Code analysis

# Things to Address

The code used in the implementation of this program is efficient and the result of significant research into user interfaces and Object Orientated Design. There are however some aspects of the code I feel it is important to address in MainWindow.Xaml.CS which implements the core module for this program. Specifically that at first glance it may appear that I have made use of “global variables” when in fact these are examples of public Enums in use as they are a declaration of a dictionary of named integer constants that are the blueprint for the Enum, as described in Microsofts documentation for the language C# (2019, <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/enum>).

namespace A\_level\_course\_work\_Logic\_Gate

{

public enum Drag\_State { Null,Main\_Can,Sub\_Can,Link\_Mode\_Sub}

public enum IO\_Type { Null, Gate, IO }

public enum Detection\_State { Null, Detected }

public enum Gate\_Type { And, Nand, Not, Or, Nor, Xor, Xnor, Transformer }

public partial class MainWindow : Window, Canvas\_Variables

{

As a result these are considered to be constant and do not hold a value as a result of them being Namespace variables and Enums are used in this manner across every class in the namespace. I have also considered the implementation of class structures to take into account the lack of multiple inheritance within the C# language, which results in more robust code and far more limited interplay/side-effects between functions.

When you create a method that just sets the variable to the value of the parameter that is just the same as Variable {set;} in C#. When you create a method that returns the variable that is just the same as Variable {get;} in C#. So you can either have 2 methods for every one of your variables taking up 7 lines of code which are generally far less efficient than the built in functions or you could just add { get; set;} to each variables where they apply. This isn’t about breaking the way getters and setters are used but taking advantage of the facilities available within the language.

My research into the use of Getters and setters included the following websites and texts:

* How to use getter and setter: <https://javasolutionsguide.blogspot.com/2016/04/encapsulation.html>
* C# Getter and setter: <https://www.dotnetperls.com/property>
* Programming C#: by O’Rielly (2018)

Instance/Class Variables

public partial class MainWindow : Window, Canvas\_Variables

{

//varaibles that need to be accessed all around the code

public List<Gate\_Class> Gate\_List { get; set; } = new List<Gate\_Class>();

public List<Line\_Class> Line\_List { get; set; } = new List<Line\_Class>();

//make these custom classes

public List<Input\_Button> Input\_Button\_List { get; set; } = new List<Input\_Button>();

public List<Output\_Circle> Output\_Circle\_List { get; set; } = new List<Output\_Circle>();

private File\_Creation\_Class File\_Worker { get; set; }

Instance variables are variables that can be accessed in the object they are in, so the class. A good way of thinking about it is that global variables can be declared in one locations and accessed in another without any links, instance variables can’t do this, and they required to be passed by attributes or interfaces.

Java but still covers the topic: <https://www.quora.com/What-is-the-difference-between-global-variables-and-instance-variables-in-Java>

# Structure of project

The MainWindow is the main part of the project. It holds all the data and has most of the methods for the interactions. This is why a lot of the other class will have MainWindow as an attribute. This is bad practice if the mainwindow is bidirectional. So I made sure that the variable doesn’t have a setter and it is only unidirectional.

The final setup for how each class was based off is due to how variables linked together and their composition with each other. This is because I could’ve put input\_Button and Output\_Circle inside gate class but that doesn’t always make sense as a gate might have no input or output buttons but the gate will still waste the memory hold variables that aren’t used. It’s also the same case for the line class as they are always created on the output port.

I keep all my variables at the top of the class then the constructor. For the ordering of methods it’s just based on when I created them.

I try and kept the amount of code in one method to a minimum, make functions for any overlapping methods and a method should only have one function. I think I achieved this requirements. Some methods might be deemed too long to by some but I personally don’t like having methods that are 4-6 lines per each one.

The classes that the user interact with are all stored in list because new objects can be added without a limit and removed to never be returned.

The program is split into lots of documents. Each class gets their own document and if it’s a group they get their own folder. This is for ease of access and readability of the class. There is also external documents that store the images for the gates. This is done with a xaml file structure and allows easy changes and third party software. It’s also important for vector images and the rescaling factor that it provides.

# MainWindow

Structure of class and variables:

The MainWindow is the default class that the code will start with. It inherits window but also an interface called canvas variables. This is because C# doesn’t support multiple inheritance canvas class which should inherit MainWindow but can’t due to it already inheriting Canvas needs some variables. So I send them as an interface as those are the overlapping variables needed between those 2 classes.

The Variables are all laid out on the top with the constructor underneath. With the variables they should all be private as MainWindow never inherits (So really making them public, protect or private makes zero difference in this class.) however due to the state of Visual Studio 2019 a problem means I can’t change them off public. So the problem is in an interface to change the declaration type to private instead of the default public you need C# 8.0 or greater. I’m using .net framework and the default C# is 7.4. You would think you could just change the C# version to 8.0 or greater as the framework does support it but VS19 doesn’t. Microsoft says to change it manually in the program file but reports says it doesn’t work and gets override (I didn’t try as I don’t even know where to start with messing that file type).

With the variables in this class they are the main data type and data storage for everything in the program. This is the backbone that every class works between and due to C# not support multiple inheritance I’m using composition classes (This causes a problem with the file system). I also have the custom getter and setter for Objects that need them.

public partial class MainWindow : Window, Canvas\_Variables

{

//varaibles that need to be accessed all around the code

public List<Gate\_Class> Gate\_List { get; set; } = new List<Gate\_Class>();

public List<Line\_Class> Line\_List { get; set; } = new List<Line\_Class>();

//make these custom classes

public List<Input\_Button> Input\_Button\_List { get; set; } = new List<Input\_Button>();

public List<Output\_Circle> Output\_Circle\_List { get; set; } = new List<Output\_Circle>();

private File\_Creation\_Class File\_Worker { get; set; }

//program info

public bool Drag { get; set; } = false;

public uint Delay\_Intervals { get; set; } = 1;

public int Drag\_Num { get; set; } = 0;

private bool \_link = false;

public bool Sim\_Running { get; set; } = false;

public bool Sim\_Busy { get; set; } = false;

public bool Saved { get; set; } = false;

public string File\_Location { get; set; } = "";

public bool Link

{

get { return \_link; }

set

{

\_link = value;

if (\_link)

{

Link\_Button.Content = "Link";

}

else if (!\_link)

{

Link\_Button.Content = "Drag";

}

}

}

private Drag\_State drag\_mode = Drag\_State.Null;

public Drag\_State Drag\_Mode

{

get { return drag\_mode; }

set

{

switch (value)

{

case (Drag\_State.Null):

Drag = false;

break;

default:

Drag = true;

break;

}

drag\_mode = value;

}

}

public int Linking\_ID { get; set; } = 0;

public bool IO\_Active { get; set; } = false;

//UI elements that can't be added in the XAML

public Canvas\_Class Sub\_Canvas { get; set; }

public Rectangle BackGround\_Rect { get; set; }

//Threads that do the work simulatnously with the UI element

private BackgroundWorker \_worker = new BackgroundWorker();

private BackgroundWorker Simulator\_Worker = new BackgroundWorker();

Progress\_Bar\_Window Progress\_Window = new Progress\_Bar\_Window(0);

Constructor:

There are 2 stages to the constructor. The first is the Mainwindow constructor which just allocates the methods for the background workers. It also sets up the file class.

The second constructor is for when the canvas has loaded in the WPF. This is because canvas class requires other variables from both XAML and Mainwindow needed to be loaded to be passed through to the canvas class. Canvas Class couldn’t be added to the XAML directly as it’s a custom override class of a UI object they needed to be added in the CS code that’s the reasoning for adding it separate from the other UI objects. I also added the rectangle to the canvas just to make the UI look better and fix the area that the user has to use.

public MainWindow()

{

InitializeComponent();

\_worker.DoWork += WorkerDoWork;

\_worker.RunWorkerCompleted += WorkerRunWorkerCompleted;

Simulator\_Worker.DoWork += Simulator\_Work;

File\_Worker = new File\_Creation\_Class(this);

}

//Need the UI to load before adding the canvas as it's added in the CS instead of the XAML

private void Canvas\_Border\_Loaded(object sender, RoutedEventArgs e)

{

Sub\_Canvas = new Canvas\_Class(this, ref Canvas\_Border, this);

SetUp\_Canvas();

}

public void SetUp\_Canvas()

{

Canvas\_Border.Child = Sub\_Canvas;

BackGround\_Rect = new Rectangle { Height = 4000, Width = 4000, Fill = Brushes.White };

Sub\_Canvas.Children.Add(BackGround\_Rect);

Canvas.SetLeft(BackGround\_Rect, -1000);

Canvas.SetTop(BackGround\_Rect, -1000);

}

Methods:

Rect\_Button\_MouseLeftButtonDown:

This is the method for which you click the rectangles and a new one spawns in. The first IF statement just checks to see that you aren’t already dragging or doing something right now and that you are in the correct mode for adding a new rectangle.

It then switches the drag mode from Null to Main\_Can this just tells the program that the object being dragged is on the Main Canvas which covers the whole window. For my system I had 2 options. The gates could spawn on the canvas when the button is pushed and then the user can move to where they wanted or the gate could spawn on their click and they drag it to where they wanted. I liked the second option more because it saves the user from dragging the mouse to the middle of the screen and then going to where they want to place it. With the second option they just need to drag it to where they want. This might save someone half a second but it will save them from getting frustrated if they have to place lots of gates at once.

It then enters a switch case with a string condition. The string is based on the sender tag, as all the rectangular buttons are connected to the same method the switch case decipherers which new gate type should be. As the tags for the rectangles are hard coded the string inputs are enclosed and there should be any worry for an undetected gate type.

At the end it just updates the drag number so that the new gate is known to the system to be moved when the mouse moves.

private void Rect\_Button\_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)

{

if (!Drag && !Link)

{

Drag\_Mode = Drag\_State.Main\_Can;

switch ((sender as Rectangle).Tag)

{

case "And":

Gate\_List.Add(new And\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Nand":

Gate\_List.Add(new Nand\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Not":

Gate\_List.Add(new Not\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Or":

Gate\_List.Add(new Or\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Nor":

Gate\_List.Add(new Nor\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Xor":

Gate\_List.Add(new Xor\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Xnor":

Gate\_List.Add(new Xnor\_Gate\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

case "Transformer":

Gate\_List.Add(new Transformer\_Class(Main\_Canvas, Sub\_Canvas.Scale\_Factor, Output\_Circle\_List, Line\_List, Input\_Button\_List));

break;

}

Drag\_Num = Gate\_List.Count() - 1;

}

}

Main\_Canvas\_MouseLeftButtonUp:

While all this method does is call method Add\_Rect\_Sub\_FIX\_BUG this is due to Canvas class also needs to access the same code. Because XAML eventhandlers are private and have 2 parameters sender and eventarg this meant the canvas class couldn’t access the method. So to fix this I just moved the code to a new method which both classes could access and then called the method in the event arg. While this is a botched solution it’s required as you either have to write out the same code twice or create a new method.

private void Main\_Canvas\_MouseLeftButtonUp(object sender, MouseButtonEventArgs e)

{

if (Drag && !Link)

{

Add\_Rect\_Sub\_FIX\_BUG();

}

}

/// <rectangle added to subcanvas>

/// first check is to make sure it's inside the subcanvas/background\_rect border.

/// then added it to the subcanvas and set the variables.

/// </summary>

public void Add\_Rect\_Sub\_FIX\_BUG()

{

Point Pos\_Rect = Mouse.GetPosition(BackGround\_Rect);

Point Pos\_Border = Mouse.GetPosition(Canvas\_Border);

Point Pos\_Sub = Mouse.GetPosition(Sub\_Canvas);

(Detection\_State State, int ThrowAway) = Sub\_Canvas.Rect\_detection(Gate\_List[Drag\_Num].Rect.Width, Gate\_List[Drag\_Num].Rect.Height, Drag\_Num);

//checks if it's in side the border and if it's inside the area that is allowed in the border.

bool check = (Pos\_Rect.X < 0 || Pos\_Rect.X > 4000 || Pos\_Rect.Y < 0 || Pos\_Rect.Y > 4000) ||

(Pos\_Border.X < 0 || Pos\_Border.X > Canvas\_Border.ActualWidth || Pos\_Border.Y < 0 || Pos\_Border.Y > Canvas\_Border.ActualHeight)

|| (State == Detection\_State.Detected) ? false : true;

Main\_Canvas.Children.Remove(Gate\_List[Drag\_Num].Rect);

Drag\_Mode = Drag\_State.Null;

if (check)

{

Gate\_List[Drag\_Num].Rect.Width = Gate\_List[Drag\_Num].Rect.Width / Sub\_Canvas.Scale\_Factor;

Gate\_List[Drag\_Num].Rect.Height = Gate\_List[Drag\_Num].Rect.Height / Sub\_Canvas.Scale\_Factor;

Sub\_Canvas.Children.Add(Gate\_List[Drag\_Num].Rect);

Gate\_List[Drag\_Num].Rect\_Move(Pos\_Sub);

}

else

{

Gate\_List.RemoveAt(Drag\_Num);

}

}

Link\_Button\_Click:

Just checks to make sure the user isn’t dragging anything or trying to break the system then reverses the mode type. This always reverses it just not’s the current value.

private void Link\_Button\_Click(object sender, RoutedEventArgs e)

{

if (!Drag)

{

Link = !Link;

}

}

Input\_Output\_Button\_Click:

This method is split into to two parts. When the button is active it needs and then pressed it needs to remove the inputs, When the button is disabled it needs to add any existing inputs and fill the rest that are missing some. This is seen in the first if statement.

The first IF statement it goes through the input and output lists and checks the gates that they’re connected to are using the inputs. This is because the input and output buttons can be added then removed then replaced with a line but when added again the line gets priority so they are never added but remain in the list. Why I choose this is because I want the code to run as seeming less as possible and not freeze while it updates the whole data structure of the program just to remove 1 output from the list. If the gate is taking inputs it change the variable in the gate and remove it from canvas. The same happens for the output list.

The second part of the IF statement is a bit more complicated but it’s just to make sure that the existing input and output buttons are added before new ones are added. I wanted this because I think it will be better for the user to be able to keep a certain set up and without it being reset when they didn’t ask it to. It was also needed for saving files. If a user saves a file with input and output it allows it to be opened exactly as they left it.

So a method called Add\_Existing\_IO just goes through the list of inputs and outputs and finds out if the port that they are allocated to is free and that the gate is still alive. If that is true then the existing input/output is added to the canvas and the variables are updated on the gate.

For the new input and outputs, it goes through the list of gates and will first check if it’s on the canvas or not. After that it will do 2 checks for the inputs, these checks are to see if the input port is free and depending on what type of gate it they might only have 1 input port instead of the standard 2.

It’s the same for the outputs but the normal is 1 port but if it’s the transformer gate it will have 3 so there is a check for each of those.

For adding the new input and outputs the code could be slightly neater and robust if you sacrifice memory. Each gate class could hold the information for which gates ports they have. This would result in 1 for loop and 1 generic if statement. This would make it more robust system as if you wanted to add any new gate class it would work without changing this method. But I don’t plan on adding any more gate types as I cover all the ones needed. Also the additional memory that would be needed for every single even though they would just be constants wouldn’t be worth it.

After all the inputs and outputs are added the gate will just calculate their output just in case it’s a Nand gate where 2 zeros will result in a 1.

private void Input\_Output\_Button\_Click(object sender, RoutedEventArgs e)

{

if (IO\_Active)

{

IO\_Active = false;

for (int i = 0; i < Input\_Button\_List.Count; i++)

{

if (Gate\_List[Input\_Button\_List[i].Input\_ID].Input[Input\_Button\_List[i].Input\_Port].Input\_Type == IO\_Type.IO)

{

Gate\_List[Input\_Button\_List[i].Input\_ID].Input[Input\_Button\_List[i].Input\_Port].Input\_Type = IO\_Type.Null;

Sub\_Canvas.Children.Remove(Input\_Button\_List[i]);

}

}

for (int i = 0; i < Output\_Circle\_List.Count; i++)

{

if (Gate\_List[Output\_Circle\_List[i].Output\_ID].Output[Output\_Circle\_List[i].Output\_Port].Output\_Type == IO\_Type.IO)

{

Gate\_List[Output\_Circle\_List[i].Output\_ID].Output[Output\_Circle\_List[i].Output\_Port].Output\_Type = IO\_Type.Null;

Output\_Circle\_List[i].Remove\_UI();

}

}

}

else

{

Add\_IO\_Method();

}

}

/// <Adding IO buttons>

/// This is it's own method as when you load a file this part of the method needs to be called to add the gates.

/// the first part is just checking already existing gates in the input\_button\_List and output\_Circle\_List

/// and adding them to the canvas.

/// The second part is just find out which gate on the screen still doesn't have any IO and adds it.

/// </summary>

private void Add\_IO\_Method()

{

IO\_Active = true;

Add\_Existing\_IO();

for (int i = 0; i < Gate\_List.Count; i++)

{

if (Gate\_List[i].Alive)

{

if (Gate\_List[i].Input[0].Input\_Type == IO\_Type.Null)

{

Input\_Assignment(i, 0);

}

if (Gate\_List[i].Input[1].Input\_Type == IO\_Type.Null && Gate\_List[i].Type != Gate\_Type.Not && Gate\_List[i].Type != Gate\_Type.Transformer)

{

Input\_Assignment(i, 1);

}

if (Gate\_List[i].Output[0].Output\_Type == IO\_Type.Null)

{

Output\_Assignment(i, 0);

}

if (Gate\_List[i].Output[1].Output\_Type == IO\_Type.Null && Gate\_List[i].Type == Gate\_Type.Transformer)

{

Output\_Assignment(i, 1);

}

if (Gate\_List[i].Output[2].Output\_Type == IO\_Type.Null && Gate\_List[i].Type == Gate\_Type.Transformer)

{

Output\_Assignment(i, 2);

}

Gate\_List[i].Gate\_Output\_Calc();

}

}

}

private void Add\_Existing\_IO()

{

for (int i = 0; i < Input\_Button\_List.Count; i++)

{

int ID = Input\_Button\_List[i].Input\_ID;

if (Gate\_List[ID].Input[Input\_Button\_List[i].Input\_Port].Input\_Type == IO\_Type.Null && Gate\_List[ID].Alive)

{

Gate\_List[ID].Input[Input\_Button\_List[i].Input\_Port].Input\_Type = IO\_Type.IO;

Gate\_List[ID].Input[Input\_Button\_List[i].Input\_Port].Input\_ID = i;

Gate\_List[ID].Input[Input\_Button\_List[i].Input\_Port].Input\_bit = Input\_Button\_List[i].Bit;

Sub\_Canvas.Children.Add(Input\_Button\_List[i]);

Input\_Button\_List[i].Align\_Box(Gate\_List[Input\_Button\_List[i].Input\_ID]);

}

}

for (int i = 0; i < Output\_Circle\_List.Count; i++)

{

int ID = Output\_Circle\_List[i].Output\_ID;

if (Gate\_List[ID].Output[Output\_Circle\_List[i].Output\_Port].Output\_Type == IO\_Type.Null && Gate\_List[ID].Alive)

{

Gate\_List[ID].Output[Output\_Circle\_List[i].Output\_Port].Output\_Type = IO\_Type.IO;

Gate\_List[ID].Output[Output\_Circle\_List[i].Output\_Port].Output\_ID = i;

Output\_Circle\_List[i].Add\_UI();

Output\_Circle\_List[i].Align\_Circle(Gate\_List[Output\_Circle\_List[i].Output\_ID]);

}

}

}

/// <summary>

/// Adds a new input\_Button to input\_Button\_List and sets the value of the gate to fit the new values it should have.

/// </summary>

/// <param name="i"></just the position in the list, Gate Num>

/// <param name="Port"></which input slot it needs to be allocated it>

public void Input\_Assignment(int i, int Port)

{

Input\_Button\_List.Add(new Input\_Button(i, Port, Gate\_List, Sub\_Canvas, this));

//can't be in constructor because UI\_Elemtent isn't loaded until the constructor finished.

Input\_Button\_List.Last().Align\_Box(Gate\_List[i]);

Gate\_List[i].Input[Port].Input\_Type = IO\_Type.IO;

Gate\_List[i].Input[Port].Input\_ID = Input\_Button\_List.Count - 1;

}

//Same as input(Above)

public void Output\_Assignment(int i, int Port)

{

Output\_Circle\_List.Add(new Output\_Circle(i, Port, Sub\_Canvas, this));

Output\_Circle\_List.Last().Align\_Circle(Gate\_List[i]);

Gate\_List[i].Output[Port].Output\_Type = IO\_Type.IO;

Gate\_List[i].Output[Port].Output\_ID = Output\_Circle\_List.Count - 1;

}

Link\_Input\_Align:

This is used to tell the lines and input buttons the position they need to go to be in line with their port number.

It’s the same for the output Method

public double[] Link\_Input\_Align(Gate\_Class Gate, int Input\_Num)

{

//not gate, input is in the center of the gate compare to the other gates whic has 2 on the side

if (Gate.Type == Gate\_Type.Not)

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 5, Canvas.GetTop(Gate.Rect) + 38 };

}

//this is similar to the not gate but because it's a sqaure not a rectangle it needed to be moved in the X axis more

else if (Gate.Type == Gate\_Type.Transformer)

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 12.5, Canvas.GetTop(Gate.Rect) + 38 };

}

//the rest all follow the same setup as the and gate

else

{

if (Input\_Num == 0)

{

return new double[] { Canvas.GetLeft(Gate.Rect), Canvas.GetTop(Gate.Rect) + 15 };

}

//else if not needed here but Input\_ID isn't a secure variable type.

else if (Input\_Num == 1)

{

return new double[] { Canvas.GetLeft(Gate.Rect), Canvas.GetTop(Gate.Rect) + 62 };

}

}

return new double[] { -1, -1 };

}

//Same as input

public double[] Link\_Output\_Align(Gate\_Class Gate, int Output\_Num)

{

//special gate class with 3 exit

if (Gate.Type == Gate\_Type.Transformer)

{

if (Output\_Num == 0)

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 75, Canvas.GetTop(Gate.Rect) + 23.8 };

}

else if (Output\_Num == 1)

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 75, Canvas.GetTop(Gate.Rect) + 36 };

}

else if (Output\_Num == 2)

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 75, Canvas.GetTop(Gate.Rect) + 51 };

}

}

//not gate

else if (Gate.Type == Gate\_Type.Not)

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 109.5, Canvas.GetTop(Gate.Rect) + 36 };

}

//every other gate

else

{

return new double[] { Canvas.GetLeft(Gate.Rect) + 115, Canvas.GetTop(Gate.Rect) + 35.7 };

}

return new double[] { -1, -1 };

}

Delay\_Lable\_TextChanged:

This is the event handler for the amount of time that the simulator has to wait before it can carry on working. There were 2 ways of doing this code. I could use try and catch statements or use Uint.TryPhase. I didn’t use TryPhase as even if you inputted a value that isn’t accepted it wouldn’t crash but it would reset the value to the previous accepted value. This is because with tryphase it takes to parameters. The second one is a new unsigned bit variable. If the string is rejected then the new value of the variable is null. This cause lots of problems and then you’re left with another if statement to check that the variable isn’t null. So I choose the try and catch statements as then it only required 1 if statement and no additional variable to store the value.

In the try statement if the value isn’t accepted the value of Delay\_Intervals doesn’t change then then when you change the text of the label it will go back to the last accepted value. This is perfect for what I want as it tells the user straight away that the value isn’t acceptable.

private void Delay\_Lable\_TextChanged(object sender, TextChangedEventArgs e)

{

try

{

if (Delay\_Lable.Text == "")

{

Delay\_Intervals = 0;

Delay\_Lable.Text = "0";

}

Delay\_Intervals = Convert.ToUInt32(Delay\_Lable.Text);

}

catch

{

}

Delay\_Lable.Text = Convert.ToString(Delay\_Intervals);

Delay\_Lable.CaretIndex = Delay\_Lable.Text.Length;

}

Run\_Button\_Click:

This is where the calculations for what the simulator is doing. This is what the user cares about so it needs to not freeze the UI and I didn’t want the simulator to just be just an animation. This button is to start and stop the simulator at any time. Just like every other button it’s based on a Boolean variable that is either active or not. If it’s active it will just change the Simulator running variable to false. Because of how background workers work this also updates the variable in the thread. This will be clearer in the method for the simulator. The button content also changes to fit the new state.

If the simulator is not running and is not busy then the background worker can start up again. The reason behind needing 2 variables to be able to tell what is happening is due to the simulator wait time. Because there is a wait between each cycle of the simulator the user could press the button to stop it and changing the variable to false, then pressing it again to activate the simulator again while the other one is still active. Also due to the variable now being true again the original simulator wouldn’t close down. While this wouldn’t cause a bug or break anything this isn’t an acquirable solution. So there is 2 variables, one to tell the simulator to end the loop and close down, and another to let the program know that the background worker has closed. This is important that both ends have their own Boolean variable so that each can confirm with each other on the state of the program.

private void Run\_Button\_Click(object sender, RoutedEventArgs e)

{

if (Sim\_Running)

{

Sim\_Running = false;

Run\_Button.Content = "Run";

}

else if(!Sim\_Running && !Sim\_Busy)

{

Sim\_Busy = true;

Sim\_Running = true;

Run\_Button.Content = "Stop";

Clean\_Up\_Method();

Add\_IO\_Method();

Simulator\_Worker.RunWorkerAsync();

}

}

Simulator\_Work:

This is the actually code that the background worker will do. It first finds all the active starting points for the code. This is important as you can have multiple branches of circuits that aren’t connected but you want them all to run. It’s also important to remove any duplicating Start points. This is because it will be possible to have 2 starting gates together and if the ordering was to be first, second and first again you could override one of the steps in the simulation. It’s also better for efficiency.

It then enters a while loop with 2 conditions. It will either run out of nodes and then then it will exit it or the user can stop the simulator and it will exit.

It then cycles through the active gates by calculating the new output and changing the colour to red to help the user see what is happening. It will also go through all the outputs of the gate and see if it’s linked to another gate. If it is then will add the gate ID to the next gate list and update its input. The code also includes await dispatcher to change the window UI. This is because the worker thread is not the parent thread so an invoke has to happen.

When it’s gone through all the gates it will remove all the duplicated next gate ID. Waits the designated time set by the user. If this is zero then the UI objects will still update even without a task delay and it will cause a horrible effect if the circuit is in a loop but otherwise it will give you an output instantly. After the delay the UI will all be set back to black. The active gate list will equal the next gate list and a check will be done to see if there is still gates left to be calculated.

When it exits the while loop the content of the button will be changed to Run and sim busy will be false as the worker will close when it reaches the end of the block of code.

private async void Simulator\_Work(object sender, DoWorkEventArgs e)

{

List<int> Active\_Gates = Set\_Up\_Start\_Sim();

while (Sim\_Running)

{

List<int> Next\_Gate = new List<int>();

for (int i = 0; i < Active\_Gates.Count; i++)

{

Gate\_List[Active\_Gates[i]].Gate\_Output\_Calc();

await Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Gate\_List[Active\_Gates[i]].Rect.Stroke = Brushes.Red));

for (int x = 0; x < 3; x++)

{

if (Gate\_List[Active\_Gates[i]].Output[x].Output\_Type == IO\_Type.Gate)

{

await Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Line\_List[Gate\_List[Active\_Gates[i]].Output[x].Line\_ID].Change\_UI\_Red()));

Gate\_List[Line\_List[Gate\_List[Active\_Gates[i]].Output[x].Line\_ID].Input\_ID].Input[Line\_List[Gate\_List[Active\_Gates[i]].Output[x].Line\_ID].Input\_Num].Input\_bit = Gate\_List[Active\_Gates[i]].Gate\_Bit;

Next\_Gate.Add(Line\_List[Gate\_List[Active\_Gates[i]].Output[x].Line\_ID].Input\_ID);

}

}

}

Next\_Gate = Remove\_Duplicats(Next\_Gate);

await Task.Delay(Convert.ToInt32(Delay\_Intervals));

Set\_Input\_Back(Active\_Gates);

Active\_Gates = Next\_Gate.ToList();

Next\_Gate.RemoveRange(0, Next\_Gate.Count);

if (Active\_Gates.Count == 0)

{

Sim\_Running = false;

}

}

await Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Run\_Button.Content = "Run"));

Sim\_Busy = false;

}

private List<int> Set\_Up\_Start\_Sim()

{

List<int> Active\_Gates = new List<int>();

for (int i = 0; i < Input\_Button\_List.Count; i++)

{

if (Gate\_List[Input\_Button\_List[i].Input\_ID].Alive)

{

Active\_Gates.Add(Input\_Button\_List[i].Input\_ID);

}

}

for (int i = 0; i < Active\_Gates.Count - 1; i++)

{

bool OverLap = false;

for (int x = i + 1; x < Active\_Gates.Count; x++)

{

if (Active\_Gates[x] == Active\_Gates[i])

{

OverLap = true;

}

}

if (OverLap)

{

Active\_Gates.RemoveAt(i);

i -= 1;

}

}

return Active\_Gates;

}

private List<int> Remove\_Duplicats(List<int> Next\_Gate)

{

for (int i = 0; i < Next\_Gate.Count - 1; i++)

{

bool OverLap = false;

for (int x = i + 1; x < Next\_Gate.Count; x++)

{

if (Next\_Gate[x] == Next\_Gate[i])

{

OverLap = true;

}

}

if (OverLap)

{

Next\_Gate.RemoveAt(i);

i -= 1;

}

}

return Next\_Gate;

}

private async void Set\_Input\_Back(List<int> Active\_Gates)

{

for (int i = 0; i < Active\_Gates.Count; i++)

{

await Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Gate\_List[Active\_Gates[i]].Rect.Stroke = Brushes.Black));

for (int x = 0; x < 3; x++)

{

if (Gate\_List[Active\_Gates[i]].Output[x].Output\_Type == IO\_Type.Gate)

{

await Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Line\_List[Gate\_List[Active\_Gates[i]].Output[x].Line\_ID].Change\_UI\_Black()));

}

}

}

}

Clean\_Up\_Method:

This is the method that is used to start the clean-up of the unused data and variables in the program.

The method is in the IF statement because if it’s running and you do a clean-up between the wait and start of a new loop then there is a potential for the gates to become out of line with the list of active nodes. So I thought it would just be easier to remove that option when it’s running. When you are removing data from the list everything becomes out of order with the links If you were to add a new gate and a new link during this process of everything being reorganized you have a possibility to get an out of range error. This is why I have the progress window and it’s running a showdialog. The ShowDialog command just means that the new window halts the mainwindow thread until it is closed. This is why the background worker is there. It creates a new thread and does the work of the reorganizing. Its thread doesn’t get halted by the showdialog so when it’s finished its work it will remove the progress window and then unfreezing the MainWindow thread.

private void System\_Clean\_Up(object sender, RoutedEventArgs e)

{

Clean\_Up\_Method();

}

/// <summary>

/// Creates a new window UI that hold the progress bar.

/// Starts the background worker to clean up the system.

/// The MainWindow is holded untill the system clean up is complete.

/// </summary>

public void Clean\_Up\_Method()

{

if (!Sim\_Busy)

{

Progress\_Window = new Progress\_Bar\_Window(Gate\_List.Count());

\_worker.RunWorkerAsync();

Progress\_Window.ShowDialog();

}

}

/// <summary>

/// goes through all the list in the program and removes unused objects and resorts the lists to work with the change.

/// </summary>

private void WorkerDoWork(object sender, DoWorkEventArgs e)

{

//remove unused input and output

Remove\_Unsed\_Output();

Remove\_Unsed\_Input();

Remove\_Unsed\_Line();

Remove\_Unsed\_Gate();

}

Remove\_Unsed\_X:

The X just means that it’s for the different type of methods all with roughly the same name. Each of the methods goes through the list of their type and removes any unused variables. It will also readjust the variables that they linked to.

private void Remove\_Unsed\_Output()

{

for (int i = 0; i < Output\_Circle\_List.Count; i++)

{

if (Gate\_List[Output\_Circle\_List[i].Output\_ID].Output[Output\_Circle\_List[i].Output\_Port].Output\_Type == IO\_Type.Gate || Gate\_List[Output\_Circle\_List[i].Output\_ID].Alive == false)

{

for (int x = 0; x < Gate\_List.Count; x++)

{

if (Gate\_List[x].Output[Output\_Circle\_List[i].Output\_Port].Output\_ID > i && Gate\_List[x].Output[Output\_Circle\_List[i].Output\_Port].Output\_Type == IO\_Type.IO)

{

Gate\_List[x].Output[Output\_Circle\_List[i].Output\_Port].Output\_ID -= 1;

}

}

Output\_Circle\_List.RemoveAt(i);

i -= 1;

}

}

}

private void Remove\_Unsed\_Input()

{

for (int i = 0; i < Input\_Button\_List.Count; i++)

{

if (!Gate\_List[Input\_Button\_List[i].Input\_ID].Alive)

{

for (int x = 0; x < Gate\_List.Count; x++)

{

if (Gate\_List[x].Input[Input\_Button\_List[i].Input\_Port].Input\_ID > i && Gate\_List[x].Input[Input\_Button\_List[i].Input\_Port].Input\_Type == IO\_Type.IO)

{

Gate\_List[x].Input[Input\_Button\_List[i].Input\_Port].Input\_ID -= 1;

}

}

Input\_Button\_List.RemoveAt(i);

i -= 1;

}

}

}

private void Remove\_Unsed\_Line()

{

for (int i = 0; i < Line\_List.Count; i++)

{

if (Gate\_List[Line\_List[i].Input\_ID].Input[Line\_List[i].Input\_Num].Input\_Type != IO\_Type.Gate || Gate\_List[Line\_List[i].Input\_ID].Alive == false)

{

for (int x = 0; x < Gate\_List.Count; x++)

{

for (int y = 0; y < 3; y++)

{

if (Gate\_List[x].Output[y].Line\_ID > i)

{

Gate\_List[x].Output[y].Line\_ID -= 1;

}

}

for (int z = 0; z < 2; z++)

{

if (Gate\_List[x].Input[z].Line\_ID > i)

{

Gate\_List[x].Input[z].Line\_ID -= 1;

}

}

}

Line\_List.RemoveAt(i);

i -= 1;

}

}

}

private void Remove\_Unsed\_Gate()

{

for (int i = 0; i < Gate\_List.Count; i++)

{

if(!Gate\_List[i].Alive)

{

for (int x = 0; x < Gate\_List.Count; x++)

{

for (int z = 0; z < 3; z++)

{

if (Gate\_List[x].Output[z].Output\_Type==IO\_Type.Gate && Gate\_List[x].Output[z].Output\_ID>i)

{

Gate\_List[x].Output[z].Output\_ID -= 1;

}

}

for (int z = 0; z < 2; z++)

{

if (Gate\_List[x].Input[z].Input\_Type == IO\_Type.Gate && Gate\_List[x].Input[z].Input\_ID > i)

{

Gate\_List[x].Input[z].Input\_ID -= 1;

}

}

}

for (int z = 0; z < Input\_Button\_List.Count; z++)

{

if(Input\_Button\_List[z].Input\_ID>i)

{

Input\_Button\_List[z].Input\_ID -= 1;

}

}

for (int z = 0; z < Output\_Circle\_List.Count; z++)

{

if(Output\_Circle\_List[z].Output\_ID >i)

{

Output\_Circle\_List[z].Output\_ID -= 1;

}

}

for (int z = 0; z < Line\_List.Count; z++)

{

if(Line\_List[z].Output\_ID>i)

{

Line\_List[z].Output\_ID -= 1;

}

if(Line\_List[z].Input\_ID>i)

{

Line\_List[z].Input\_ID -= 1;

}

}

if(Sub\_Canvas.Last\_Rect>i)

{

Sub\_Canvas.Last\_Rect -= 1;

}

Gate\_List.RemoveAt(i);

i -= 1;

}

}

}

MenuItem\_New\_X:

For each of the menu methods they are linked to the methods that are found in the File\_Creation class. This is because mainwindow class was doing too much and it didn’t link directly to what MainWindow is trying to achieve.

private void WorkerRunWorkerCompleted(object sender, RunWorkerCompletedEventArgs e)

{

Progress\_Window.Close();

}

private void MenuItem\_Load\_Click(object sender, RoutedEventArgs e)

{

File\_Worker.MenuItem\_Load\_Click\_Method();

}

private void MenuItem\_SaveAs\_Click(object sender, RoutedEventArgs e)

{

File\_Worker.MenuItem\_SaveAs\_Click\_Method();

}

private void MenuItem\_Save\_Click(object sender, RoutedEventArgs e)

{

File\_Worker.MenuItem\_Save\_Click\_Method();

}

private void MenuItem\_New\_Click(object sender, RoutedEventArgs e)

{

File\_Worker.MenuItem\_New\_Click\_Method();

}

private void MenuItem\_Exit\_Click(object sender, RoutedEventArgs e)

{

Environment.Exit(0);

}

Reset\_Program:

For resetting the program I had 3 options. I could reload the whole window and just remove all data, I could make MainWindow equal a new MainWindow or change the variables of MainWindow. I didn’t want to reload the MainWindow as that would take a couple of seconds for the window to be rendered again. Just making the MainWindow equal a new mainwindow would reset everything without the lag but it would also reset variables like the time delay which the user would think wouldn’t change. So I choose just to reset the variables that need to be reset in this function. This gives the quickest solution and also most desirable for the user.

public void Reset\_Program()

{

Gate\_List = new List<Gate\_Class>();

Line\_List = new List<Line\_Class>();

Input\_Button\_List = new List<Input\_Button>();

Output\_Circle\_List = new List<Output\_Circle>();

Drag = false;

Delay\_Intervals = 1;

Drag\_Num = 0;

\_link = false;

Sim\_Running = false;

Saved = false;

File\_Location = "";

drag\_mode = Drag\_State.Null;

Linking\_ID = 0;

IO\_Active = false;

Sub\_Canvas = new Canvas\_Class(this, ref Canvas\_Border, this);

SetUp\_Canvas();

}

# Canvas Class

Structure of Class and variables:

Canvas Class is the custom UI class that the user places all their objects on. It can do this because it inherits the class from the WPF Canvas. This gives the class access to custom event handlers and features that can directly change the way it is presented. The variables that don’t overlap with mainwindow are declared in the class and the ones that do are sent over as an attribute under the interface Canvas\_Variables. I’m also sending the mainwindow over as canvas class needs to access methods held in there. This isn’t a huge problem as it’s only got a getter and no setter so there is no chance for accidental changes in the code.

In the constructor the attributes are linked to the variables of the class. Then the canvas is set up. The first part just makes the background grey. This servers 2 purposes, the first is to let the user know when they are out of bounds from the canvas, the second is due with how the detection works with WPF. When there is a colour change in the UI on the mouse cursor the UI will fire MouseMove events. It’s really weird why this happens but a simple fix is just to change the colour of 2 objects by 1 bit colour or just do a completely different colour.

The second part is the rendering of the canvas. It has the scaler and translation. The scaler is used when the user zooms in and out and that works from a point. The translation is for when the user drags the canvas across the screen. These are both added to a transformation group so that they work together and don’t over right each other.

public class Canvas\_Class : Canvas

{

//variabls for the class

public Point Old\_Pos { get; set; } = new Point();

public double Scale\_Factor { get; set; } = 1;

public Point Old\_Rect { get; set; } = new Point();

public TranslateTransform Translate\_Action { get; set; }

public ScaleTransform Scale\_Action { get; set; }

public TransformGroup Transforms\_Group { get; set; }

public bool MovingCanvas { get; set; } = false;

public Border \_Canvas\_Border { get; set; }

public int Last\_Rect { get; set; } = -1;

public Canvas\_Variables variables;

private MainWindow \_MainWind { get; }

/// <summary>

/// The constructor just sets the value for the canvas and adds the transformation set up.

/// </summary>

/// <param name="Variables"></This a more secure way of accessing and transfering varaibles in a bidirectional way>

/// <param name="Canvas\_Border"></Needed as a reference point for mouse postion>

/// <param name="MainWind"></Needed to call methods in the class>

public Canvas\_Class( Canvas\_Variables Variables, ref Border Canvas\_Border, MainWindow MainWind)

{

variables = Variables;

\_Canvas\_Border = Canvas\_Border;

\_MainWind = MainWind;

Background = Brushes.Gray;

Translate\_Action = new TranslateTransform(0, 0);

Scale\_Action = new ScaleTransform(1, 1, 0, 0);

Transforms\_Group = new TransformGroup();

Transforms\_Group.Children.Add(Translate\_Action);

Transforms\_Group.Children.Add(Scale\_Action);

RenderTransform = Transforms\_Group;

}

OnMouseMove:

In mouse move event 2 things need to happen, if it’s over a gate that gate border should highlight to let the user know that they can interact with it and if they are moving the canvas then the new position for the canvas should be used.

With the rectangle lighting up it need to change the old one to black this is why there is the IF statement.

For the moving of canvas it’s based on the last known position of the mouse. It works out the difference and just translate it by that much.

protected override void OnMouseMove(MouseEventArgs e)

{

Detection\_State State;

int ID;

(State, ID) = Rect\_detection(0, 0, -1);

if (State == Detection\_State.Detected)

{

if (Last\_Rect != -1)

{

variables.Gate\_List[Last\_Rect].Rect.Stroke = Brushes.Black;

}

variables.Gate\_List[ID].Rect.Stroke = Brushes.LightGreen;

Last\_Rect = ID;

}

if (MovingCanvas)

{

Translate\_Action.X += (e.GetPosition(this).X - Old\_Pos.X);

Translate\_Action.Y += (e.GetPosition(this).Y - Old\_Pos.Y);

}

Old\_Pos = e.GetPosition(this);

}

OnMouseWheel:

For the zooming of the canvas I had to use a number which could be done fully with a floating point number. That’s why I’m using 2^-4 as my scaling number. This is because any multiple of this number can be used as floating point and a rounding error won’t occur.

There is a Boolean variable just to make sure there was a change, this is because although it won’t change the scale of the canvas the middle with be repositioned as for the scale factor method I’m doing the where it’s from the middle of the cursor.

protected override void OnMouseWheel(MouseWheelEventArgs e)

{

bool change = false;

Point Pos = Mouse.GetPosition(\_Canvas\_Border);

if (e.Delta > 0)

{

if (Scale\_Factor != 0.0625)

{

Scale\_Factor -= 0.0625;

change = true;

}

}

else if (e.Delta < 0)

{

if (Scale\_Factor != 2)

{

Scale\_Factor += 0.0625;

change = true;

}

}

if (change)

{

Scale\_Action.ScaleX = Scale\_Factor;

Scale\_Action.ScaleY = Scale\_Factor;

Scale\_Action.CenterX = Pos.X;

Scale\_Action.CenterY = Pos.Y;

}

}

OnMouseLeftButtonDown:

3 things can happen when you click down, activate canvas moving, start to move a gate or a line will be added and start to be dragged.

For moving the canvas you just need to be clicking on nothing and can’t be dragging anything. If this is true then it will change the variable for dragging canvas and the old position will be recorded so that it is the most up-to-date position of the canvas.

If it’s moving a gate then you need to click a gate, not be in link mode and not currently dragging anything. This will update the drag number, change the drag mode and store the old position of the rectangle. This is because if you fail to move it to a new location then it will be sent back to the old position.

For adding a new line all the same conditions apply but it should be in link mode. It will then add a new line and check that the gate that is selected has a free output.

protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)

{

(Detection\_State State, int detected) = Rect\_detection(0, 0, -1);

if(!variables.Drag &&State == Detection\_State.Null)

{

Old\_Pos = e.GetPosition(this);

MovingCanvas = true;

}

if (!variables.Drag && !variables.Link && State == Detection\_State.Detected)

{

variables.Drag\_Num = detected;

variables.Drag\_Mode = Drag\_State.Sub\_Can;

Old\_Rect = new Point(Convert.ToDouble(Canvas.GetLeft(variables.Gate\_List[variables.Drag\_Num].Rect)), Convert.ToDouble(Canvas.GetTop(variables.Gate\_List[variables.Drag\_Num].Rect)));

}

else if (!variables.Drag && variables.Link && State == Detection\_State.Detected)

{

Line\_M1\_Down(detected);

}

}

private void Line\_M1\_Down(int detected)

{

variables.Linking\_ID = detected;

variables.Drag\_Num = variables.Line\_List.Count();

variables.Line\_List.Add(new Line\_Class(detected, this, \_MainWind, detected, true));

//if X == -2 then there was no aviable output for the new link to be made(already deleted the last line)

if (variables.Line\_List.Last().Output\_Num != -2)

{

variables.Drag\_Mode = Drag\_State.Link\_Mode\_Sub;

variables.Gate\_List[detected].Output[variables.Line\_List.Last().Output\_Num].Line\_ID = variables.Drag\_Num;

variables.Line\_List[variables.Drag\_Num].Link\_Output\_Align\_Line(variables.Gate\_List[detected]);

}

else

{

variables.Line\_List[variables.Drag\_Num].Remove\_Class();

}

}

OnMouseLeftButtonUp:

This does the opposite of mouse down. It will deactivate all current processes that are taken place.

The first IF statement is used to just fix a weird bug that would occur due to the ordering of the event handlers. Normally the event handler should be based on when they were added. As mainwindow left mouse click is added before the sub canvas left mouse click the mainwindow event handlers should get priority. But due to how mainwindow area overlaps with subcanvas this doesn’t always happens. So I’ve just got an IF condition so that it will redirect it to the MainWindow function in this circumstance.

The second IF statement if you are dragging a rectangle. If the mouse is over a rectangle and there is overlap then the rectangle will move back to the old position otherwise it will be dropped where it is.

The third IF statement is for new lines. It will work out what input port it should go to base on the user. If it is full then it will try and find the next available port. If there is no available ports then the line will be removed. Otherwise it will change all the variables needed to know what the new links are between gates and lines.

protected override void OnMouseLeftButtonUp(MouseButtonEventArgs e)

{

MovingCanvas = false;

if(variables.Drag && !variables.Link && variables.Drag\_Mode == Drag\_State.Main\_Can)

{

\_MainWind.Add\_Rect\_Sub\_FIX\_BUG();

}

if (variables.Drag && !variables.Link)

{

variables.Drag\_Mode = Drag\_State.Null;

(Detection\_State State, int Null) = Rect\_detection(variables.Gate\_List[variables.Drag\_Num].Rect.Width, variables.Gate\_List[variables.Drag\_Num].Rect.Height, variables.Drag\_Num);

if (State == Detection\_State.Detected)

{

variables.Gate\_List[variables.Drag\_Num].Rect\_Move(Old\_Rect);

variables.Gate\_List[variables.Drag\_Num].Move\_IO();

}

}

else if(variables.Drag && variables.Link)

{

Line\_M1\_Up();

}

}

private void Line\_M1\_Up()

{

int X = -1;

(Detection\_State State, int detection) = Rect\_detection(0, 0, -1);

variables.Drag\_Mode = Drag\_State.Null;

if (State == Detection\_State.Detected && detection != variables.Linking\_ID)

{

X = Link\_Input\_Vaildation(detection);

}

if (X != -1)

{

//This a block of code that is only done one time and isn't assocaited with each other so making a method wouldn't make a lot of sense but it's a lot of just nothing.

variables.Line\_List[variables.Drag\_Num].Input\_ID = detection;

variables.Line\_List[variables.Drag\_Num].Input\_Num = X;

variables.Line\_List[variables.Drag\_Num].Link\_Input\_Align\_Line(variables.Gate\_List[detection]);

variables.Gate\_List[detection].Input[X].Input\_Type = IO\_Type.Gate;

variables.Gate\_List[detection].Input[X].Input\_ID = variables.Linking\_ID;

variables.Gate\_List[detection].Input[X].Line\_ID = variables.Drag\_Num;

variables.Gate\_List[variables.Linking\_ID].Output[variables.Line\_List.Last().Output\_Num].Output\_ID = detection;

variables.Gate\_List[variables.Linking\_ID].Output[variables.Line\_List.Last().Output\_Num].Output\_Type = IO\_Type.Gate;

}

else if (X == -1)

{

variables.Line\_List[variables.Drag\_Num].Remove\_Class();

}

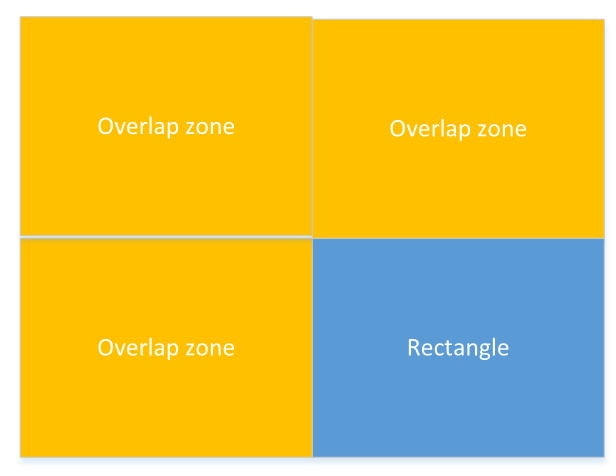
}

Rect\_detection:

This method just goes through the whole gate list in a linear search and tests the ranges of the boundaries and finds out if there is an overlap. It’s adjustable for different types of gates and different areas that need to be scanned. This is desirable because it means the function can be used for if you want to find if the cursor is over a gate or if the gate that is being added to the canvas is overlapping with another gate.

The method returns a tuple of an Enum and int. The Enum could be a Boolean expression just saying if it detected an overlap or not but it was easier to understand if it was an Enum. The int is the ID of the rectangle that was detected. This is -1 by default.

The search to detect if there is an overlap is based on the 2 top left corners of the different points. If the cursor is in the orange/blue area then there is an overlap.



The orange zone will only be there if you are testing an object overlapping. There will be no orange for testing the cursor. It does this by testing the top left corner for both the object and gate.

public (Detection\_State, int) Rect\_detection(double Width, double Height,int Drag\_Num)

{

Detection\_State State = Detection\_State.Null;

int Detected = -1;

Point Pos\_Sub = Mouse.GetPosition(this);

for (int i = 0; i < variables.Gate\_List.Count(); i++)

{

Gate\_Class Rect = variables.Gate\_List[i];

double Rect\_X = GetLeft(variables.Gate\_List[i].Rect);

double Rect\_Y = GetTop(variables.Gate\_List[i].Rect);

if (Pos\_Sub.X + Width > Rect\_X && Pos\_Sub.X < Rect\_X + Rect.Rect.Width && Pos\_Sub.Y + Height > Rect\_Y && Pos\_Sub.Y < Rect\_Y + Rect.Rect.Height && i != Drag\_Num && Rect.Alive)

{

Detected = i;

State = Detection\_State.Detected;

}

}

return (State, Detected);

}

OnMouseRightButtonUp:

For the right button it only needs 1 activation and then the event happens. This is on the button being released as it means if you accidently push it down you can move it off the object that you don’t want to remove.

It only has to 2 purposes, to remove a gate or to remove a line.

To remove an object the program shouldn’t be dragging anything and it needs to be on a gate.

If it’s removing a gate it will change the alive state of the gate to false and if it was the last rectangle that was touched by the cursor (Which it should be but not guaranteed) then the variable will change to -1. The rectangle will be removed from the canvas but not from the list, this is because I don’t want the UI to freeze while it resorts all the data.

It will then enter 2 for loops, the first is for the outputs of the gate. This will check if the output type was an output or a connection. If it’s an output then it will just remove the output circle from the canvas. If it’s a connection then it will remove the line from the canvas and reset the values of the input that it was connected too to null. For the second for loop it will do the same but for the inputs.

If it’s removing a line from the gate it will always be the output line. If it’s the transformer gate type then it needs to work out which to remove based on the position of the cursor. Once that is determined then it will reset the output port and the corresponding input port to null.

protected override void OnMouseRightButtonUp(MouseButtonEventArgs e)

{

(Detection\_State State, int detection) = Rect\_detection(0, 0, -1);

//checks you're not dragging and that you clicked on a gate.

if (!variables.Drag && State == Detection\_State.Detected)

{

//see which mode of the program you're in because each one will have a different action

if (!variables.Link)

{ //completely romoves the gate.

Remove\_Gate(detection);

}

else if (variables.Link) //if in linked mode then it should act as tho you're doing the oppersite of adding a connection.

{

Remove\_Line(detection);

}

}

}

private void Remove\_Gate(int detection)

{

variables.Gate\_List[detection].Alive = false;

if(detection == Last\_Rect)

{

Last\_Rect = -1;

}

Children.Remove(variables.Gate\_List[detection].Rect);

for (int i = 0; i < 3; i++) //this is for each output of the gate after it's been removed

{

if (variables.Gate\_List[detection].Output[i].Output\_Type == IO\_Type.Gate)

{ //removes the line "connecting" the 2 gates.

variables.Line\_List[variables.Gate\_List[detection].Output[i].Line\_ID].Remove\_UI();

for (int x = 0; x < 2; x++) // this is to determin which input the gate is connected to

{

if (variables.Gate\_List[variables.Gate\_List[detection].Output[i].Output\_ID].Input[x].Input\_ID == detection && variables.Gate\_List[variables.Gate\_List[detection].Output[i].Output\_ID].Input[x].Input\_Type == IO\_Type.Gate)

{ //changes the state of the gate it's connected to too null so that it can accept another input.

variables.Gate\_List[variables.Gate\_List[detection].Output[i].Output\_ID].Input[x].Input\_Type = IO\_Type.Null;

}

}

variables.Gate\_List[detection].Output[i].Output\_Type = IO\_Type.Null;

}

else if (variables.Gate\_List[detection].Output[i].Output\_Type == IO\_Type.IO)

{

variables.Output\_Circle\_List[variables.Gate\_List[detection].Output[i].Output\_ID].Remove\_UI();

//remove the ellipses from the canvas in the output\_Circle list with the ID in the

}

}

for (int i = 0; i < 2; i++) //does the same but for input

{

if (variables.Gate\_List[detection].Input[i].Input\_Type == IO\_Type.Gate)

{

variables.Line\_List[variables.Gate\_List[detection].Input[i].Line\_ID].Remove\_UI();

for (int x = 0; x < 3; x++)

{

if (variables.Gate\_List[variables.Gate\_List[detection].Input[i].Input\_ID].Output[x].Output\_ID == detection)

{

variables.Gate\_List[variables.Gate\_List[detection].Input[i].Input\_ID].Output[x].Output\_Type = IO\_Type.Null;

}

}

}

else if (variables.Gate\_List[detection].Input[i].Input\_Type == IO\_Type.IO)

{

Children.Remove(variables.Input\_Button\_List[variables.Gate\_List[detection].Input[i].Input\_ID]);

}

}

}

private void Remove\_Line(int detection)

{

int Output\_Num = Output\_Slot(detection);

if (variables.Gate\_List[detection].Output[Output\_Num].Output\_ID != -1 && variables.Gate\_List[detection].Output[Output\_Num].Output\_Type == IO\_Type.Gate)

{

for (int i = 0; i < 2; i++)

{

if (variables.Gate\_List[variables.Gate\_List[detection].Output[Output\_Num].Output\_ID].Input[i].Input\_ID == detection)

{

variables.Gate\_List[variables.Gate\_List[detection].Output[Output\_Num].Output\_ID].Input[i].Input\_Type = IO\_Type.Null;

}

}

variables.Line\_List[variables.Gate\_List[detection].Output[Output\_Num].Line\_ID].Remove\_UI();

variables.Gate\_List[detection].Output[Output\_Num].Output\_Type = IO\_Type.Null;

}

}

# Canvas Interface

While I cover above that you need C# 8.0 to get private variables in the interface you also need C#8.0 to store methods. This is the Add\_Rect\_Sub\_FIX\_BUG method due to it being required in canvas class. This is why canvas class requires both the interface and the whole MainWindow Class to be sent as attributes. To minimize the use of the MainWindow variable it only has a setter and is used just to call that method.

The main reason for using the interface to link between the 2 big classes is due to the overlapping variables and functions. Have a connection like this means that the canvas class can be more independent in the sense that if you took the class to another program you would only need to have the interface integrated in to have a working solution.

The variables that it holds is just program states and the list of the UI objects types.

public interface Canvas\_Variables

{

List<Gate\_Class> Gate\_List { get; set; }

List<Line\_Class> Line\_List { get; set; }

List<Input\_Button> Input\_Button\_List { get; set; }

List<Output\_Circle> Output\_Circle\_List { get; set; }

bool Drag { get; set; }

bool Link { get; set; }

int Drag\_Num { get; set; }

Drag\_State Drag\_Mode { get; set; }

int Linking\_ID { get; set; }

}

# File Creation Class and File Classes

File Creation Class:

This class has all the methods for the menu item option. It has Save as, Save, Load and new file. The only variable it needs is the MainWindow. While the mainwindow and file creation class are linked closely together neither can inherit the other as they both depend on each other. This is why there is an association of MainWindow in the class. No additional variables are required for the class other than the parameters for each method.

MainWindow MainWind { get; }

public File\_Creation\_Class(MainWindow \_MainWind)

{

MainWind = \_MainWind;

}

MenuItem\_Load\_Click\_Method:

Depending on if the current work has been saved or not will determine what happens in the method. If the work has been saved then it will tell you that it’s saved then resets the program and then open the load menu. The load menu is just the openfiledialog that windows supply.

If the work hasn’t already been saved then it will tell you that your work will be deleted if you carry on. If they don’t want it deleted then it will cancel the open file method by changing a Boolean variable. If they do want to delete their work then it will reset the program without saving it.

When that all happens then it will open the openfiledialog window where the user can pick the file they want to test. Once they have picked the file they want to open it will try and read it. The whole opening part of the program is in a try and catch state. This is so that if they open the wrong file they will get an error message.

If the file is accepted then it will be sent to another method that will unload the file. This is basically just decompiling the file class which can be serialized back into the class that the program uses. The order in which the decompile works is important. Because of classes depend on others only certain variables can be set at a time. It’s really long lists of variables being set and isn’t pleasant to look at but it’s the only way. The compiler of the class can’t be changed just to fit the unload needs and creating a new function doesn’t server much purpose as this set of code is only used here (only reason would be to make it easier to look at and break down each part into smaller chunks but I don’t mind myself because nothing of value is to be looked at here other than setting variables).

public void MenuItem\_Load\_Click\_Method()

{

bool CarryOn = true;

if (MainWind.File\_Location != "")

{

Save\_File();

MainWind.Reset\_Program();

var result = MessageBox.Show("Your work has been saved", "New File", MessageBoxButton.OK);

}

else

{

var result = MessageBox.Show("You haven't saved your work, Do you want to delete it?", "New File", MessageBoxButton.OKCancel);

if (result == MessageBoxResult.OK)

{

MainWind.Reset\_Program();

}

else

{

CarryOn = false;

}

}

if (CarryOn)

{

OpenFileDialog openFileDialog = new OpenFileDialog();

if (openFileDialog.ShowDialog() == true)

{

openFileDialog.InitialDirectory = @"C:\Documents";

MainWind.File\_Location = openFileDialog.FileName;

Stream stream = new FileStream(MainWind.File\_Location, FileMode.Open, FileAccess.Read);

IFormatter formatter = new BinaryFormatter();

try

{

File\_Class Loaded\_File = (File\_Class)formatter.Deserialize(stream);

File\_Unload(Loaded\_File);

}

catch

{

MessageBox.Show("The file you tried to open doesn't work with this program", "Failed", MessageBoxButton.OK);

}

}

}

}

public void File\_Unload(File\_Class Loaded\_File)

{

for (int x = 0; x < Loaded\_File.Inputs.Count; x++)

{

MainWind.Input\_Button\_List.Add(new Input\_Button(Loaded\_File.Inputs[x].Input\_ID, Loaded\_File.Inputs[x].Input\_Port, MainWind.Gate\_List, MainWind.Sub\_Canvas, MainWind));

MainWind.Input\_Button\_List.Last().Change\_X\_Y(Loaded\_File.Inputs[x].X, Loaded\_File.Inputs[x].Y);

}

for (int x = 0; x < Loaded\_File.Output.Count; x++)

{

MainWind.Output\_Circle\_List.Add(new Output\_Circle(Loaded\_File.Output[x].Output\_ID, Loaded\_File.Output[x].Output\_Port, MainWind.Sub\_Canvas, MainWind));

MainWind.Output\_Circle\_List.Last().Change\_X\_Y(Loaded\_File.Output[x].X, Loaded\_File.Output[x].Y);

}

for (int x = 0; x < Loaded\_File.Lines.Count; x++)

{

MainWind.Line\_List.Add(new Line\_Class(Loaded\_File.Lines[x].Output\_ID, MainWind.Sub\_Canvas, MainWind, Loaded\_File.Lines[x].Input\_ID, false));

MainWind.Line\_List.Last().Output\_Num = Loaded\_File.Lines[x].Output\_Num;

MainWind.Line\_List.Last().Input\_Num = Loaded\_File.Lines[x].Input\_Num;

MainWind.Line\_List.Last().Line\_Lable.Content = Loaded\_File.Lines[x].Content\_Copy;

MainWind.Line\_List.Last().UI\_Line.X1 = Loaded\_File.Lines[x].X1;

MainWind.Line\_List.Last().UI\_Line.X2 = Loaded\_File.Lines[x].X2;

MainWind.Line\_List.Last().UI\_Line.Y1 = Loaded\_File.Lines[x].Y1;

MainWind.Line\_List.Last().UI\_Line.Y2 = Loaded\_File.Lines[x].Y2;

MainWind.Line\_List.Last().X1 = Loaded\_File.Lines[x].X1;

MainWind.Line\_List.Last().X2 = Loaded\_File.Lines[x].X2;

MainWind.Line\_List.Last().Y1 = Loaded\_File.Lines[x].Y1;

MainWind.Line\_List.Last().Y2 = Loaded\_File.Lines[x].Y2;

MainWind.Line\_List.Last().UI\_Line.Stroke = Brushes.Black;

MainWind.Line\_List.Last().Move\_Label();

}

for (int i = 0; i < Loaded\_File.Gates.Count; i++)

{

switch (Loaded\_File.Gates[i].Type)

{

case (Gate\_Type.And):

MainWind.Gate\_List.Add(new And\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Nand):

MainWind.Gate\_List.Add(new Nand\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Not):

MainWind.Gate\_List.Add(new Not\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Or):

MainWind.Gate\_List.Add(new Or\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Nor):

MainWind.Gate\_List.Add(new Nor\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Xor):

MainWind.Gate\_List.Add(new Xor\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Xnor):

MainWind.Gate\_List.Add(new Xnor\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

case (Gate\_Type.Transformer):

MainWind.Gate\_List.Add(new Transformer\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

default:

MainWind.Gate\_List.Add(new And\_Gate\_Class(MainWind.Main\_Canvas, MainWind.Sub\_Canvas.Scale\_Factor, MainWind.Output\_Circle\_List, MainWind.Line\_List, MainWind.Input\_Button\_List));

break;

}

MainWind.Gate\_List.Last().Alive = Loaded\_File.Gates[i].Alive;

MainWind.Gate\_List.Last().Gate\_Bit = Loaded\_File.Gates[i].\_Gate\_Bit;

for (int x = 0; x < 3; x++)

{

MainWind.Gate\_List.Last().Output[x].Output\_ID = Loaded\_File.Gates[i].Output[x].\_output\_ID;

MainWind.Gate\_List.Last().Output[x].Line\_ID = Loaded\_File.Gates[i].Output[x].\_line\_ID;

MainWind.Gate\_List.Last().Output[x].Output\_Port = Loaded\_File.Gates[i].Output[x].\_output\_port;

MainWind.Gate\_List.Last().Output[x].Output\_Type = Loaded\_File.Gates[i].Output[x].\_output\_Type;

}

for (int x = 0; x < 2; x++)

{

MainWind.Gate\_List.Last().Input[x].Input\_ID = Loaded\_File.Gates[i].Input[x].\_Input\_ID;

MainWind.Gate\_List.Last().Input[x].Line\_ID = Loaded\_File.Gates[i].Input[x].\_line\_ID;

MainWind.Gate\_List.Last().Input[x].Input\_Type = Loaded\_File.Gates[i].Input[x].\_Input\_Type;

}

MainWind.Main\_Canvas.Children.Remove(MainWind.Gate\_List.Last().Rect);

MainWind.Sub\_Canvas.Children.Add(MainWind.Gate\_List.Last().Rect);

Canvas.SetLeft(MainWind.Gate\_List.Last().Rect, Loaded\_File.Gates[i].X);

Canvas.SetTop(MainWind.Gate\_List.Last().Rect, Loaded\_File.Gates[i].Y);

}

for (int x = 0; x < Loaded\_File.Inputs.Count; x++)

{

MainWind.Input\_Button\_List.Last().Bit = Loaded\_File.Inputs[x].\_Bit;

}

}

MenuItem\_SaveAs\_Click\_Method:

SaveAs and Save methods are similar that the amount of overlapping code might as well be one method which they both call. There is only one difference and that is that save should check to see if the file has already been saved and just save it whereas SaveAs should no matter what open the file explorer and let you save it.

Save as method opens the SaveFileDialog and creates the file path and changes saved to true. It will then call save file method for which the overlap of SaveAs and Save code (Covered below).

public void MenuItem\_SaveAs\_Click\_Method()

{

Save\_AS();

}

private void Save\_AS()

{

SaveFileDialog saveFileDialog = new SaveFileDialog();

if (saveFileDialog.ShowDialog() == true)

{

MainWind.Saved = true;

saveFileDialog.InitialDirectory = @"C:\Documents";

MainWind.File\_Location = saveFileDialog.FileName;

Save\_File();

}

}

Save\_File:

In Save file this has the IF statement due to needing a file path to be able to save the file. If saved is true then the system will be cleaned up to reduce the file size as much as possible. The file will be decompressed into a file type that can be serialized. A new FileStream will be created with the designated file location. A formatter is also created with the binary file type used. It then saves the file as binary under the settings of the FileStream, after its finishes it will then close.

If the file hasn’t been saved before then it will call the SaveAs method for which it will then be given a file path and then calls the method again and follows the process above.

public void MenuItem\_Save\_Click\_Method()

{

Save\_File();

}

public void Save\_File()

{

if (MainWind.Saved)

{

MainWind.Clean\_Up\_Method();

File\_Class Save = File\_Creation();

Stream stream = new FileStream(MainWind.File\_Location, FileMode.Create, FileAccess.Write);

IFormatter formatter = new BinaryFormatter();

formatter.Serialize(stream, Save);

stream.Close();

}

else

{

Save\_AS();

}

}

File\_Creation:

Because WPF objects can’t be serialized custom classes that just store variables are created. These are used to reduce file space as well. They each have custom constructors so that the variables are assigned as soon at the class is created.

public File\_Class File\_Creation()

{

File\_Class Save = new File\_Class();

for (int i = 0; i < MainWind.Gate\_List.Count(); i++)

{

Save.Gates.Add(new File\_Version\_Gate(MainWind.Gate\_List[i].Type, MainWind.Gate\_List[i].Alive, MainWind.Gate\_List[i].Input, MainWind.Gate\_List[i].Gate\_Bit, MainWind.Gate\_List[i].Output, Canvas.GetLeft(MainWind.Gate\_List[i].Rect), Canvas.GetTop(MainWind.Gate\_List[i].Rect)));

}

for (int i = 0; i < MainWind.Line\_List.Count; i++)

{

Save.Lines.Add(new File\_Version\_Line(Convert.ToString(MainWind.Line\_List[i].Line\_Lable.Content), MainWind.Line\_List[i].Output\_ID, MainWind.Line\_List[i].Output\_Num, MainWind.Line\_List[i].Input\_ID, MainWind.Line\_List[i].Input\_Num, MainWind.Line\_List[i].X1, MainWind.Line\_List[i].X2, MainWind.Line\_List[i].Y1, MainWind.Line\_List[i].Y2));

}

for (int i = 0; i < MainWind.Input\_Button\_List.Count; i++)

{

Save.Inputs.Add(new File\_Version\_Input(MainWind.Input\_Button\_List[i].Bit, MainWind.Input\_Button\_List[i].Input\_ID, MainWind.Input\_Button\_List[i].Input\_Port, Canvas.GetLeft(MainWind.Input\_Button\_List[i]), Canvas.GetTop(MainWind.Input\_Button\_List[i])));

}

for (int i = 0; i < MainWind.Output\_Circle\_List.Count; i++)

{

Save.Output.Add(new File\_Version\_Output(MainWind.Output\_Circle\_List[i].Output\_ID, MainWind.Output\_Circle\_List[i].Output\_Port, Canvas.GetLeft(MainWind.Output\_Circle\_List[i].Circle), Canvas.GetTop(MainWind.Output\_Circle\_List[i].Circle)));

}

return Save;

}

MenuItem\_New\_Click\_Method:

Gives a yes no message box asking if you want to delete everything. If they click yes then it will reset the program without saving anything.

public void MenuItem\_New\_Click\_Method()

{

MessageBoxResult messageBoxResult = MessageBox.Show("Are you sure you want to remove everything? Nothing will be kept!", "New Window", MessageBoxButton.YesNo);

if (messageBoxResult == MessageBoxResult.Yes)

{

MainWind.Reset\_Program();

}

}

# Gate Classes

The Base class which each of the sub classes inherit off holds all methods and templates that the children classes should hold.

For the variables it holds the some characteristics about the properties of the gate state. It also hold the classes for inputs, output and output bit. The input and output classes are arrays because there is a fix amount of inputs and outputs that the gates can have.

For Gate\_bit it has its own custom setter which means that as soon as the gate bit is set it will either change the output circle to the correct setting or update the variable of the gate it is linked to.

The base class has an abstract method called Gate\_Output\_Calc this is because each child class should override this method for each of their individual gate outputs.

public abstract class Gate\_Class

{

public Gate\_Type Type { get; set; }

public Rectangle Rect { get; set; }

public bool Alive { get; set; } = true;

//data storages for the input and output

public Input\_Class[] Input = new Input\_Class[] { new Input\_Class(), new Input\_Class() };

List<Output\_Circle> \_Output\_Circle\_List { get; set; }

List<Line\_Class> \_Line\_List { get; set; }

List<Input\_Button> \_Input\_Button\_List { get; set; }

/// <summary>

/// depending on what type of output it is it will update and change the corrosonding values.

/// </summary>

private bool \_Gate\_Bit;

public bool Gate\_Bit {

get

{ return \_Gate\_Bit; }

set

{

\_Gate\_Bit = value;

for (int i = 0; i < 3; i++)

{

if(Output[i].Output\_Type==IO\_Type.IO)

{

if (\_Gate\_Bit == true)

{

\_Output\_Circle\_List[Output[i].Output\_ID].Bit = true;

}

else

{

\_Output\_Circle\_List[Output[i].Output\_ID].Bit = false;

}

}

else if(Output[i].Output\_Type == IO\_Type.Gate)

{

if(\_Gate\_Bit==true)

{

Set\_Label\_1(i);

}

else

{

Set\_Label\_0(i);

}

}

}

}

}

public Output\_Class[] Output { get; set; } = new Output\_Class[] { new Output\_Class(), new Output\_Class(), new Output\_Class() };

public Gate\_Class(List<Output\_Circle> Output\_Circle\_List,List<Line\_Class> Line\_List,List<Input\_Button> Input\_Button\_List)

{

\_Output\_Circle\_List = Output\_Circle\_List;

\_Line\_List = Line\_List;

\_Input\_Button\_List = Input\_Button\_List;

Gate\_Output\_Calc();

}

protected void Part\_Constructor(Canvas Main\_Canvas)

{

Main\_Canvas.Children.Add(Rect);

Point Pos = Mouse.GetPosition(Main\_Canvas);

Rect\_Move(Pos);

}

Move\_IO:

This is such a useful function as it means that the links that the gate has to each of its UI objects can be updated on a gate bases. All it does is go through each of the gate input and output and moves the UI object to be aligned with the new position of the gate.

public void Move\_IO()

{

for (int i = 0; i < 3; i++)

{

if(Output[i].Output\_Type == IO\_Type.Gate)

{

\_Line\_List[Output[i].Line\_ID].Link\_Output\_Align\_Line(this);

}

else if(Output[i].Output\_Type == IO\_Type.IO)

{

\_Output\_Circle\_List[Output[i].Output\_ID].Align\_Circle(this);

}

}

for (int i = 0; i < 2; i++)

{

if (Input[i].Input\_Type==IO\_Type.Gate)

{

\_Line\_List[Input[i].Line\_ID].Link\_Input\_Align\_Line(this);

}

else if(Input[i].Input\_Type == IO\_Type.IO)

{

\_Input\_Button\_List[Input[i].Input\_ID].Align\_Box(this);

}

}

}

Children Classes

Each Children class has 3 things different about them. In their constructor they will have a unique setting for the rectangle UI. The height, width and fill will changed based on the type of gate. The type will be different for each and the method that they need to override for the calculation of the output needs to be the truth table for their gate.

And:

public And\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["And\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.And;

}

public override void Gate\_Output\_Calc()

{

if (Input[0].Input\_bit && Input[1].Input\_bit == true)

{

Gate\_Bit = true;

}

else

{

Gate\_Bit = false;

}

}

Nand:

public Nand\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Nand\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Nand;

}

public override void Gate\_Output\_Calc()

{

if (Input[0].Input\_bit && Input[1].Input\_bit == true)

{

Gate\_Bit = false;

}

else

{

Gate\_Bit = true;

}

}

Not:

public Not\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Not\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Not;

}

public override void Gate\_Output\_Calc()

{

if (Input[0].Input\_bit == true)

{

Gate\_Bit = false;

}

else

{

Gate\_Bit = true;

}

}

Or:

public Or\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Or\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Or;

}

public override void Gate\_Output\_Calc()

{

if (Input[0].Input\_bit || Input[1].Input\_bit == true)

{

Gate\_Bit = true;

}

else

{

Gate\_Bit = false;

}

}

Nor:

public Nor\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Nor\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Nor;

}

public override void Gate\_Output\_Calc()

{

if (Input[0].Input\_bit || Input[1].Input\_bit == true)

{

Gate\_Bit = false;

}

else

{

Gate\_Bit = true;

}

}

Xor:

public Xor\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Xor\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Xor;

}

public override void Gate\_Output\_Calc()

{

if ((Input[0].Input\_bit == true && Input[1].Input\_bit == true) || (Input[0].Input\_bit == false && Input[1].Input\_bit == false))

{

Gate\_Bit = false;

}

else

{

Gate\_Bit = true;

}

}

Xnor:

public Xnor\_Gate\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 115 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Xnor\_Gate\_L"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Xnor;

}

public override void Gate\_Output\_Calc()

{

if ((Input[0].Input\_bit == true && Input[1].Input\_bit == true) || (Input[0].Input\_bit == false && Input[1].Input\_bit == false))

{

Gate\_Bit = true;

}

else

{

Gate\_Bit = false;

}

}

Transformer:

public Transformer\_Class(Canvas Main\_Canvas, double \_Scale\_Factor, List<Output\_Circle> Output\_Circle\_List, List<Line\_Class> Line\_List, List<Input\_Button> Input\_Button\_List) : base(Output\_Circle\_List, Line\_List, Input\_Button\_List)

{

Rect = new Rectangle { Height = 75 \* \_Scale\_Factor, Width = 85 \* \_Scale\_Factor, Stroke = Brushes.Black, Fill = Application.Current.Resources["Transformer"] as Brush };

Part\_Constructor(Main\_Canvas);

Type = Gate\_Type.Transformer;

}

public override void Gate\_Output\_Calc()

{

if (Input[0].Input\_bit == true)

{

Gate\_Bit = true;

}

else

{

Gate\_Bit = false;

}

}

# Input and Output Class

This is just an extension of gate class but it needed to be its own class because it’s a group of data and repeating data due to it being in an array. The purpose is just to hold the value of each ports in the input and output of the gate.

For the Input/Output\_Type variable it has a custom setter so that the other values changes with the appropriate state that it’s in.

Output Class:

public class Output\_Class

{

private int \_output\_ID = -1;

private IO\_Type \_output\_Type = IO\_Type.Null;

private int \_line\_ID = -1;

public int Output\_Port { get; set; }

public int Output\_ID {

get {return \_output\_ID ; }

set {\_output\_ID =value; } }

public IO\_Type Output\_Type {

get { return \_output\_Type; }

set {

switch(value)

{

case (IO\_Type.Null):

\_output\_ID = -1;

\_line\_ID = -1;

break;

case (IO\_Type.IO):

\_line\_ID = -1;

break;

}

\_output\_Type = value;

}

}

public int Line\_ID {

get {return \_line\_ID; }

set { \_line\_ID = value; } }

public Output\_Class()

{

Output\_Type = IO\_Type.Null;

}

}

Input Class:

public class Input\_Class

{

private bool \_input\_bit = false;

private int \_input\_ID = -1;

private IO\_Type \_input\_Type = IO\_Type.Null;

private int \_line\_ID = -1;

public bool Input\_bit {

get { return \_input\_bit; }

set { \_input\_bit = value; }

}

public int Input\_ID {

get { return \_input\_ID; }

set { \_input\_ID = value; } }

public IO\_Type Input\_Type

{

get { return \_input\_Type; }

set {

switch(value)

{

case (IO\_Type.Null):

\_input\_ID = -1;

\_line\_ID = -1;

break;

case (IO\_Type.Gate):

\_input\_bit = false;

break;

case (IO\_Type.IO):

\_line\_ID = -1;

\_input\_bit = false;

break;

}

\_input\_Type = value;

}

}

public int Line\_ID {

get { return \_line\_ID; }

set { \_line\_ID = value; } }

public Input\_Class()

{

Input\_Type = IO\_Type.Null;

}

}

# Progress window

While there isn’t much to say on a nearly empty window, this bit of the code is what the user will see when they aren’t meant to interact with the MainWindow.

The only thing of use is to understand that the size of the work will change and so the progress bar is adjustable to however much is needed to be done.

public partial class Progress\_Bar\_Window : Window

{

public Progress\_Bar\_Window(int max)

{

InitializeComponent();

Bar.Minimum = 0;

Bar.Value = 0;

Bar.Maximum = max;

}

}

# Input Button Class

This is separate from input class as that is just a group of data that is linked to the input ports of the gate. This is about the UI objects that get added when you what to add your starting points. The Class inherits the button class so that it can have custom event handlers and all the variables needed for the input button class are in one place.

There is only 3 variables to this class because that’s all it needs, the main one to look at is Bit. It is important because it has its own custom setter which will in real time update the gate output. This allows the user to test single gates truth tables without needing to run the simulator. However it won’t then calculate the next gate after. This is the design I want because if it was to all change at once then the user wouldn’t understand what happened and how they got that output.

So in the setter it will change the attributes of the button class that it inherits to fit the desired effect and change the gate class variable to fit the new change. It will also tell the gate to calculate the new input. This process doesn’t repeat as when the gate sends the new output to another gate the class Input doesn’t have the custom setter that will tell the gate to calculate a new value.

Input ID and Input Port is just the ID for the gate and the port slot that it is connected to.

Because the class inherits the Button class it allows for a custom click event for the input. It nots the current value, as this is Boolean it just turns into the other value. While all that changed is the variable it’s due to the bit setter that also changes the visuals of the button and updates the gate.

public class Input\_Button : Button

{

private bool \_Bit = false;

/// <summary>

/// Changes the feature of the button for what input bit it is automatically.

/// </summary>

public bool Bit { get { return \_Bit; }

set

{

\_Bit = value;

if(!value)

{

Content = 0;

Foreground = Brushes.Black;

Background = Brushes.White;

\_Gate\_List[Input\_ID].Input[Input\_Port].Input\_bit = false;

}

else

{

Background = Brushes.Black;

\_Gate\_List[Input\_ID].Input[Input\_Port].Input\_bit = true;

Content = 1;

Foreground = Brushes.White;

}

\_Gate\_List[Input\_ID].Gate\_Output\_Calc();

}

}

public int Input\_ID { get; set; }

public int Input\_Port;

public List<Gate\_Class> \_Gate\_List = new List<Gate\_Class>();

private MainWindow \_MainWind { get; set; }

public Input\_Button(int ID, int Port\_Num, List<Gate\_Class> Gate\_List, Canvas\_Class Sub\_Canvas, MainWindow MainWind)

{

\_MainWind = MainWind;

\_Gate\_List = Gate\_List;

Input\_ID = ID;

Input\_Port = Port\_Num;

Sub\_Canvas.Children.Add(this);

Background = Brushes.White;

Content = 0;

Foreground = Brushes.Black;

Height = 20;

Width = 20;

}

//make this bit depend. So when the bit variable changes so does everything else.

protected override void OnClick()

{

Bit = !Bit;

}

public void Align\_Box(Gate\_Class Gate)

{

double[] hold = \_MainWind.Link\_Input\_Align(Gate, Input\_Port);

Change\_X\_Y(hold[0], hold[1]);

}

public void Change\_X\_Y(double x, double y)

{

Canvas.SetLeft(this, x-20);

Canvas.SetTop(this, y-10);

}

}

# Output Circle Class

Unlike the input class the circle class couldn’t inherit the ellipsis class so it’s just a variable of the class. The Class also contains a label for easy reading and understanding of what all the different colour system uses. As there is 2 UI Elements and they are both variables a few more methods were required. It also needs less variables as storing the bit of the value is pointless as it isn’t separate from the gate bit it’s connected to.

For the custom setter of Bit it updates the visual of the circle and label to relate to the new value. Because this will most like happen during the simulation part of the program which is working in a different thread from the main thread which has all the access to the WPF objects it needs to send a dispatch invoke. It calls a method as everything in that method just sets a different UI part. Before it was 3 lines of dispatching to change the values one at a time but now it does them all at ones.

Add and Remove UI are methods because I had lots of repeated code where they were being removed or added.

public class Output\_Circle

{

public Ellipse Circle = new Ellipse { Height = 20, Width = 20, Fill = Brushes.White, Stroke = Brushes.Black, StrokeThickness = 1 };

public Label Output\_Lab = new Label { Content = "0", Width = 16, Height = 29, Foreground = Brushes.Black };

public int Output\_ID { get; set; }

public int Output\_Port;

private MainWindow \_MainWind { get; }

public Output\_Circle(int ID, int Port\_Num, Canvas\_Class Sub\_Canvas,MainWindow MainWind)

{

Output\_ID = ID;

Output\_Port = Port\_Num;

Sub\_Canvas.Children.Add(Circle);

Sub\_Canvas.Children.Add(Output\_Lab);

\_MainWind = MainWind;

}

public bool Bit

{

set

{

if (value == false)

{

Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Circle\_0()));

}

else

{

Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, new Action(() => Circle\_1()));

}

}

}

private void Circle\_0()

{

Circle.Fill = Brushes.White;

Output\_Lab.Foreground = Brushes.Black;

Output\_Lab.Content = "0";

}

private void Circle\_1()

{

Circle.Fill = Brushes.Black;

Output\_Lab.Foreground = Brushes.White;

Output\_Lab.Content = "1";

}

public void Add\_UI()

{

\_MainWind.Sub\_Canvas.Children.Add(Circle);

\_MainWind.Sub\_Canvas.Children.Add(Output\_Lab);

}

public void Remove\_UI()

{

\_MainWind.Sub\_Canvas.Children.Remove(Circle);

\_MainWind.Sub\_Canvas.Children.Remove(Output\_Lab);

}

public void Align\_Circle(Gate\_Class Gate)

{

double[] hold = \_MainWind.Link\_Output\_Align(Gate, Output\_Port);

Change\_X\_Y(hold[0], hold[1]);

}

public void Change\_X\_Y(double x, double y)

{

Canvas.SetLeft(Circle, x);

Canvas.SetTop(Circle, y-10);

Canvas.SetLeft(Output\_Lab, x+2);

Canvas.SetTop(Output\_Lab, y-13);

}

}

# Line Class

Line Class couldn’t inherit the WPF Line Class so it’s just a variable of the class. Like the same with Output Circle Class I have a label to let the user know what the output of the gate is.

All the rest of the variables are just to sort out the location and position of the line. The class doesn’t need to store any values because when the gate class updates the input of another class it will just go straight to it.

In the constructor it has an IF statement, this is because when you load a file and it creates a new line class it will try and alien the output port with a gate. But as the gates have to be added last when the file is created the method can’t find the gate that it’s connected to. So the ordering is different if it’s a new or loaded class.

public class Line\_Class

{

public Line UI\_Line { get; set; } = new Line { StrokeThickness = 4, Stroke = Brushes.Red };

public Label Line\_Lable { get; set; } = new Label { Content = "0", Width = 16, Height = 29, Foreground = Brushes.Black };

public int Output\_ID { get; set; } = -1;

public int Output\_Num { get; set; } = -1; //this should only change if the gate type is the multiple output(type 7)

public int Input\_ID { get; set; } = -1;

public int Input\_Num { get; set; } = -1;

public double X1,Y1, X2, Y2;

public List<Line\_Class> \_Line\_List { get; set; }

public Canvas\_Class \_Sub\_Canvas { get; set; }

private MainWindow \_MainWind { get; }

public Line\_Class(int \_Output\_ID, Canvas\_Class Sub\_Canvas, MainWindow MainWind,int \_Input\_ID,bool New\_Class)

{

\_Sub\_Canvas = Sub\_Canvas;

\_MainWind = MainWind;

\_Sub\_Canvas.Children.Add(UI\_Line);

\_Sub\_Canvas.Children.Add(Line\_Lable);

Output\_ID = \_Output\_ID;

\_Line\_List = MainWind.Line\_List;

Track\_Mouse();

Input\_ID = \_Input\_ID;

if(New\_Class)

{

Output\_Num = \_Sub\_Canvas.Link\_Output\_Vaildation(\_Input\_ID);

}

}

public void Track\_Mouse()

{

Point Pos = Mouse.GetPosition(\_Sub\_Canvas);

UI\_Line.X2 = Pos.X;

UI\_Line.Y2 = Pos.Y;

X2 = Pos.X;

Y2 = Pos.Y;

Move\_Label();

}

public void Change\_X1\_Y1(double X, double Y)

{

UI\_Line.X1 = X;

UI\_Line.Y1 = Y;

}

public void Change\_X2\_Y2(double X, double Y)

{

UI\_Line.X2 = X;

UI\_Line.Y2 = Y;

}

public void Move\_Label()

{

double X = (X2 - X1)/2 - 5+X1;

double Y = (Y2 - Y1)/2 - 23+Y1;

Canvas.SetLeft(Line\_Lable, X);

Canvas.SetTop(Line\_Lable, Y);

}

//change this so that the values are generic and then just have it so that the X and Y coords are changed directly and don't need the method to do it.(A lot of work :(

public void Link\_Input\_Align\_Line(Gate\_Class Gate)

{

UI\_Line.Stroke = Brushes.Black;

double[] hold = \_MainWind.Link\_Input\_Align(Gate, Input\_Num);

Change\_X2\_Y2(hold[0], hold[1]);

X2 = hold[0];

Y2 = hold[1];

Move\_Label();

}

public void Link\_Output\_Align\_Line(Gate\_Class Gate)

{

double[] hold = \_MainWind.Link\_Output\_Align(Gate, Output\_Num);

Change\_X1\_Y1(hold[0], hold[1]);

X1 = hold[0];

Y1 = hold[1];

Move\_Label();

}

public void Remove\_UI()

{

\_Sub\_Canvas.Children.Remove(UI\_Line);

\_Sub\_Canvas.Children.Remove(Line\_Lable);

}

public void Remove\_Class()

{

Remove\_UI();

\_Line\_List.Remove(this);

}

public void Change\_UI\_Black()

{

UI\_Line.Stroke = Brushes.Black;

Line\_Lable.Foreground = Brushes.Black;

}

public void Change\_UI\_Red()

{

UI\_Line.Stroke = Brushes.Red;

Line\_Lable.Foreground = Brushes.Red;

}

}