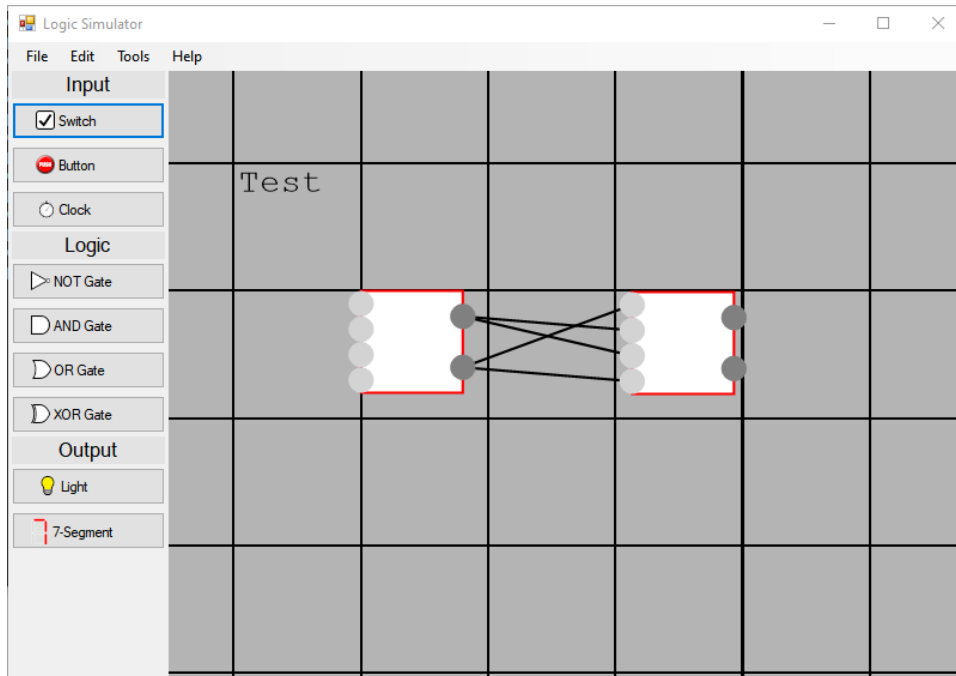
    4 references
    public Input(Node node, PointF pos)
    {
        Node = node;
        Pos = pos;
    }
}

8 references
public class Output
{
    public Node Node;
    public PointF Pos;

    public bool Value;
    public List<Input> Inputs = new List<Input>();

    2 references
    public Output(Node node, PointF pos)
    {
        Node = node;
        Pos = pos;
    }
}
```

A Node has a list of inputs and a list of outputs. One output can send a value to many inputs, and each input reads from one output. In the editor, the user can use the mouse to drag and drop connections from outputs to inputs.



```
foreach (var node in circuit.Nodes)
{
    foreach (var o in node.Outputs)
    {
        var posx = node.Rect.X + o.Pos.X;
        var posy = node.Rect.Y + o.Pos.Y;
        var r = 5;

        var rect = RectangleF.FromLTRB(posx - r, posy - r, posx + r, posy + r);
        if (rect.Contains(worldCursor))
        {
            Console.WriteLine("wire");

            dragType = DragType.Wire;
            draggedOutput = o;

            return;
        }
    }

    if (node.Rect.Contains(worldCursor))
    {
        dragType = DragType.Node;
        draggedNode = node;
        return;
    }
}
```

This code is added to the MouseDown function to check if the cursor is over an output when starting a drag and drop operation.

```
private void DrawWireDrag(Graphics g)
{
    if (dragType != DragType.Wire)
    {
        return;
    }

    g.DrawLine(Pens.Red, dragStart, ScreenToWorld(lastMousePos));
}
```

While the user is dragging a wire, a line is drawn from the start of the drag to the cursor. This visualization is included to make the interface more intuitive for the user.

```
private void CircuitViewControl_MouseUp(object sender, MouseEventArgs e)
{
    if (dragType == DragType.Wire)
    {
        PointF worldCursor = ScreenToWorld(e.Location);

        foreach (var node in circuit.Nodes)
        {
            foreach (var i in node.Inputs)
            {
                var posx = node.Rect.X + i.Pos.X;
                var posy = node.Rect.Y + i.Pos.Y;
                var r = 5;

                var rect = RectangleF.FromLTRB(posx - r, posy - r, posx + r, posy + r);
                if (rect.Contains(worldCursor))
                {
                    Console.WriteLine("drop");

                    if (!draggedOutput.Inputs.Contains(i))
                    {
                        if (i.Source != null)
                        {
                            i.Source.Inputs.Remove(i);
                        }

                        i.Source = draggedOutput;
                        draggedOutput.Inputs.Add(i);
                    }

                    goto loopend;
                }
            }
        }

        loopend:;

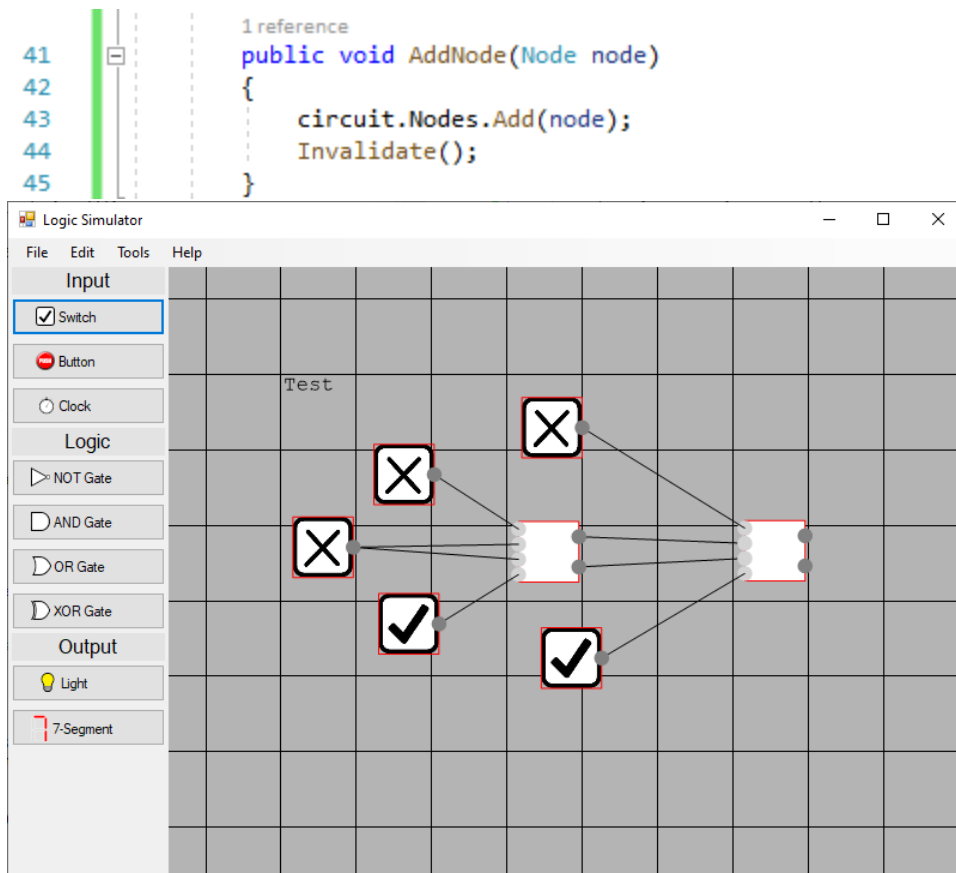
        dragType = DragType.None;
        draggedNode = null;
        draggedOutput = null;

        Invalidate();
    }
}
```

When the user releases the cursor, the program checks if it is over an Input connection. If it is not, the operation is cancelled. If the input is already connected to a different output, it gets overridden.

In order to build complex circuits, the user must be able to add components.

```
1 reference
21 private void sswitchButton_Click(object sender, EventArgs e)
22 {
23     circuitViewControl1.AddNode(new Switch(circuitViewControl1.ScreenCentre));
24 }
```



Exceptions

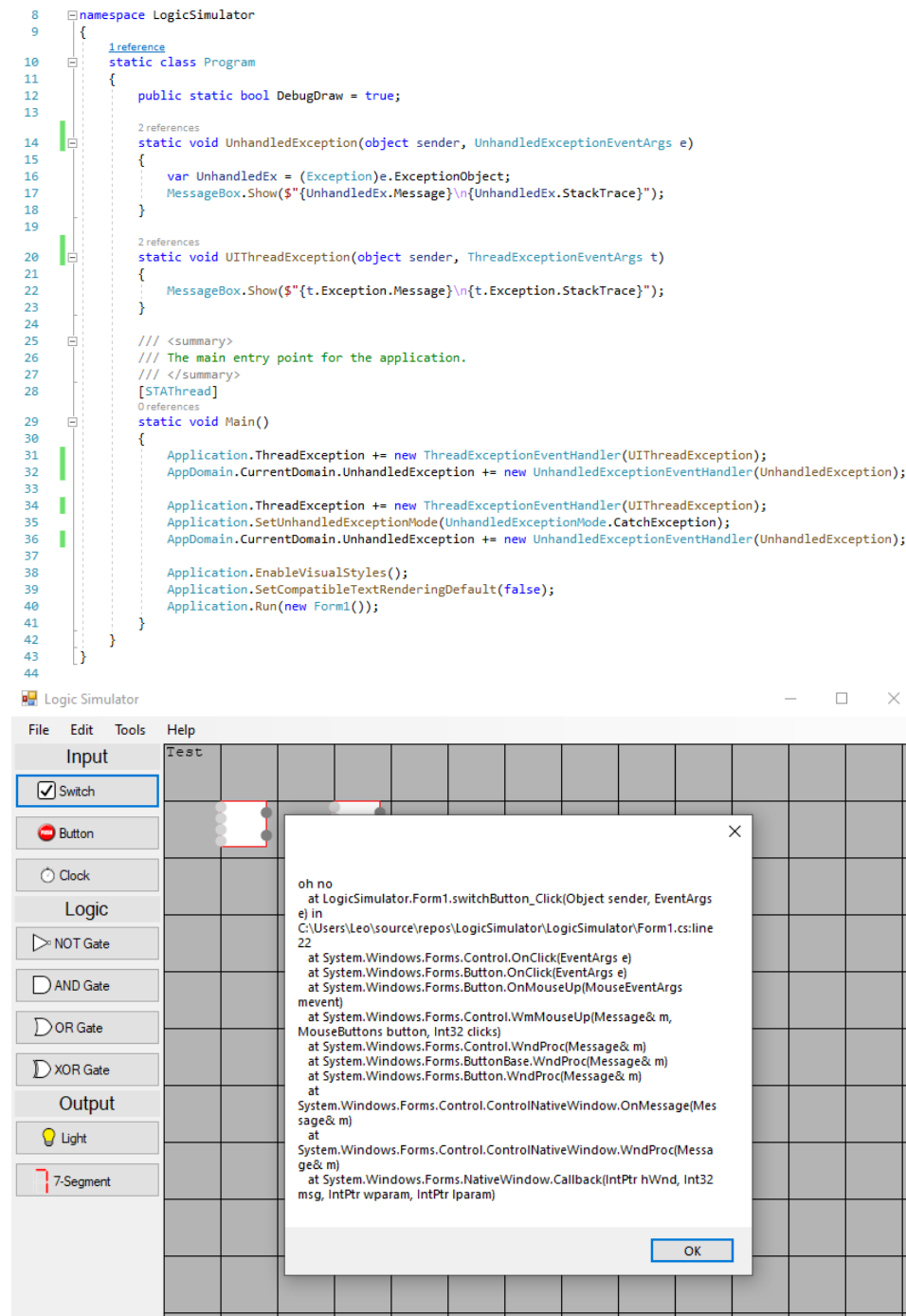
By default in Windows Forms if there is an exception the whole program will crash. In this section I added exception handling code so that if a user clicks a button that executes code with an uncaught exception they will have an opportunity to save their work.

```

11 namespace LogicSimulator
12 {
13     4 references
14     public partial class Form1 : Form
15     {
16         1 reference
17         public Form1()
18         {
19             InitializeComponent();
20         }
21
22         1 reference
23         private void switchButton_Click(object sender, EventArgs e)
24         {
25             throw new Exception("oh no");
26         }
27     }
28 }

```


Above is an example of some code that would throw an exception and cause the program to crash, however the code below causes the exception to be printed in a message box and the program to continue.



Circuit Evaluation

In this section I implement the main algorithm of the program.

Shown below is part 1 of the evaluation algorithm. It builds a list of the components that could be updated if “root” is updated.

```
public void EvaluateCircuit(Node root)
{
    var pes = new HashSet<Node>();
    var queue = new Queue<Node>();

    foreach (var output in root.Outputs) {
        foreach (var i in output.Inputs)
        {
            var node = i.Node;
            if (!queue.Contains(node))
            {
                queue.Enqueue(node);
            }
        }
    }
    while (queue.Count != 0)
    {
        var node = queue.Dequeue();
        pes.Add(node);
        foreach (var output in node.Outputs)
        {
            foreach (var i in output.Inputs)
            {
                var node2 = i.Node;
                if (!pes.Contains(node2))
                {
                    queue.Enqueue(node2);
                }
            }
        }
    }
}
```

Shown below is part 2 of the evaluation algorithm. For a full explanation, refer to the design section.

```
queue = new Queue<Node>();
queue.Enqueue(root);
var visited = new HashSet<Node>();
foreach (var output in root.Outputs)
{
    foreach (var i in output.Inputs)
    {
        var node = i.Node;
        if (!queue.Contains(node))
        {
            queue.Enqueue(node);
        }
    }
}
while (queue.Count != 0)
{
    var node = queue.Dequeue();
    var doUpdate = true;
    foreach (var input in node.Inputs)
    {
        var n = input.Source.Node;
        if (pes.Contains(n) && !visited.Contains(n))
        {
            doUpdate = false;
            break;
        }
    }
    if (!doUpdate)
    {
        break;
    }

    node.Update();
    visited.Add(node);

    foreach (var output in node.Outputs)
    {
        foreach (var i in output.Inputs)
        {
            var node2 = i.Node;
            if (!visited.Contains(node2))
            {
                queue.Enqueue(node2);
            }
        }
    }
}
```

In order to test the evaluation algorithm, there must be real components.

The LightNode class is used to show the output of the circuit. It is drawn on the screen as a light bulb that it turned on or off depending on the input.

```
public class LightNode : OutputNode
{
    static int nodeSize = 40;

    1 reference
    public LightNode(PointF pos) : base(new RectangleF(pos.X, pos.Y, nodeSize, nodeSize))
    {
        Inputs = new Input[]
        {
            new Input(this, new PointF(20, 40))
        };
    }

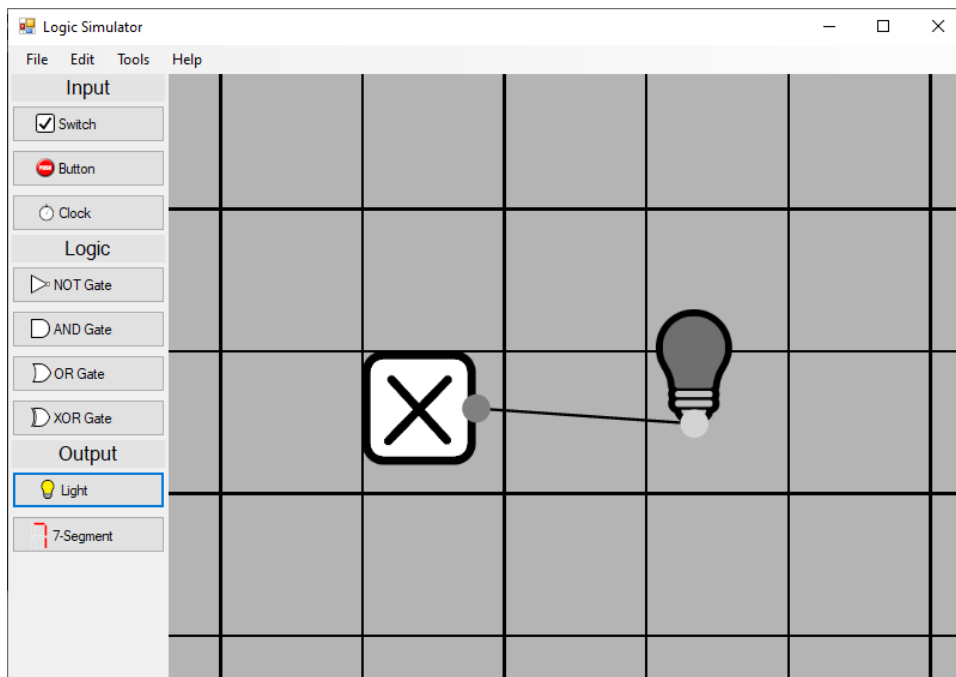
    bool isOn = false;

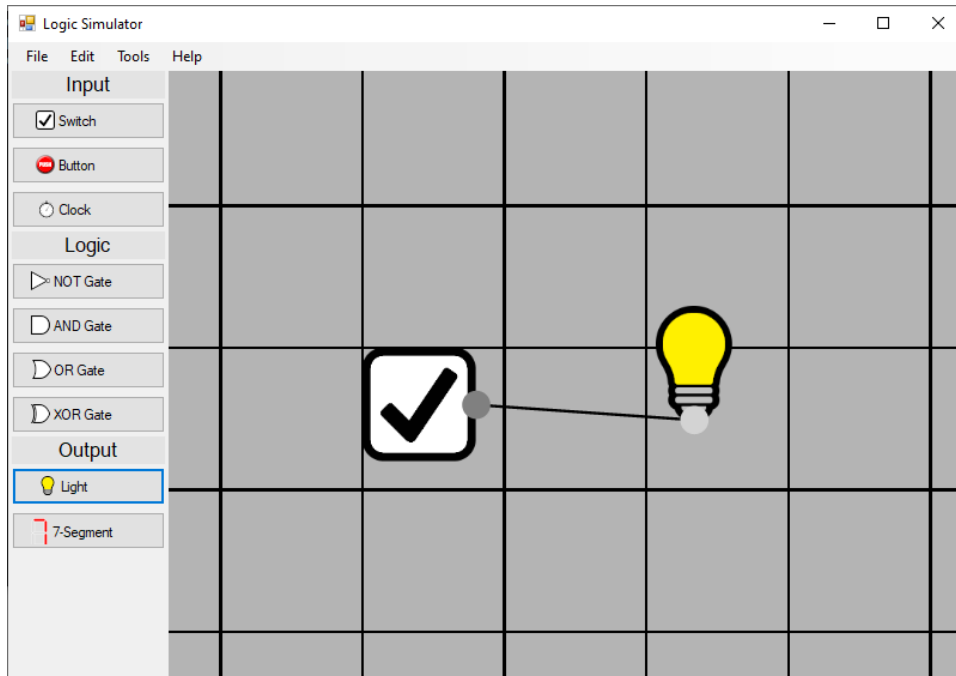
    5 references
    public override void Update()
    {
        isOn = Inputs[0].Source.Value;
    }

    static Bitmap offImg = Properties.Resources.light_off;
    static Bitmap onImg = Properties.Resources.light_on;

    9 references
    public override void OnPaint(Graphics g)
    {
        Bitmap img = offImg;
        if (isOn)
        {
            img = onImg;
        }
        g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

        base.OnPaint(g);
    }
}
```





The NotGate class is used to compute the boolean NOT function. The other gates use very similar code, so I will not include them all here.

```

public class NotGate : LogicNode
{
    static int nodeSize = 40;

    1 reference
    public NotGate(PointF pos) : base(new RectangleF(pos.X, pos.Y, nodeSize, nodeSize))
    {
        Inputs = new Input[]
        {
            new Input(this, new PointF(0, 20))
        };
        Outputs = new Output[]
        {
            new Output(this, new PointF(40, 20))
        };
    }

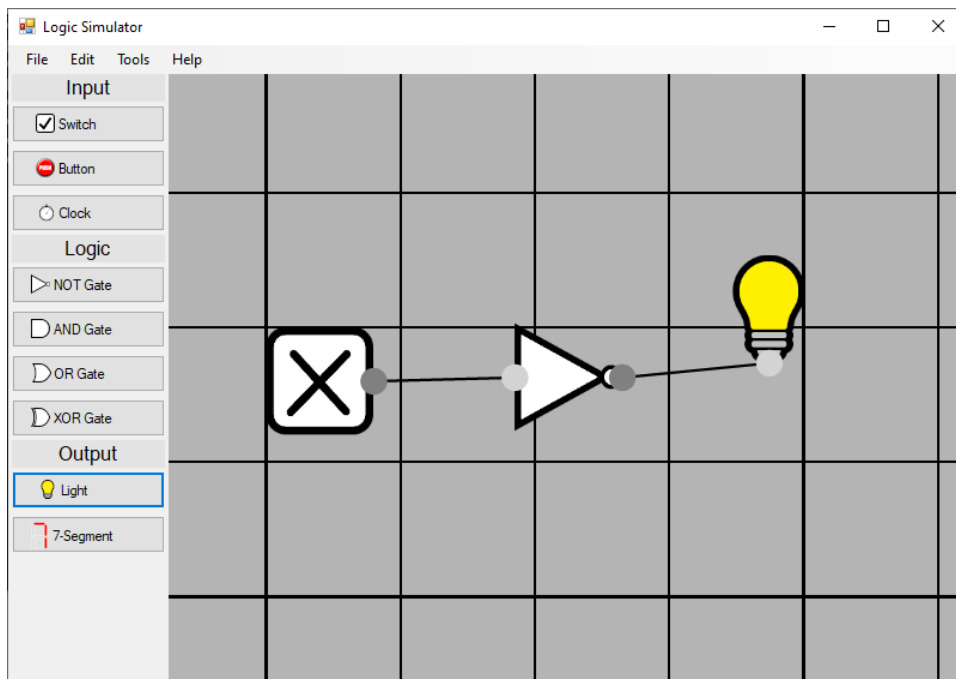
    5 references
    public override void Update()
    {
        Outputs[0].Value = !Inputs[0].Source.Value;
    }

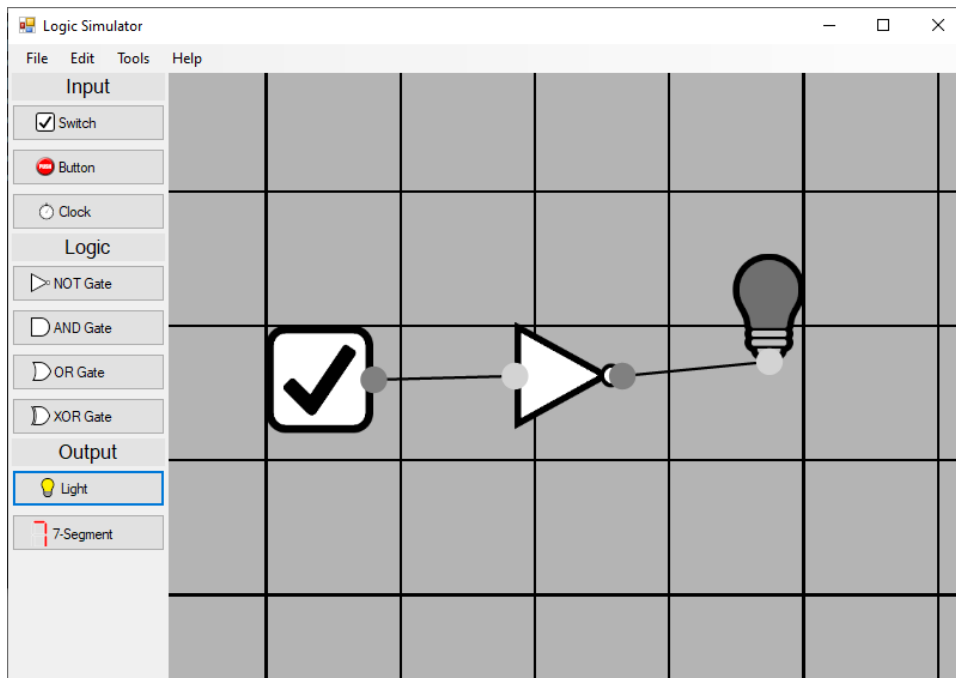
    static Bitmap img = Properties.Resources.not_gate;

    9 references
    public override void OnPaint(Graphics g)
    {
        g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

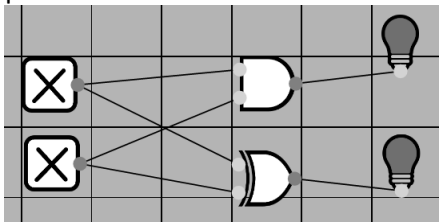
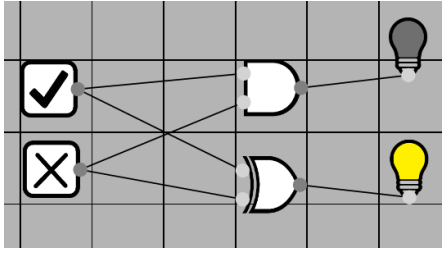
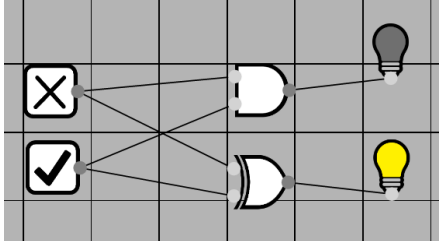
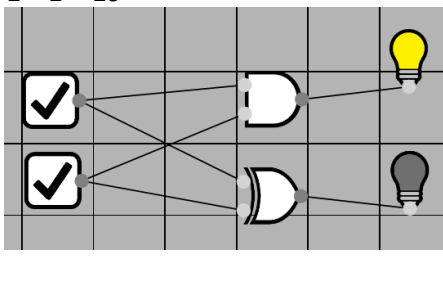
        base.OnPaint(g);
    }
}

```

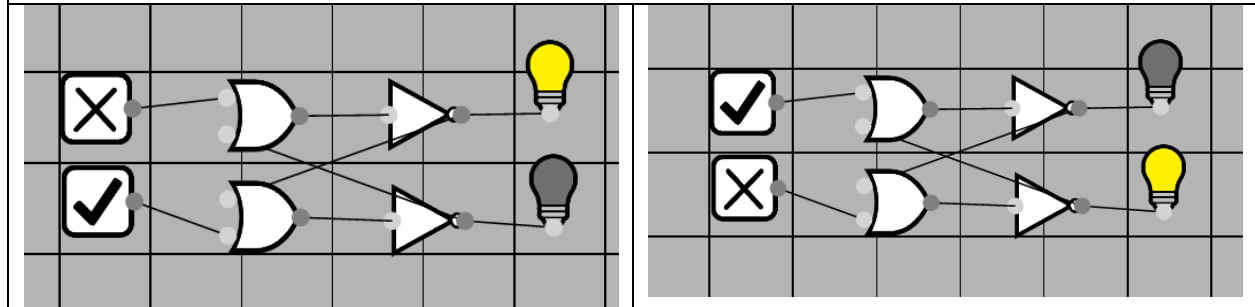




Evaluation Algorithm Testing

<p>Input: 00 Output: 00 $0 + 0 = 0$ pass</p> 	<p>Input: 10 Output: 01 $1 + 0 = 1$</p> 
<p>Input: 01 Output: 01 $0 + 1 = 1$ pass</p> 	<p>Input: 11 Output: 10 $1 + 1 = 10$</p> 
<p>This circuit is an SR-latch, a simple 1-bit memory cell. When both switches are turned off the circuit remembers the current value. When the top switch it turned on, the top light is turned off and the bottom light is turned on (even if the switch is turned off again). When the bottom switch is turned on, the top light is turned on and the bottom light is turned off.</p>	

While testing this circuit I realised that the evaluation algorithm is not able to handle circuits with cycles. This is because the algorithm waits to update a node until all of its inputs have been updated, but in a circuit with cycles this can lead to an infinite loop. To make this circuit work, I had to disable the check for if an input had been updated yet.



Saving/Loading

When building a large circuit the user must be able to save and load circuits on permanent storage to make sure that they don't lose their work between sessions. I initially planned to use the well-known Newtonsoft.JSON library to serialize the circuit to a JSON file, but encountered issues.

The first issue is that the circuit is a graph rather than a tree, which causes the serializer to get stuck in an infinite loop. This issue can be solved by providing a JsonSerializerSettings object with ReferenceLoopHandling = ReferenceLoopHandling.Serialize. TODO: Insert picture of fixed code.

The second issue is that Newtonsoft.JSON doesn't support deserializing polymorphic objects. This means that the serializer doesn't know if a JSON string was serialized from an AND gate or an OR gate (for example). This problem could be solved by writing custom JsonConvert classes for every class that needs to be serialized, but that would be tedious.

Instead, I decided to use the built-in BinaryFormatter, a class for writing objects to binary files. The only change necessary was adding the [Serializable] attribute to every class that needs to be serialized.

TODO: Insert picture of class with [Serializable] attribute, picture of BinaryFormatter serialization code.


```
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
    var d = new SaveFileDialog();
    if (d.ShowDialog() == DialogResult.OK)
    {
        using (var f = d.OpenFile())
        {
            var b = new BinaryFormatter();
            b.Serialize(f, circuitViewControll1.circuit);
        }
    }
}

private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    var d = new OpenFileDialog();
    if (d.ShowDialog() == DialogResult.OK)
    {
        using (var f = d.OpenFile())
        {
            var b = new BinaryFormatter();
            var circuit = (Circuit)b.Deserialize(f);
            circuitViewControll1.circuit = circuit;
            circuitViewControll1.Invalidate();
        }
    }
}
```

Something that should be noted is that BinaryFormatter is insecure, using it to deserialize an untrusted file could lead to arbitrary code execution. However, it is suitable for this project as it makes the saving and loading code simple and a hacker is unlikely to use an A-level project as an attack vector into a computer. Shown below is a program that demonstrates the insecurity of BinaryFormatter.

```
[Serializable]
5 references
class Safety
{
    1 reference
    public Safety()
    {
        Console.WriteLine("hooray");
    }
}

[Serializable]
3 references
class Danger
{
    public string Path;

    1 reference
    public Danger(string path)
    {
        Path = path;
    }

    0 references
    ~Danger()
    {
        Console.WriteLine($"Deleting file {Path}");
    }
}
```

```

0 references
static byte[] GoodClient()
{
    var s = new MemoryStream();
    var b = new BinaryFormatter();

    var o = new Safety();
    b.Serialize(s, o);

    return s.ToArray();
}

1 reference
static byte[] EvilHackerClient()
{
    var s = new MemoryStream();
    var b = new BinaryFormatter();

    var o = new Danger("C:\\Windows\\System32\\some_important.file");
    b.Serialize(s, o);

    return s.ToArray();
}

// The program expects message to contain a serialized Safety instance,
// but the user could be tricked into opening a file from a malicious source
// that contains a serialized instance of another class.
// The program tries to cast the Danger object to Safety, but fails with an InvalidCastException.
// However, the Danger object has already been created, and when the finalizer runs, it can delete files.
1 reference
static Safety UnsuspectingVictim(byte[] message)
{
    var s = new MemoryStream(message);
    var b = new BinaryFormatter();
    return (Safety)b.Deserialize(s);
}

static void Main(string[] args)
{
    //byte[] b = GoodClient();
    byte[] b = EvilHackerClient();

    try
    {
        Safety s = UnsuspectingVictim(b);
    }
    catch
    {
        GC.Collect();
        GC.WaitForPendingFinalizers();
    }
}

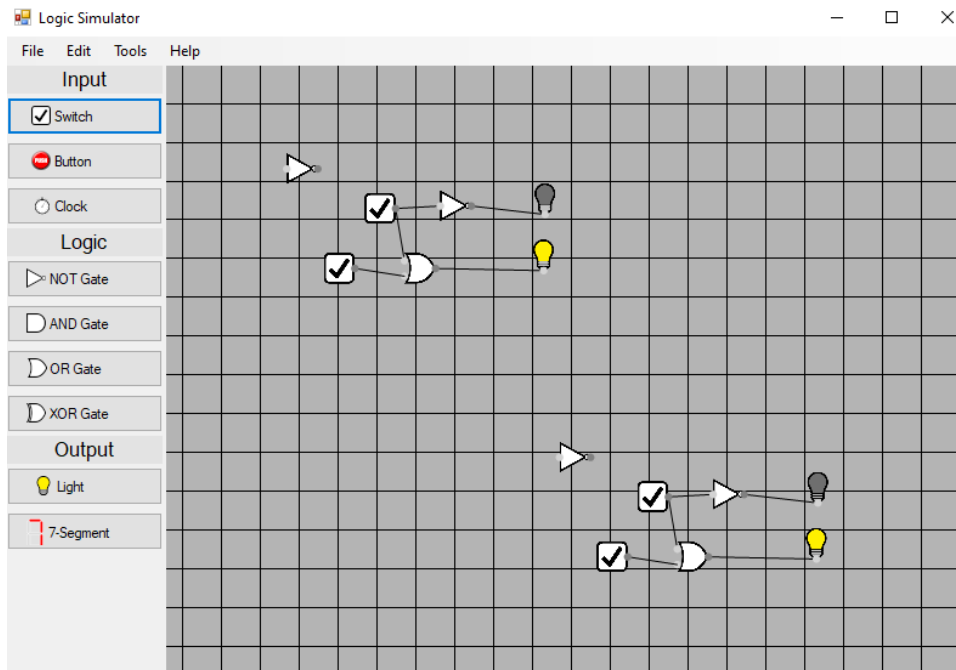
```

Import

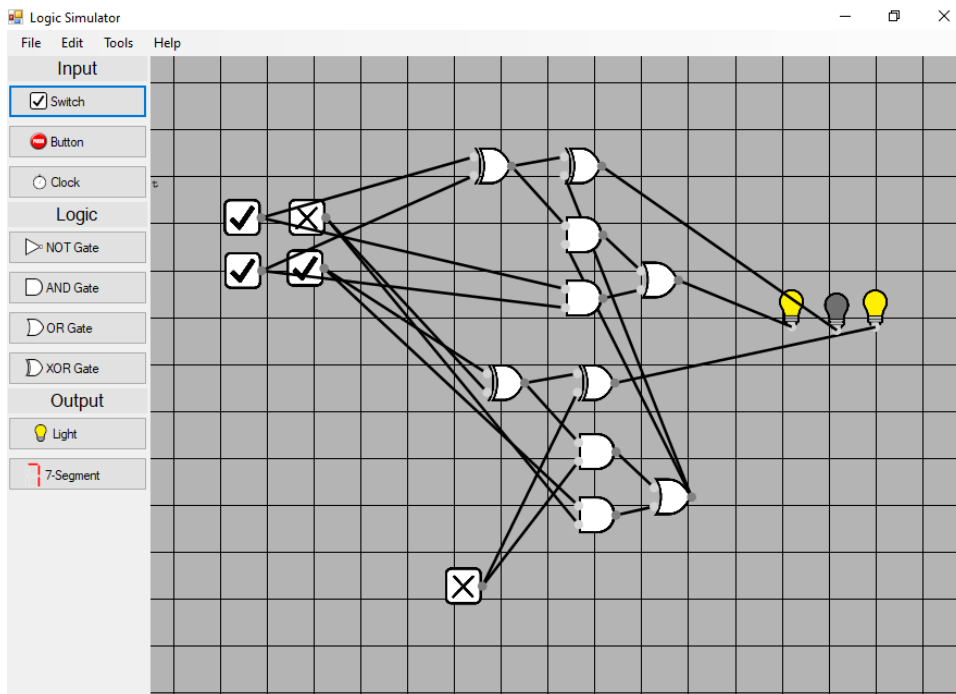
In the design section I listed ICs (reusable collections of components) as a project success requirement. After writing the saving and loading code, I have realised a better way to implement reuse of modular circuits: let the filesystem do the work. The way I will implement this feature is with an “import button”. Importing is similar to loading a file, but instead of replacing all components, the ones from the imported file are added to the current editor.

When the file is imported, all of the components are placed relative to the centre of the screen, which means that an offset has to be added to all of the coordinates.

Here is a screenshot showing a file containing a circuit that was copied using the import feature.



Here is an example of a circuit made using the import feature. I initially created a full adder and saved it to a file. I then imported that file and wired up the carry-out to create a 2-bit adder. The picture shows the circuit adding $2 + 3 = 5$. By repeatedly saving and importing the user could create an adder for binary numbers of any bit length.



However, this approach is not complete. The components are supposed to appear in the middle of the screen, but they do not. This is because the components in the imported file may not be centred at the

origin. To fix this, I will loop through the components and subtract the minimum x and y coordinate from each one.

```
private void importButton_Click(object sender, EventArgs e)
{
    // Import is like open, but the components are added to the current file instead of replacing them.

    var d = new OpenFileDialog();
    if (d.ShowDialog() == DialogResult.OK)
    {
        using (var f = d.OpenFile())
        {
            var b = new BinaryFormatter();
            var circuit = (Circuit)b.Deserialize(f);

            var minx = circuit.Nodes.Select(n => n.Rect.X).Min();
            var miny = circuit.Nodes.Select(n => n.Rect.Y).Min();

            // The components are added with their positions relative to the centre of the screen.
            var basePos = circuitViewControl1.ScreenCentre;
            basePos.X -= minx;
            basePos.Y -= miny;

            foreach (var n in circuit.Nodes)
            {
                n.Rect.X += basePos.X;
                n.Rect.Y += basePos.Y;
                circuitViewControl1.circuit.Nodes.Add(n);
            }

            circuitViewControl1.Invalidate();
        }
    }
}
```

Input and Output

The user has commented that the program would be easier to use if there were decimal input and output components.

The decimal output takes 4 binary inputs and displays a decimal number up to 15. Below is an excerpt from the decimal output code. The inputValue function returns the value of the i'th input as a number (or 0 if nothing is connected). In the Update function, the bits are added up and converted to a string for display.

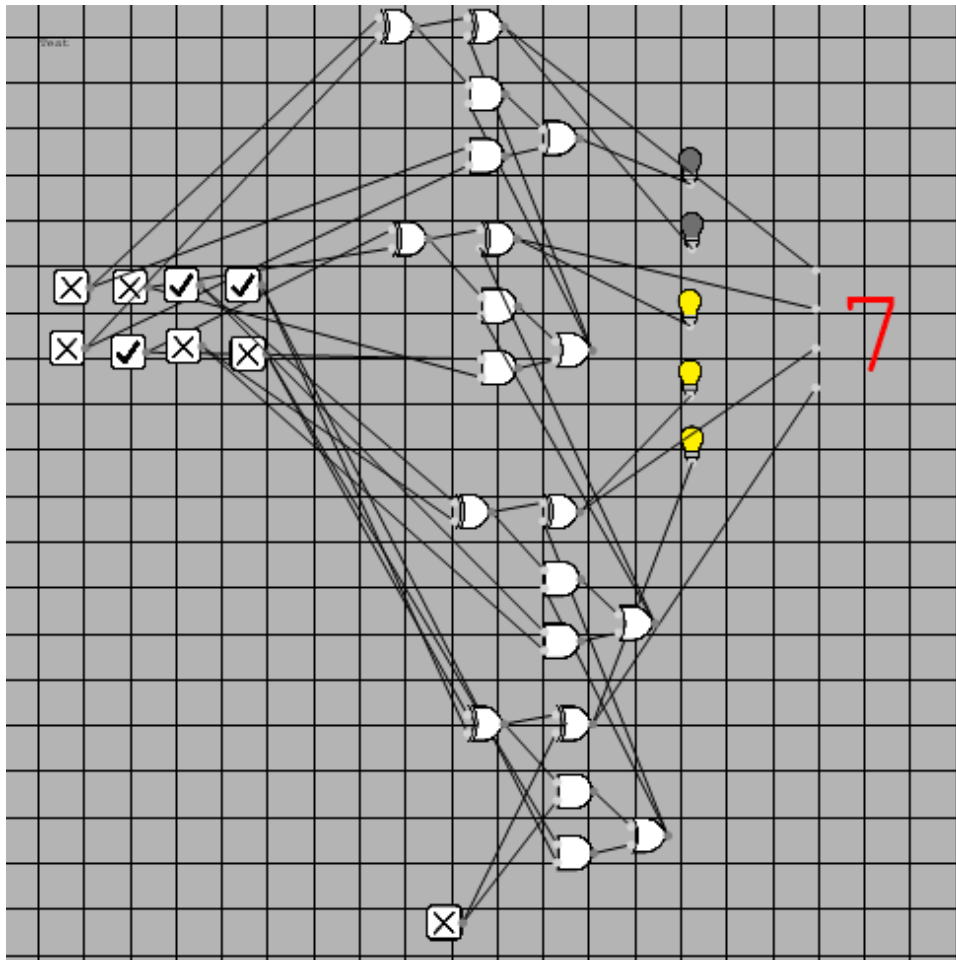
```
int inputValue(int i)
{
    return (Inputs[i].Source?.Value ?? false) ? 1 : 0;
}
```

```
// The number to display
int number = 0;
string numStr = "0";
```

9 references

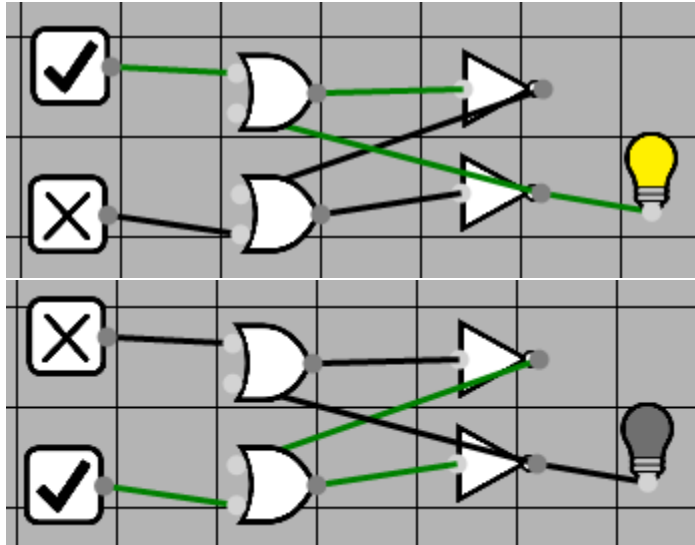
```
public override void Update()
{
    number =
        inputValue(0) * 8 +
        inputValue(1) * 4 +
        inputValue(2) * 2 +
        inputValue(3);

    numStr = number.ToString();
}
```



User Feedback

A user has given the feedback that it is very hard to understand what a circuit does, especially sequential circuits. They suggested that this is fixed by colouring the wires green when they output a true value. Here is an SR-latch circuit (the top switch sets the state to on, the bottom switch sets the state to off). The user said that the addition of the green lines makes it easier to understand why the circuit works.



Evaluation

User Evaluation Questionnaire

This is a questionnaire that users will fill in to evaluate the program.

- Rate the ease of using the UI of the program out of 10.
 - 9
- Is the camera panning and zooming useful?
 - Yes
- Do you think that the import feature adds value to the program?
 - Yes
- Does the program provide a sufficient range of components for building circuits?
 - Yes, but a push-button would be nice.
- Does the program present circuits in a way that makes them easy to understand. Rate out of 10.
 - 8. The wires should be able to curve.

Success Criteria Evaluation

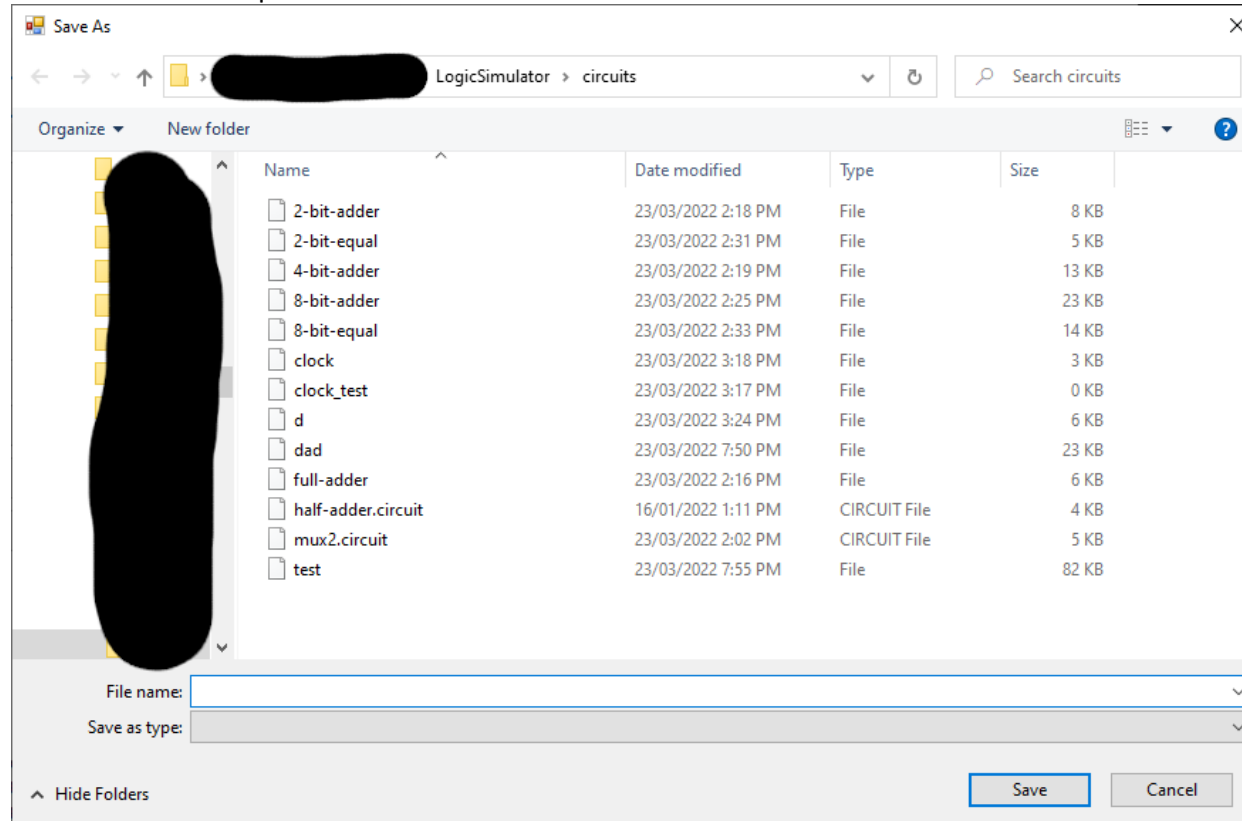
Here I shall check every point from the success criteria and evaluate how well they were implemented.

The user must be able to save and load circuits in files so that they can use them later.

Here is the Save dialog box, which is opened by typing Ctrl-S and is used to save circuits to the disc as a file. The dialog is open in a folder that I have used to store the circuits I have created while testing the program.

*Note: Loading is also implemented, but the dialog looks almost identical, so a screenshot is not included.

This criterion was implemented well. ✓



The program must be able to simulate circuits with deterministic results.

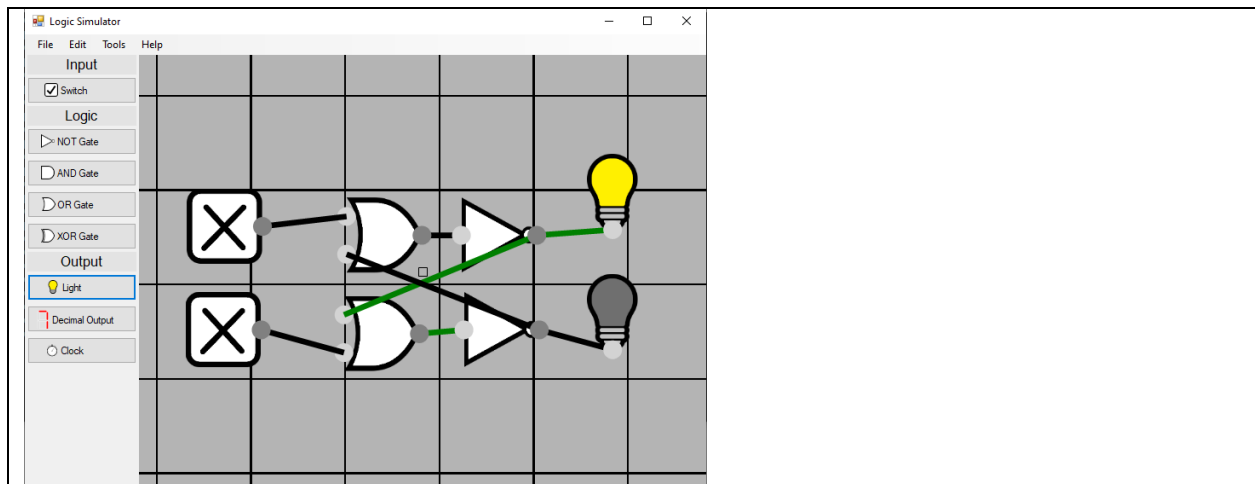
If the same circuit is saved and then loaded back up it will still work correctly.

✓

The program should support sequential (stateful) logic as well as the simpler combinational logic.

As demonstrated earlier in this document, this program is capable of simulating an SR-latch, a circuit used to store a single bit of memory.

✓



The program will include a clock component to periodically advance a stateful circuit.

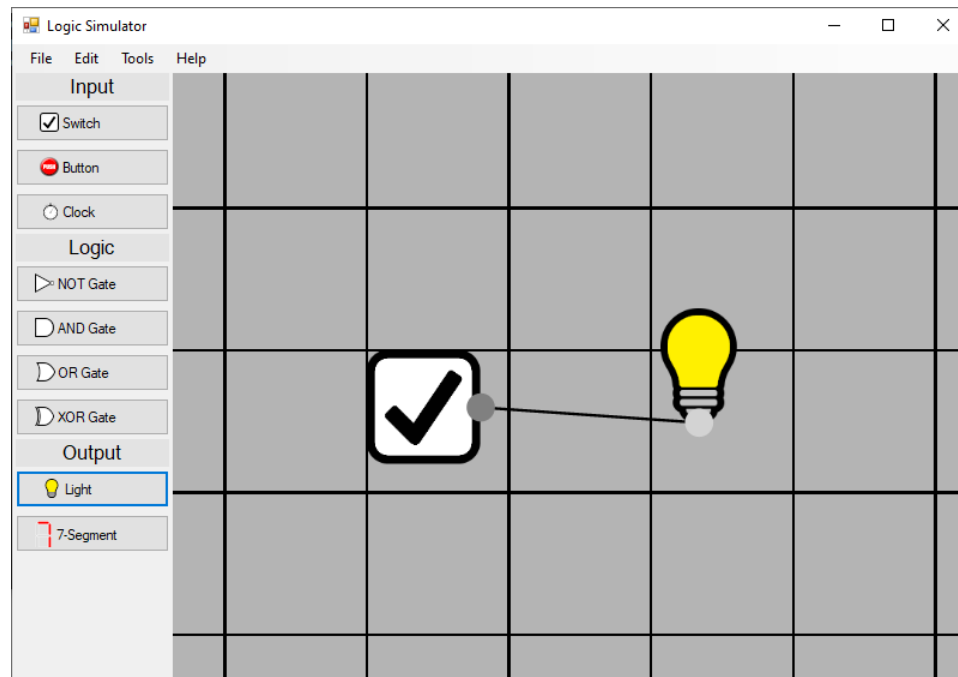
I did not implement this feature.

✗

The program will include input components such as buttons, to allow the user to interact with the circuit.

The program includes one type of input: the toggle switch. I did not implement the push-button as I did not think it adds much value to the program.

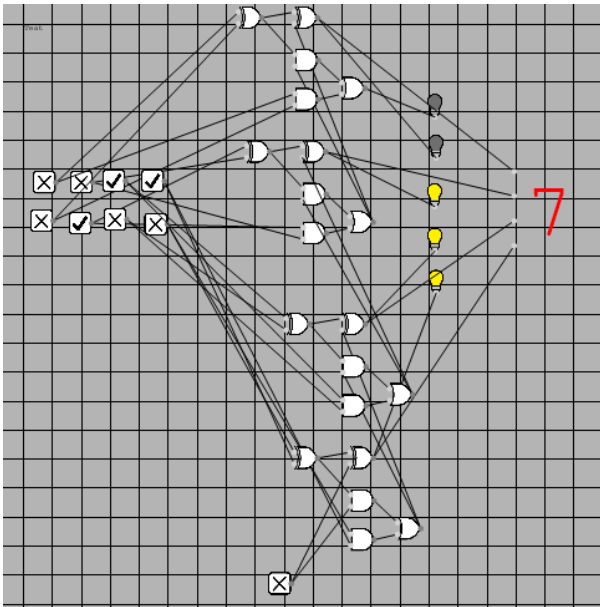
✓



The program will include output components such as a light, to show the user the result of the circuit's calculation.

The program includes two types of output: the light bulb and hexadecimal digit display.

The inclusion of the hexadecimal digit display naturally leads to the question “why is there no hexadecimal digit input component?”. To support this feature properly, there would need to be an editable textbox embedded in the viewport. Due to the limitations of GDI (which Windows Forms is based off), UI controls cannot be affected by arbitrary graphics transformations such as scaling and translations. This would be possible in a more sophisticated GUI framework such as HTML5 or WPF, but those are also more complex and a whole rewrite at this stage is not worth it for such a small feature.

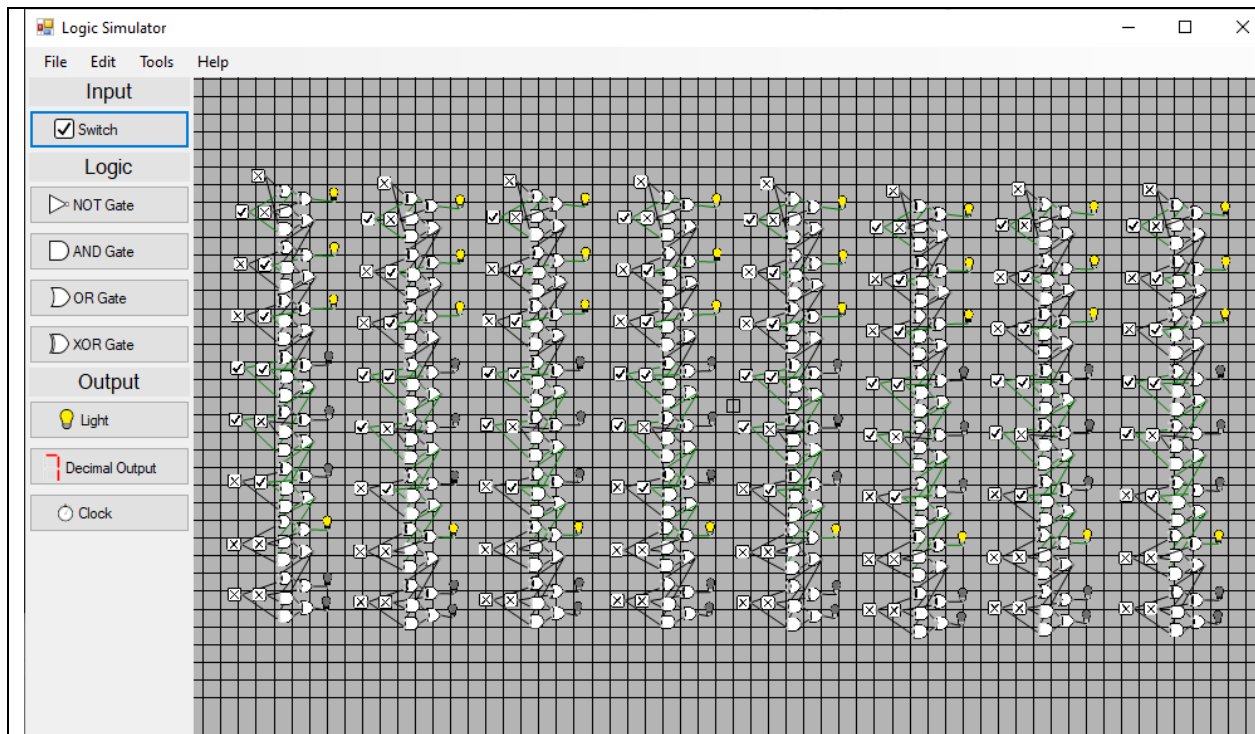


The program must have the capability to create virtual integrated circuits.

This criterion is discussed in the following section: “Requirements Evaluation”.

The program must be able to simulate complex circuits (such as a large adder, or memory) without crashing or locking up the UI.

As a stress test I have loaded up 8 copies of the 8-bit adder circuit. The UI still performs well even with over a hundred components on screen.



Requirements Evaluation

During development, I realized that there were better ways to fulfil certain requirements by changing them. For example, I originally intended for there to be an integrated circuit system to allow reuse of common groups of components. That would have been quite complicated to implement, but I realised that with minimal changes I could make the load feature “import” a circuit, achieving the same result.

There were some features that I did not have the time to implement. Copy and paste, undo/redo, and selecting multiple items at once. For many purposes the functionality of copy and paste is adequately implemented by saving and importing, so time was better spent working on other parts of the project.

Undo/redo could have been implemented by making a copy of the entire circuit every time a change is made. The easiest way to make a copy of the circuit would be to serialize it into a byte array in memory (like saving the file but keeping the data in memory rather than writing it to the disk). However, this would use a very large amount of memory even with a small number of changes made. Another common solution for undo/redo is to store a stack of undo actions. Every time a change is made to the circuit a function that undoes that change is pushed onto the stack. For example, if the user deletes a component, the undo function adds it back. When the user presses Ctrl-Z the function at the top of the undo stack is executed. Then a redo function must be added to another stack. As you can see, this solution would have been quite complicated.

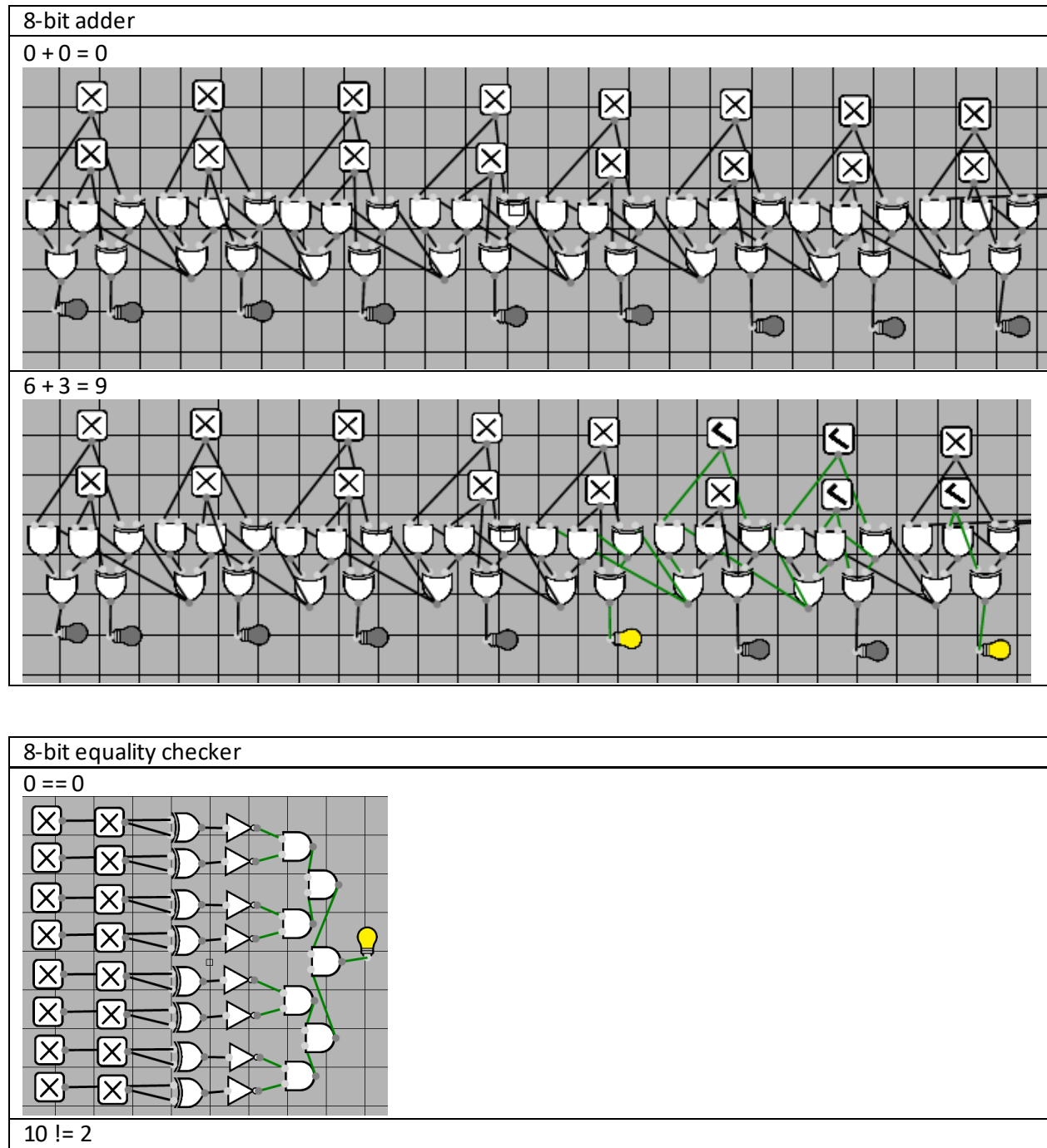
Overall, I think that the program meets the requirements, with the most important success criteria implemented well. The user can build circuits with input and output components for interactivity, and they can save their work and reload it.

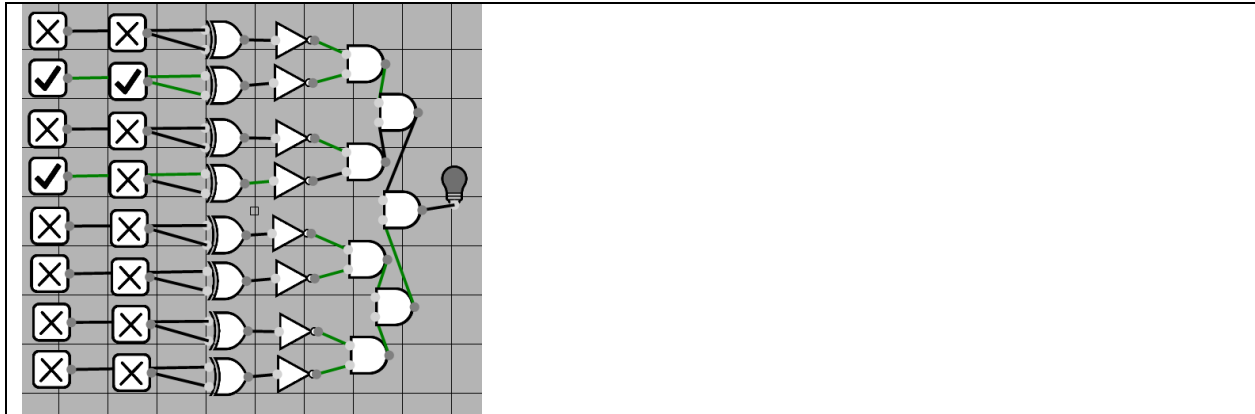
I have structured the code to make it easy to maintain and update in the future. All of the user interaction code is in one file, and each component has its own file. This means that if there is an error

with one of these features it will be obvious where to look for the bug. The saving and loading is modular, which makes it easy to extend when new components are added. There just needs to be a [Serializable] attribute added to the class. The variables have suitable names and the code is sufficiently commented to make the code easy to understand for other programmers.

Testing of the Completed Solution

Here are some tests of the completed program with various large circuits.





Appendix – Code Listing

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace LogicSimulator
{
    static class Program
    {
        public static bool DebugDraw = false;

        static void UnhandledException(object sender,
        UnhandledExceptionEventArgs e)
        {
            var UnhandledEx = (Exception)e.ExceptionObject;

            MessageBox.Show($"{UnhandledEx.Message}\n{UnhandledEx.StackTrace}");
        }

        static void UIThreadException(object sender, ThreadExceptionEventArgs
        t)
        {
            MessageBox.Show($"{t.Exception.Message}\n{t.Exception.StackTrace}");
        }
    }
}
```

```

    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        // Add exception handlers to show an error message when an
        unhandled exception occurs rather than crashing the program.
        // This gives the user the chance to save their work.

        Application.ThreadException += new
        ThreadExceptionHandler(UIThreadException);
        AppDomain.CurrentDomain.UnhandledException += new
        UnhandledExceptionHandler(UnhandledException);

        Application.ThreadException += new
        ThreadExceptionHandler(UIThreadException);

        Application.SetUnhandledExceptionMode(UnhandledExceptionMode.CatchException);
        AppDomain.CurrentDomain.UnhandledException += new
        UnhandledExceptionHandler(UnhandledException);

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}

```

Form1.cs

```

using LogicSimulator.Simulation;
using LogicSimulator.Simulation.Nodes;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Runtime.Serialization.Formatters.Binary;
using System.Text;

```

```
using System.Threading.Tasks;
using System.Windows.Forms;

namespace LogicSimulator
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // Functions for adding components to the circuit.

            private void switchButton_Click(object sender, EventArgs e)
            {
                circuitViewControl1.AddNode(new
Switch(circuitViewControl1.ScreenCentre));
            }

            private void lightButton_Click(object sender, EventArgs e)
            {
                circuitViewControl1.AddNode(new
LightNode(circuitViewControl1.ScreenCentre));
            }

            private void notGateButton_Click(object sender, EventArgs e)
            {
                circuitViewControl1.AddNode(new
NotGate(circuitViewControl1.ScreenCentre));
            }

            private void andGateButton_Click(object sender, EventArgs e)
            {
                circuitViewControl1.AddNode(new
AndGate(circuitViewControl1.ScreenCentre));
            }
        }
    }
}
```

```
private void orGateButton_Click(object sender, EventArgs e)
{
    circuitViewControl1.AddNode(new
OrGate(circuitViewControl1.ScreenCentre));
}

private void xorGateButton_Click(object sender, EventArgs e)
{
    circuitViewControl1.AddNode(new
XorGate(circuitViewControl1.ScreenCentre));
}
private void _7segButton_Click(object sender, EventArgs e)
{
    circuitViewControl1.AddNode(new
DecimalOutputNode(circuitViewControl1.ScreenCentre));
}

private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Show a dialog to allow the user to select where to save the
file.

    var d = new SaveFileDialog();
    if (d.ShowDialog() == DialogResult.OK)
    {
        using (var f = d.OpenFile())
        {
            var b = new BinaryFormatter();
            b.Serialize(f, circuitViewControl1.circuit);
        }
    }
}

private void openToolStripMenuItem_Click(object sender, EventArgs e)
{
    // Show a dialog to allow the user to select where to open a file
from.

    var d = new OpenFileDialog();
    if (d.ShowDialog() == DialogResult.OK)
```



```

        {
            using (var f = d.OpenFile())
            {
                var b = new BinaryFormatter();
                var circuit = (Circuit)b.Deserialize(f);
                circuitViewControl1.circuit = circuit;
                circuitViewControl1.Invalidate();
            }
        }
    }

    private void importButton_Click(object sender, EventArgs e)
    {
        // Import is like open, but the components are added to the current
        file instead of replacing them.

        var d = new OpenFileDialog();
        if (d.ShowDialog() == DialogResult.OK)
        {
            using (var f = d.OpenFile())
            {
                var b = new BinaryFormatter();
                var circuit = (Circuit)b.Deserialize(f);

                var minx = circuit.Nodes.Select(n => n.Rect.X).Min();
                var miny = circuit.Nodes.Select(n => n.Rect.Y).Min();

                // The components add added with their positions relative
                to the centre of the screen.
                var basePos = circuitViewControl1.ScreenCentre;
                basePos.X -= minx;
                basePos.Y -= miny;

                foreach (var n in circuit.Nodes)
                {
                    n.Rect.X += basePos.X;
                    n.Rect.Y += basePos.Y;
                    circuitViewControl1.circuit.Nodes.Add(n);
                }
            }
        }
    }

```

```
        circuitViewControl1.Invalidate();
    }
}
}
```

CircuitViewControl.cs

```
using LogicSimulator.Simulation;
using LogicSimulator.Simulation.Nodes;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Text;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace LogicSimulator
{
    // CircuitViewControl contains the user interaction code.
    public partial class CircuitViewControl : Control
    {
        public CircuitViewControl()
        {
            InitializeComponent();

            // When the window is resized the graphics should be redrawn.
            ResizeRedraw = true;
            // Enable double buffering for smooth graphics with no frame
            tearning. DoubleBuffered = true;

            // Event handlers for user interaction.
        }
    }
}
```

```
        MouseDown += CircuitViewControl_MouseDown;
        MouseUp += CircuitViewControl_MouseUp;
        MouseMove += CircuitViewControl_MouseMove;
        MouseWheel += CircuitViewControl_MouseWheel;

        circuit = new Circuit();
    }

    // circuit is the Circuit that the user is editing.
    // All components are stored here.
    public Circuit circuit;

    // When a user clicks a button to add a component to the circuit it
must    // be added to the list of nodes and the screen is refreshed.
    public void AddNode(Node node)
    {
        circuit.Nodes.Add(node);
        Invalidate();
    }

    Font font = new Font(FontFamily.GenericMonospace, 10);

    // Camera position and zoom level.
    PointF camPos = new PointF(0, 0);
    float scale = 1;

    // Return the coordinates of the centre of the screen in world-space.
    // This is used when adding a new component to position it in the
centre.    public PointF ScreenCentre => ScreenToWorld(new PointF(Width / 2,
Height / 2));

    protected override void OnPaint(PaintEventArgs pe)
    {
        base.OnPaint(pe);
    }
}
```

```
var g = pe.Graphics;
g.SmoothingMode = SmoothingMode.AntiAlias;
g.TextRenderingHint = TextRenderingHint.AntiAlias;

// Use the camera.
g.ScaleTransform(scale, scale);
g.TranslateTransform(-camPos.X, -camPos.Y);

// Draw the background grid, which helps visualise the camera zoom.
DrawGridLines(g);

// If the user is dragging a wire between components, draw a
visualisation.
DrawWireDrag(g);

// Draw the components of the circuit (gates, buttons, etc.).
circuit.OnPaint(g);
}

private void DrawWireDrag(Graphics g)
{
    if (dragType != DragType.Wire)
    {
        return;
    }

    g.DrawLine(Pens.Red, dragStart, ScreenToWorld(lastMousePos));
}

private void DrawGridLines(Graphics g)
{
    g.SmoothingMode = SmoothingMode.HighSpeed;

    var lineDistance = 50;
    var numLinesX = (int)(Width / lineDistance / scale + 1);
    var numLinesY = (int)(Height / lineDistance / scale + 1);
```

```

        // Draw vertical lines.
        for (int i = 0; i <= numLinesX; i++)
        {
            var x = i * lineDistance + camPos.X - camPos.X % lineDistance;

            g.DrawLine(Pens.Black, x, camPos.Y, x, Height / scale +
camPos.Y);
        }
        // Draw horizontal lines.
        for (int i = 0; i <= numLinesY; i++)
        {
            var y = i * lineDistance + camPos.Y - camPos.Y % lineDistance;

            g.DrawLine(Pens.Black, camPos.X, y, Width / scale + camPos.X,
y);
        }

        g.SmoothingMode = SmoothingMode.AntiAlias;
    }

    // Convert from screen coordinates to world coordinates.
    // For example this is used to convert the position of the cursor
    // into a position in the circuit to figure out if the cursor is over
    an object.
    public PointF ScreenToWorld(PointF screen)
    {
        return new PointF(
            screen.X / scale + camPos.X,
            screen.Y / scale + camPos.Y);
    }

    private void CircuitViewControl_MouseWheel(object sender,
MouseEventArgs e)
    {
        // Zoom the camera centred around the cursor when the user scrolls
        // the mouse wheel.

```

```
float scaleChange = scale * e.Delta / 2000f;
float newScale = scale + scaleChange;

var oldWorldMouseX = e.X / scale + camPos.X;
var oldWorldMouseY = e.Y / scale + camPos.Y;

var worldMouseX = e.X / newScale + camPos.X;
var worldMouseY = e.Y / newScale + camPos.Y;

camPos.X -= worldMouseX - oldWorldMouseX;
camPos.Y -= worldMouseY - oldWorldMouseY;

scale = newScale;

Invalidate();
}

// Some state for user interaction.
DragType dragType;
PointF lastMousePos = new PointF(0, 0);
Node draggedNode = null;
PointF dragStart;
Output draggedOutput;

// Has the mouse moved since the click was pressed?
bool dragMoved;

private void CircuitViewControl_MouseMove(object sender, MouseEventArgs
e)
{
    if (dragType != DragType.None)
    {
        dragMoved = true;
    }

    // The amount that the cursor has moved.
```

```
float dx = (e.X - lastMousePos.X) / scale;
float dy = (e.Y - lastMousePos.Y) / scale;

switch (dragType)
{
    case DragType.None:
        break;

    case DragType.Camera:
        // Move the camera.
        camPos.X -= dx;
        camPos.Y -= dy;

        Invalidate();

        break;

    case DragType.Node:
        // Move a component.
        draggedNode.Rect.X += dx;
        draggedNode.Rect.Y += dy;

        Invalidate();

        break;

    case DragType.Wire:
        Invalidate();
        break;
}

lastMousePos = e.Location;
}
```

```

private void CircuitViewControl_MouseUp(object sender, MouseEventArgs
e)
{
    PointF worldCursor = ScreenToWorld(e.Location);

    if (dragType == DragType.Wire)
    {
        // Determine which node the cursor is over.
        foreach (var node in circuit.Nodes)
        {
            // Determine which input of the node the cursor is over.
            foreach (var i in node.Inputs)
            {
                var posx = node.Rect.X + i.Pos.X;
                var posy = node.Rect.Y + i.Pos.Y;
                var r = 5;

                var rect = RectangleF.FromLTRB(posx - r, posy - r, posx
+ r, posy + r);
                if (rect.Contains(worldCursor))
                {
                    Console.WriteLine("drop");

                    // Connect the output to an input.
                    if (!draggedOutput.Inputs.Contains(i))
                    {
                        // If the input is already connected,
disconnect it first.

                        if (i.Source != null)
                        {
                            i.Source.Inputs.Remove(i);
                        }

                        i.Source = draggedOutput;
                        draggedOutput.Inputs.Add(i);

                        circuit.EvaluateCircuit(draggedOutput.Node);
                    }
                }
            }
        }
    }
}

```



```

        goto loopend;
    }
}
}
}
else if (dragType == DragType.Node)
{
    // If there is a mouseup event over an interactible node,
    interact with it.
    foreach (var node in circuit.Nodes)
    {
        if (node.Rect.Contains(worldCursor))
        {
            if (node is InputNode inp && !dragMoved)
            {
                Console.WriteLine("interact");
                inp.Interact(PointF.Subtract(worldCursor, new
SizeF(node.Rect.Location)), circuit);
            }

            goto loopend;
        }
    }
}
loopend;;

dragType = DragType.None;
draggedNode = null;
draggedOutput = null;
dragMoved = false;

Invalidate();
}

private void CircuitViewControl_MouseDown(object sender, MouseEventArgs
e)
{
    // This function determines what type of drag will be started when
    the mouse is clicked.

```

```

dragType = DragType.None;
draggedNode = null;
draggedOutput = null;
dragMoved = false;

PointF worldCursor = ScreenToWorld(e.Location);
dragStart = worldCursor;

// You cannot remove items from a list while iterating through it,
// so nodes to be deleted are added
// to another list and deleted after the main iteration.
List<Node> deleteList = new List<Node>();

foreach (var node in circuit.Nodes)
{
    // If the cursor is over an input start dragging a wire.
    foreach (var o in node.Outputs)
    {
        var posx = node.Rect.X + o.Pos.X;
        var posy = node.Rect.Y + o.Pos.Y;
        var r = 5;

        var rect = RectangleF.FromLTRB(posx - r, posy - r, posx +
r, posy + r);
        if (rect.Contains(worldCursor))
        {
            Console.WriteLine("wire");

            dragType = DragType.Wire;
            draggedOutput = o;

            return;
        }
    }

    // If the cursor is over a node start dragging it.
    if (node.Rect.Contains(worldCursor))
    {

```

```
        if (e.Button == MouseButton.Left)
        {
            dragType = DragType.Node;
            draggedNode = node;
            return;
        }
        else if (e.Button == MouseButton.Right)
        {
            deleteList.Add(node);
        }
    }
}

foreach (var node in deleteList)
{
    circuit.DeleteNode(node);
}

dragType = DragType.Camera;
}
}

enum DragType
{
    None,
    Camera,
    Node,
    Wire
}
}
```

Wire.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation
{
```

```
// A node has a list of inputs and outputs that control which nodes are
connected and how.
```

```
[Serializable]
public class Input
{
    public Node Node;

    // The position of the input relative to the node.
    public PointF Pos;

    // Where this input will read a value from
    public Output Source;

    public Input(Node node, PointF pos)
    {
        Node = node;
        Pos = pos;
    }
}
```

```
[Serializable]
public class Output
{
    public Node Node;

    // The position of the output relative to the node.
    public PointF Pos;

    public bool Value;
    public List<Input> Inputs = new List<Input>();

    public Output(Node node, PointF pos)
    {
        Node = node;
        Pos = pos;
    }
}
```

```

    }
}
Node.cs
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation
{
    // Node is the superclass of all components (logic, inputs, etc.).
    [Serializable]
    public abstract class Node
    {
        // The rectangle of the screen where it is drawn, also used to
        // determine mouse hit detection.
        public RectangleF Rect;

        public Input[] Inputs = Array.Empty<Input>();
        public Output[] Outputs = Array.Empty<Output>();

        public Node(RectangleF rect)
        {
            Rect = rect;
        }

        // The Update function is used by the Circuit.EvaluateCircuit function.
        // Each subclass of Node should implement Update and use it to
        // calculate what the outputs
        // should be given the inputs.
        public abstract void Update();

        // OnPaint draws the inputs, outputs, and wires of a node.
        // Subclasses should override this method to draw themselves, and then
        // call base.OnPaint.
        public virtual void OnPaint(Graphics g)
        {
            if (Program.DebugDraw)

```

```
        {
            g.DrawRectangle(Pens.Red, Rect.X, Rect.Y, Rect.Width,
Rect.Height);
        }

        foreach (var input in Inputs)
        {
            DrawInput(g, input);
        }

        foreach (var output in Outputs)
        {
            DrawOutput(g, output);
        }
    }

    private void DrawInput(Graphics g, Input input)
    {
        float radius = 5;

        float x = Rect.X + input.Pos.X - radius;
        float y = Rect.Y + input.Pos.Y - radius;

        g.FillEllipse(
            Brushes.LightGray,
            x,
            y,
            radius * 2,
            radius * 2);
    }

    static Pen penOff = new Pen(Color.Black, 3);
    static Pen penOn = new Pen(Color.Green, 3);

    private void DrawOutput(Graphics g, Output output)
    {
        float radius = 5;
```

```

        float x = Rect.X + output.Pos.X - radius;
        float y = Rect.Y + output.Pos.Y - radius;

        foreach (var i in output.Inputs)
        {
            float x2 = i.Node.Rect.X + i.Pos.X;
            float y2 = i.Node.Rect.Y + i.Pos.Y;

            Pen pen = penOff;
            if (i.Source.Value) pen = penOn;

            g.DrawLine(pen, x + radius, y + radius, x2, y2);
        }

        g.FillEllipse(
            Brushes.Gray,
            x,
            y,
            radius * 2,
            radius * 2);
    }
}

```

ExampleNode.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation
{
    [Serializable]
    public class ExampleNode : Node
    {
        public ExampleNode(PointF pos) : base(
            new RectangleF(pos.X, pos.Y, 40, 40))
        {
        }
    }
}

```

```

    {
        Inputs = new Input[]
        {
            new Input(this, new PointF(0, 5)),
            new Input(this, new PointF(0, 15)),
            new Input(this, new PointF(0, 25)),
            new Input(this, new PointF(0, 35)),
        };
        Outputs = new Output[]
        {
            new Output(this, new PointF(40, 10)),
            new Output(this, new PointF(40, 30)),
        };
    }

    public override void Update()
    {

    }

    public override void OnPaint(Graphics g)
    {
        g.FillRectangle(Brushes.White, Rect);

        base.OnPaint(g);
    }
}

```

Circuit.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation

```



```
{  
    // The Circuit class is a container for the components.  
  
    [Serializable]  
    public class Circuit  
    {  
        public List<Node> Nodes = new List<Node>();  
  
        // Draw all of the components.  
        public void OnPaint(Graphics g)  
        {  
            foreach (var node in Nodes)  
            {  
                node.OnPaint(g);  
            }  
        }  
  
        // Remove a node from the circuit, also disconnecting all of its  
        // connections.  
        public void DeleteNode(Node node)  
        {  
            foreach (var inp in node.Inputs)  
            {  
                if (inp.Source == null)  
                {  
                    continue;  
                }  
                inp.Source.Inputs.Remove(inp);  
            }  
            foreach (var outp in node.Outputs)  
            {  
                foreach (var inp in outp.Inputs)  
                {  
                    inp.Source = null;  
                }  
            }  
            Nodes.Remove(node);  
        }  
  
        // Update the circuit using breadth first traversal of a graph.
```

// Components that are not connected to the root node (the one changed by an interaction by the user)
 // are not updated, so a simple loop over all components would be inefficient.

```
public void EvaluateCircuit(Node root)
{
    var queue = new Queue<Node>();
    queue.Enqueue(root);
    var visited = new HashSet<Node>();
    foreach (var output in root.Outputs)
    {
        foreach (var i in output.Inputs)
        {
            var node = i.Node;
            if (!queue.Contains(node))
            {
                queue.Enqueue(node);
            }
        }
    }

    while (queue.Count != 0)
    {
        var node = queue.Dequeue();
        node.Update();
        visited.Add(node);

        foreach (var output in node.Outputs)
        {
            foreach (var i in output.Inputs)
            {
                var node2 = i.Node;
                if (!visited.Contains(node2))
                {
                    queue.Enqueue(node2);
                }
            }
        }
    }
}
```

XorGate.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Takes 2 binary inputs and outputs the logical XOR of them.
    [Serializable]
    public class XorGate : LogicNode
    {
        static int nodeSize = 40;

        public XorGate(PointF pos) : base(new RectangleF(pos.X, pos.Y,
nodeSize, nodeSize))
        {
            Inputs = new Input[]
            {
                new Input(this, new PointF(0, 10)),
                new Input(this, new PointF(0, 30)),
            };
            Outputs = new Output[]
            {
                new Output(this, new PointF(40, 20))
            };
        }

        public override void Update()
        {
            if (Inputs[0].Source == null || Inputs[1].Source == null)
            {
                Outputs[0].Value = false;
                return;
            }

            Outputs[0].Value = Inputs[0].Source.Value ^ Inputs[1].Source.Value;
        }
    }
}
```

```

        static Bitmap img = Properties.Resources.xor_gate;

        public override void OnPaint(Graphics g)
        {
            g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

            base.OnPaint(g);
        }
    }
}

```

Switch.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // A switch that is toggled on and off when the user clicks on it.
    [Serializable]
    public class Switch : InputNode
    {
        static int nodeSize = 40;

        public Switch(PointF pos) : base(new RectangleF(pos.X, pos.Y, nodeSize,
nodeSize))
        {
            Outputs = new Output[]
            {
                new Output(this, new PointF(40, 20))
            };
        }

        bool isOn;

        public override void Interact(PointF point, Circuit circuit)
        {

```

```

        isOn = !isOn;
        circuit.EvaluateCircuit(this);
    }

    public override void Update()
    {
        Outputs[0].Value = isOn;
    }

    static Bitmap offImg = Properties.Resources.toggle_off;
    static Bitmap onImg = Properties.Resources.toggle_on;

    public override void OnPaint(Graphics g)
    {
        Bitmap img = offImg;
        if (isOn)
        {
            img = onImg;
        }
        g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

        base.OnPaint(g);
    }
}

```

OutputNode.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Common superclass of all output nodes.
    [Serializable]
    public abstract class OutputNode : Node
    {
        public OutputNode(RectangleF rect) : base(rect)
    }
}

```

```

        {
        }
    }
}

```

OrGate.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Takes 2 binary inputs and outputs the logical OR of them.
    [Serializable]
    public class OrGate : LogicNode
    {
        static int nodeSize = 40;

        public OrGate(PointF pos) : base(new RectangleF(pos.X, pos.Y, nodeSize,
nodeSize))
        {
            Inputs = new Input[]
            {
                new Input(this, new PointF(0, 10)),
                new Input(this, new PointF(0, 30)),
            };
            Outputs = new Output[]
            {
                new Output(this, new PointF(40, 20))
            };
        }

        public override void Update()
        {
            if (Inputs[0].Source == null || Inputs[1].Source == null)
            {
                Outputs[0].Value = false;
                return;
            }
        }
    }
}

```

```

        Outputs[0].Value = Inputs[0].Source.Value ||
Inputs[1].Source.Value;
    }

    static Bitmap img = Properties.Resources.or_gate;

    public override void OnPaint(Graphics g)
    {
        g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

        base.OnPaint(g);
    }
}

```

NotGate.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Takes one input and outputs the logical NOT.
    [Serializable]
    public class NotGate : LogicNode
    {
        static int nodeSize = 40;

        public NotGate(PointF pos) : base(new RectangleF(pos.X, pos.Y,
nodeSize, nodeSize))
        {
            Inputs = new Input[]
            {
                new Input(this, new PointF(0, 20))
            };
            Outputs = new Output[]

```

```

        {
            new Output(this, new PointF(40, 20))
        };
    }

    public override void Update()
    {
        if (Inputs[0].Source == null)
        {
            Outputs[0].Value = true;
            return;
        }

        Outputs[0].Value = !Inputs[0].Source.Value;
    }

    static Bitmap img = Properties.Resources.not_gate;

    public override void OnPaint(Graphics g)
    {
        g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

        base.OnPaint(g);
    }
}

```

LogicNode.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Superclass of all logic gate nodes
    [Serializable]
    public abstract class LogicNode : Node

```



```

    {
        protected LogicNode(RectangleF rect) : base(rect)
        {
        }
    }
}

```

LightNode.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Takes a single input and graphically shows an on or off light bulb
    image.
    [Serializable]
    public class LightNode : OutputNode
    {
        static int nodeSize = 40;

        public LightNode(PointF pos) : base(new RectangleF(pos.X, pos.Y,
nodeSize, nodeSize))
        {
            Inputs = new Input[]
            {
                new Input(this, new PointF(20, 40))
            };
        }

        bool isOn = false;

        public override void Update()
        {
            isOn = Inputs[0].Source.Value;
        }

        static Bitmap offImg = Properties.Resources.light_off;

```

```

        static Bitmap onImg = Properties.Resources.light_on;

        public override void OnPaint(Graphics g)
        {
            Bitmap img = offImg;
            if (isOn)
            {
                img = onImg;
            }
            g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

            base.OnPaint(g);
        }
    }
}

```

InputNode.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Common superclass of all Inputs.
    [Serializable]
    public abstract class InputNode : Node
    {
        protected InputNode(RectangleF rect) : base(rect)
        {
        }

        // This function is called when the user clicks on an input.
        public abstract void Interact(PointF point, Circuit circuit);
    }
}

```

DecimalOutputNode.cs

```

using System;
using System.Collections.Generic;
using System.Drawing;

```

```

using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Takes 4 binary inputs and displays them as a single decimal number from 0 to 15.
    [Serializable]
    public class DecimalOutputNode : OutputNode
    {
        static int width = 77;
        static int height = 128;

        public DecimalOutputNode(PointF pos) : base(new RectangleF(pos.X, pos.Y, width, height))
        {
            Inputs = new Input[]
            {
                new Input(this, new PointF(0, 0)),
                new Input(this, new PointF(0, 128*1/3)),
                new Input(this, new PointF(0, 128*2/3)),
                new Input(this, new PointF(0, 128)),
            };
        }

        // Get the value of the i'th input as a bit.
        int inputValue(int i)
        {
            return (Inputs[i].Source?.Value ?? false) ? 1 : 0;
        }

        // The number to display
        int number = 0;
        string numStr = "0";

        public override void Update()
        {
            // Combine the bits into a 4 bit input.
            number =

```

```
inputValue(0) * 8 +
inputValue(1) * 4 +
inputValue(2) * 2 +
inputValue(3);

numStr = number.ToString();
}

Font font = new Font(FontFamily.GenericMonospace, 100);

public override void OnPaint(Graphics g)
{
    g.DrawString(numStr, font, Brushes.Red, Rect.Location);

    base.OnPaint(g);
}
}
```

AndGate.cs

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LogicSimulator.Simulation.Nodes
{
    // Takes 2 binary inputs and outputs the logical AND of them.
    [Serializable]
    public class AndGate : LogicNode
    {
        static int nodeSize = 40;

        public AndGate(PointF pos) : base(new RectangleF(pos.X, pos.Y,
            nodeSize, nodeSize))
        {
            Inputs = new Input[]
            {
```

```
        new Input(this, new PointF(0, 10)),
        new Input(this, new PointF(0, 30)),
    };
    Outputs = new Output[]
    {
        new Output(this, new PointF(40, 20))
    };
}

public override void Update()
{
    if (Inputs[0].Source == null || Inputs[1].Source == null)
    {
        Outputs[0].Value = false;
        return;
    }

    Outputs[0].Value = Inputs[0].Source.Value &&
Inputs[1].Source.Value;
}

static Bitmap img = Properties.Resources.and_gate;

public override void OnPaint(Graphics g)
{
    g.DrawImage(img, Rect.X, Rect.Y, nodeSize, nodeSize);

    base.OnPaint(g);
}
}
```