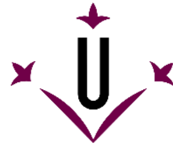


Universitat de Lleida

TREBALL FINAL DE GRAU

Estudiant: Martí Salazar Tarragó

Titulació: Grau en enginyeria electrònica industrial i automàtica

Títol de Treball Final de Grau: Creation of a combinational logic circuit sim o

Director/a: Francesc Sebé i Concepció Roig

Presentació

Mes: Setembre

Any: 2023

1. Abstract..3
2. Introduction..3
2.1. Project overview and objectives..3
2.2. Software used..3
2.2.1. Unity engine..3
2.2.2. Inkscape..4
3. Existing simulators and field research..4
3.1. Simulator examples..4
3.1.1. Simulator main features..6
3.2. Simulator features flaws..6
3.2.1. Library cluster..7
3.2.2. Slow workflow..7
3.2.3. Lack of interactivity..7
4. UX design process..7
4.1. Design objectives..7
4.1.1. Addressing user’s actions..8
4.2. Panels layout design..8
4.2.1. Layout distribution..8
4.2.2. Layout population..10
4.2.3. Layout position..11
4.3. Simulator specific panels..11
4.3.1. Add Wheel Panel..11
4.3.2. Control Wheel Panel..12
5. User actions and controls..13
6. Technical overview..14
6.1. Usage example..14
6.2. Early definitions..19
6.3. Element hierarchy..19
6.3.1. Logic gate object..20
6.4. Script management..23
6.4.1. Connections custom script..23
6.4.2. Wire custom script..28
6.4.3. Gate Logic custom script..29
6.4.4. Door Logic custom script..29
6.4.5. Door Manager custom script..31
6.4.6. DoorTypes ScriptableObject custom script..34
6.4.7. Dragger custom script..34
6.4.8. Bezier Line Renderer Controller custom script..35
6.4.9. Player Controller custom script..39
6.4.10. Buttons custom script..41
7. Conclusions..42
8. References..43

1. Abstract

This document investigates the viability of creating a fully functional combinational circuit simulator by an industrial engineering student. The conclusions and results of this project have been given by a student with no precious knowledge regarding the engine Unity 2021, nor the coding language used C#. The document begins with a brief explanation of the engine and other software used, then a discussion going step by step through the design process, UX design and script management of the simulator. Then finishes off with the conclusions.

2. Introduction

2.1. Project overview and objectives

This project has the aim of creating a combinational circuit simulator from scratch. It's important to mention that throughout the past eight months I have been spending a lot of time learning how to use Unity 2021 and Inkscape and learning how to code with C#.
This document assumes all of that knowledge and doesn't mention any of the structure and code prototyping the simulator. This is the product of all of that learning process and the condensed information that the simulator has ended up being. There will be no explanation of why I have chosen Unity or C# due to the fact that it has been out of personal preference. There are other engines and tools to be used in order to make a functional logic circuit simulator, but I won't discuss it.

2.2. Software used

2.2.1. Unity engine

To make this simulator I chose Unity 2021.3 because back when I started it was the most stable version released so far. Unity is a cross platform engine developed by Unity Technologies. It's mainly used for interactive experiences for mobile, desktop and virtual reality. It's also commonly used to create games by developers all around the world (Unity Technologies, n.d.).
The best asset that Unity presents out of personal preference is the availability of information you can find online. It's very intuitive and easy to learn. Hence the existence of all the resources for you to find and learn how to use it are of great importance. Unity is an object oriented programming engine (OOP) such as C++, Objective-C, Smalltalk, Javascript etc. This model is based on data contained in objects and procedures known as methods. These objects are class instances which is an abstract type of data that encapsulates data and functions to access it.

2.2.2. Inkscape

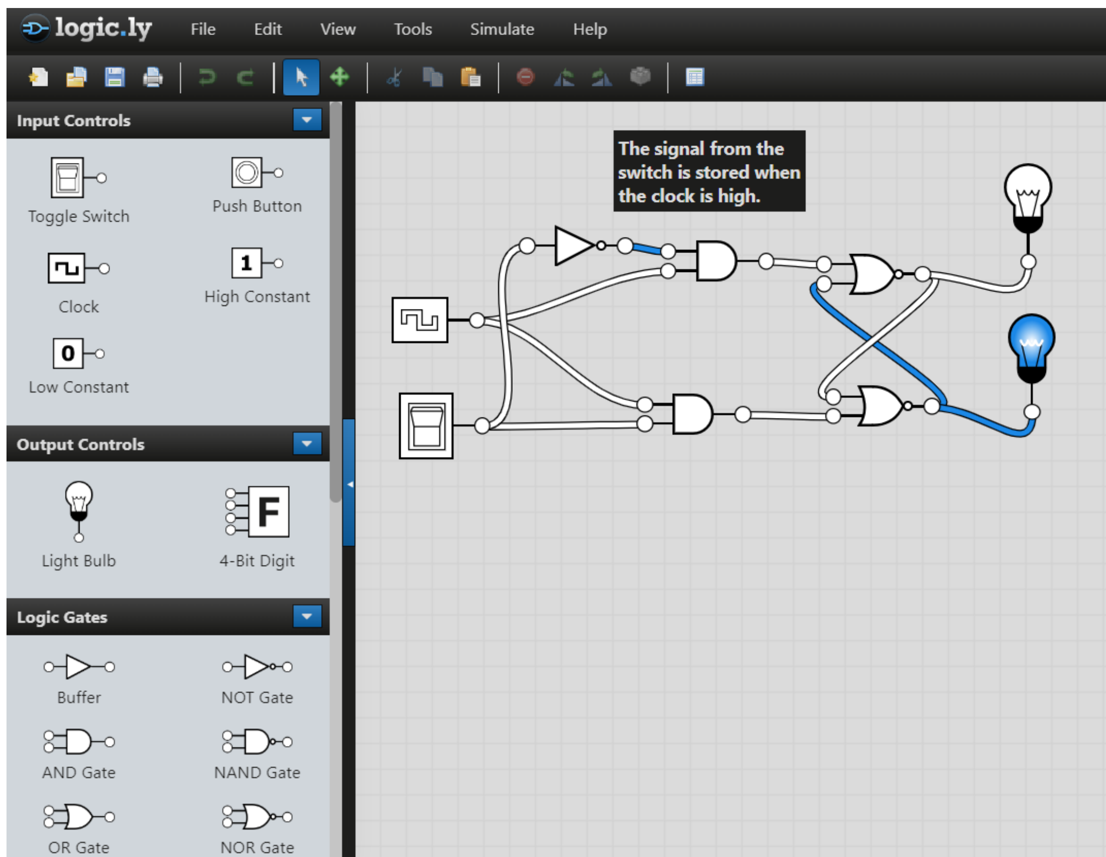
This software is an open-source vector graphics editor. It offers a rich set of features and is widely used for both artistic and technical illustrations. It has proven very helpful due to how easy it is to use. I had no previous knowledge of it but I mainly used it to create sprites for the simulator such as gate symbols, buttons and backgrounds.

3. Existing simulators and field research

3.1. Simulator examples

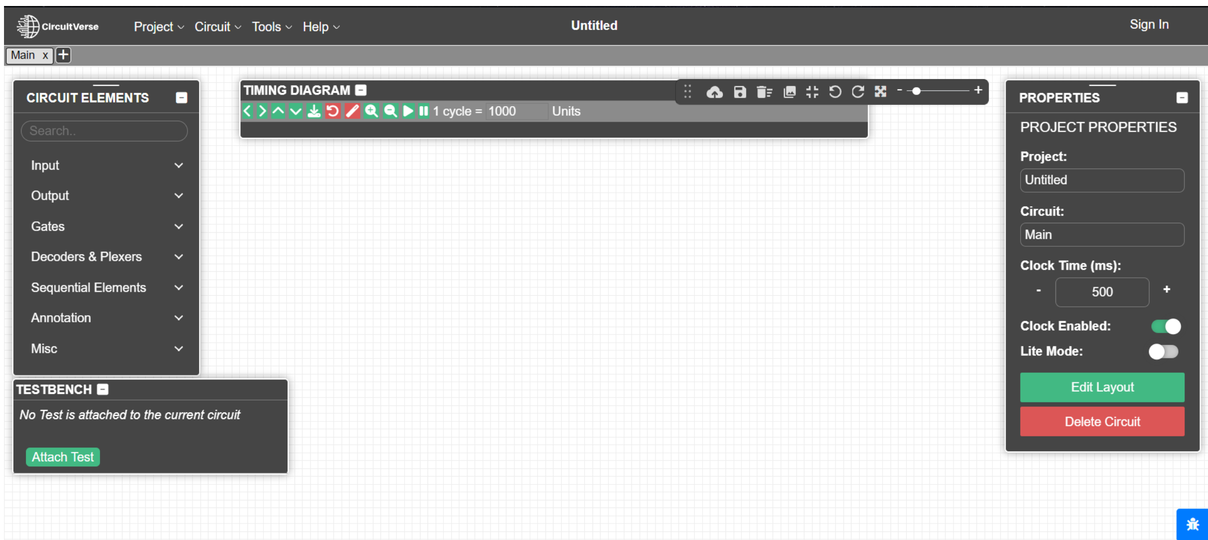
In order to create the simulator from scratch it's important to have enough evidence of what has already been developed. These are a few of the main examples I have been testing.
There are common features among all the simulators. They are commonly found on softwares that I'm trying to emulate, some examples are:

- Logic.ly (Bowler Hat LLC, n.d.):



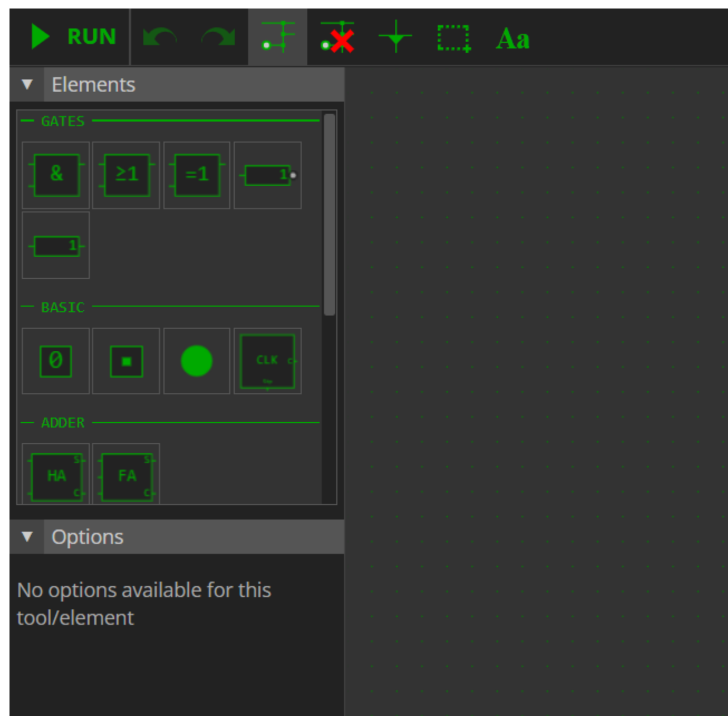
This is a demo of the product. There are certain tools such as an element library on the left and a canvas to place the elements. It includes other characteristics such as file management on the top bar.

- CircuitVerse (CircuitVerse, n.d.):

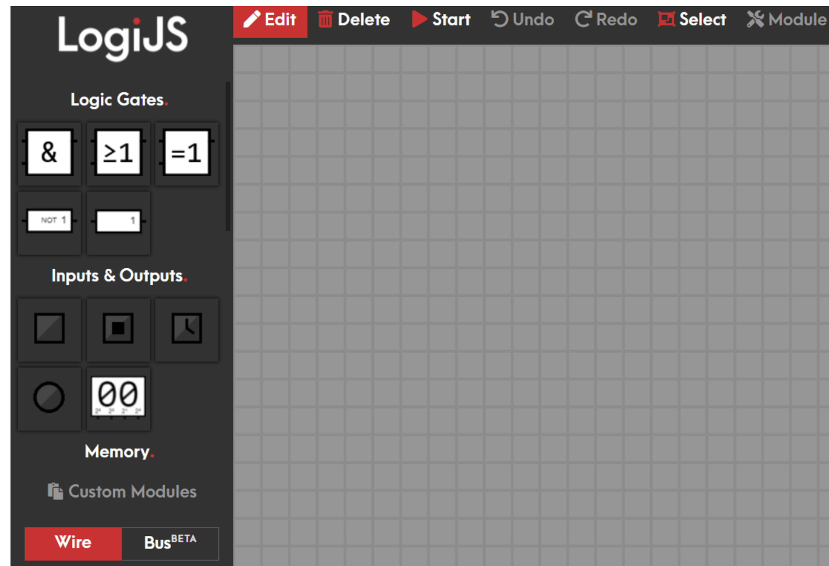


This tool which differs from the previous one on the UI. There are floating user interfaces that hold the elements and the actions.

- Simulator.io (Cristian Born, n.d.):



- LogiJS (LogiJS, n.d.):



This one looks rather simple but it's surprisingly easy to use, the symbols are big and well represented.

3.1.1. Simulator main features

There are multiple features that make a simulator easy to use. Such features can be observed at the examples I provided previously. Some of them are:

- Ability to select groups of elements and perform actions all together. This makes the workflow more dynamic. It lets the player move elements around the scene by groups and on a bigger scale, this feature is necessary for large and complex digital circuits. In my simulator I decided this feature to be secondary.
- Node visibility: when connecting and deleting wires it's important that the nodes that the wires connect to are always visible and easy to spot.
- Tooltips are important when it comes to learning how to use a software.
- Libraries: to drag and drop the elements from a UI component to the workspace, normally the components are within groups, such as logic gates, inputs and outputs.

Either the simulators are very complex and clearly engineering oriented, or too simple and intended just to play with them. This is not an undesirable feature on its own, but there is ground for improvement.

3.2. Simulator features flaws

Based on the observation, these features are important and commonly used on the simulators I tested. What's more, there are also certain shared flaws I would like to comment on:

3.2.1. Library cluster

One of the ways to organize elements in a UI is the use of a tab system. The elements are organized within parent groups. It works very well to specifically organize different elements. This system works well for large groups of elements but it's not user-friendly when the elements have a wide variety of features. When you are searching for the type of gate that you are looking for, because all the elements are cluttered it’s hard to recognize the gate because it’s hidden within tabs. So sometimes you end up spending 20 seconds just to look for a gate that’s hidden in plain sight. Most of the time the user will be using the same set of primary gates, inputs and outputs, so there is not a distinction between the elements that are used the most and the elements that are almost never used.

3.2.2. Slow workflow

As I said earlier, because there is no distinction between the most used elements, and the almost never used, when using the most common ones, you spend a short amount of time looking for the gate, since you have done it multiple times you know where it is, but it still takes around three clicks to get to the desired tab, then drag and drop the gate on the workspace, and then you still have to configure the number of inputs. It takes longer than it should.

3.2.3. Lack of interactivity

Most of the simulators lack a sense of interaction when the user is doing things, like when the user makes a change on the gate, or presses a button, or moves elements around the scene, there is no feedback.

4. UX design process

4.1. Design objectives

To solve the problems found in other simulators, we will have to directly address them and come up with a solution to satisfy issues without bringing new undesired ones to the system proposed (Neusesser, 2023).

My design process is centered around improving the workflow speed as much as possible. The highest priority is to make the simulator easy and quick to use, easy to add, delete, duplicate, change values, or move things around as quickly as possible. It is also important that it's very intuitive and self explanatory.

The user performs certain actions while operating the software. Some actions are performed more often than others. In order to improve the workflow we need to focus on the actions that the user performs the most, and improve them.

Such actions are:

1. Dragging and dropping elements.
2. Adding elements.

3. Deleting elements.
4. Editing elements as well as adjusting connections.

4.1.1. Addressing user’s actions

We don’t need to address the “drag and drop” action due to its own nature,. There is no easier way to drag and drop elements than just moving them with the mouse wherever the user needs to.

When it comes to the others, I decided that such actions will be performed through a User Interface (UI). The actions proposed are very similar to one another so it makes sense that they’re grouped within the same system. My reasoning also comes from the research made in which I have observed that all the simulators have a user interface grouping user’s actions.

I decided to group the add, delete and edit actions in the same UI panel due to the fact that all of them hold the same importance and will be used equally by the user. All that’s left to consider is the panel layout.

The UI will spawn right at the position of the mouse cursor when the player presses the right button on an element, or on the background space. The reason for this is that the panel where the user will be editing the elements, adding, editing or deleting, will be right next to the mouse position. So he doesn’t need to look at another separate part of the screen, and doesn’t need to move the mouse too much, thus reducing the time the user spends performing actions.

4.2. Panels layout design

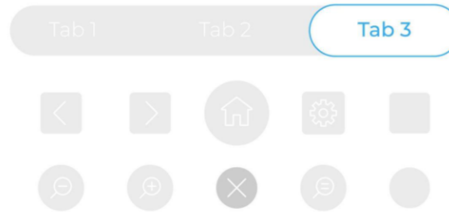
In order to create a panel, there are several things to consider: the layout of the UI, the distribution of certain elements, and how do we populate the UI (whether it will be filled with buttons, or sliders).

Responsive feedback is one of the most important aspects of a digital interface. The user needs to feel the software responds to the user’s actions. In a much bigger scale of a software product design, there are teams that analyze the data collected on user behavior, such as site views, click rates, heatmaps. They also have surveys, and before launching the software they also have testers that dive into it to give feedback to the developers, so based on this data the designers take decisions on how to improve the effectiveness of their design.

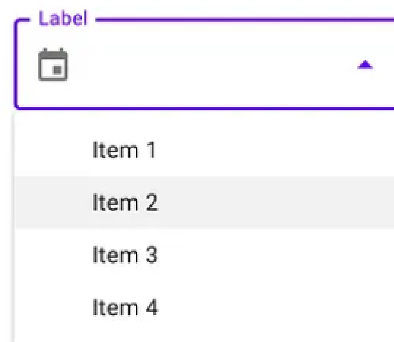
4.2.1. Layout distribution

How the elements populate the panel is of utmost importance. There are some examples to consider:

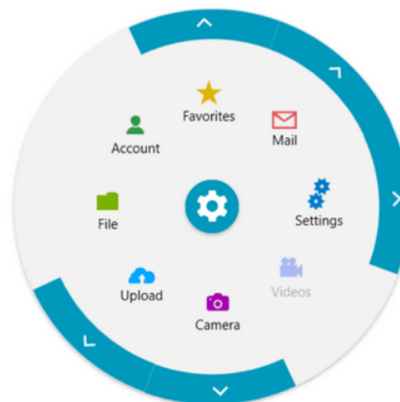
- Tabbed interface: A system of pages ordered by tabs that you can enable and disable.



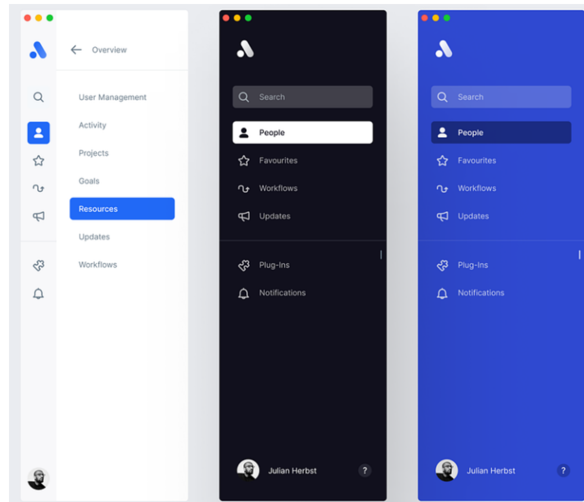
- Cascading menu: A way to order options in a hierarchical structure. It occupies very little space because the options are shown when the user interacts with the menu.



- Radial User Interface: This layout works well for a finite number of elements, and it's very easy and intuitive to use.



- Side Menus: They normally appear on the side of the screen and contain a list of related elements. They're normally used for features that are not continuously used.



These examples are the ones that I considered that hold the most relevance when it comes to responding to the design objectives that I proposed. I decided to use the radial user interface for two reasons: one being that the actions that I need to implement are finite and I don't need an ever scaling system to hold too many elements, and the second one that the fact that is radial lets me put the element that we are configuring at the center of this menu, so that the object that you are performing actions on is clearly visible and intuitive.

4.2.2. Layout population

There are multiple options to consider here, that is, how we populate the chosen layout is as important as the layout itself. Some examples are :

- An input field, where the user can write with the keyboard.
- A button that can be pressed with the mouse.
- A slider the user can drag with the mouse.
- A radial slider.
- A dropdown with multiple options the user can click.
- A pop up modal window when certain things are clicked.
- A switch or a toggle that can be enabled or disabled with a click by the user.

There are some valid options to consider here. The most suited for this project would be:

- The button. It's the skeleton of any properly designed UI. It's very easy and intuitive to operate and also very easy to represent. It comes with various shapes as well so it's very versatile to use. It lets you select specific options.
- The slider. You can slide it to suit any value that's represented there. It can be a numeric value or just different options. It's very useful to represent numerical increments.

I decided to keep it simple and only go for two different elements to populate the chosen layout. The reason is that it's very important to respond to the design objectives decided earlier. If the layout is too populated with different element types it can be hard and overwhelming for the user the first time using the software. This design also responds to the

research made earlier. Almost all of the simulators hold buttons as the main course of action in the UI.

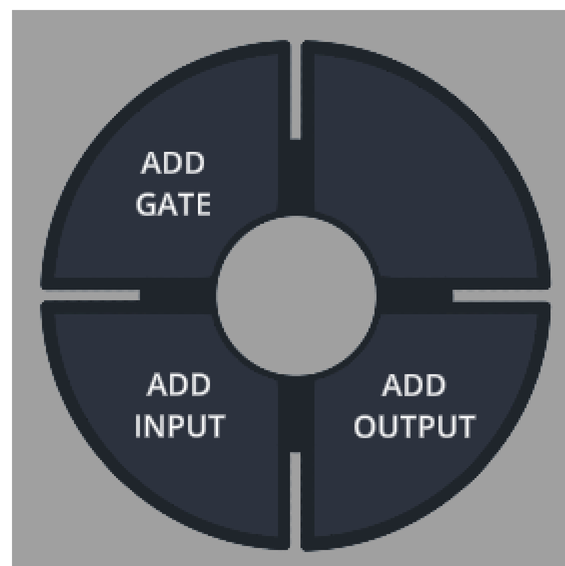
4.2.3. Layout position

The actions that are intended to be performed are about an element. Deleting, editing, and adding are actions that you perform on an element. For this reason, the panel position will be wherever the element is. It makes sense since the actions are about this element, and the user won't have to drag the mouse around the screen every time that he wants to edit an element. Because the UI will appear around the mouse, every option is as close as any other. There is the same distance for the mouse to get to each element due to the radial layout, and as mentioned earlier, I consider each action editing, adding and deleting equally important.

4.3. Simulator specific panels

Taking into consideration the scope of this project I considered two different UI wheel panels in order to fulfill the basic actions. The "Add wheel panel" will perform the adding action and the "Control Wheel Panel" will perform the other two: deleting action and editing action.

4.3.1. Add Wheel Panel



It might seem odd that there are four segments for four buttons, but only three are populated. That is because the fourth button is not implemented yet and it will control the play and pause simulation status for sequential circuits, as I mentioned earlier this is not in the scope of the project. But it's convenient to always have the option to be implemented in case this project is extended in the future.

This wheel will spawn on the mouse position whenever the user pressed the right button on an empty space on the workspace. It is suitable that in order to add an element on the screen, you press on the position it has to be located.

When one of the buttons is pressed the wheel will disappear to let the user continue working and start editing the instantiated element.

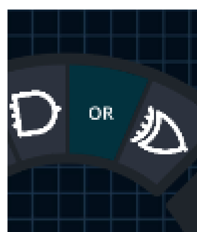
4.3.2. Control Wheel Panel

This wheel will spawn when the user presses the right button on an existing gate, it's intuitive that this panel appears when the user wants to edit an element he is clicking, the basic features are separated within sectors, to separate its functionality. So it's very easy to identify that if you want to change the gate type, you use the top sector, for example.

This wheel holds more buttons and options than the "Add Wheel Panel" and it's also bigger. There are more features to be controlled when editing gate properties.

The wheel is a great way to make the workflow much faster but we can already tell the problems that's causing. It certainly improves the workflow but at the same time it also limits the number of actions that the user can perform. Because the scope of this project is to only tackle digital combinational circuits this design is suitable.

When hovered over one of these buttons the button shape and color will change to a tooltip to improve the interaction with it.



5. User actions and controls

After the research done on other existing simulators, I have to make sure to specify the actions the user will want to perform. Such actions are:

- Move elements around the scene. This means that the user will be able to reposition elements at any stage of the simulation, and also rotate them (R key) without altering the logic state of the object. This will be achieved via drag and drop with the mouse left button. This feature is very important because it's the core of any simulator.

- Interact with these elements.
    - Wires: the user will be able to draw wires between connection points wires that connect the elements together. The elements react to these interactions, when the user connects several elements. Their logic values will be updated and shown at runtime.
    - Input interactions: the user is able to change the input values at will. Toggling the input elements with the left mouse button click.

- Change built element values.
    - Number of inputs, in case the player needs to attach more elements to the same component, this will be achieved through a button in a "Control Wheel Panel".
    - Delete element. The user needs to be able to delete any elements. This will be achieved through a button in the "Control Wheel Panel".
    - Input spacing (vertical and horizontal). In case the elements end up too close together and it difficulties visibility and interaction with the elements.

- Add elements to the scene. The user needs to add elements easily and at will.

- UI interaction. The user is able to interact with user interfaces, and has a way to make them appear, that will be done with the right button. If the button is pressed on a component, the "Control Wheel Panel" will pop up, and if it's clicked on empty space the "Add Wheel Panel" will pop up instead.

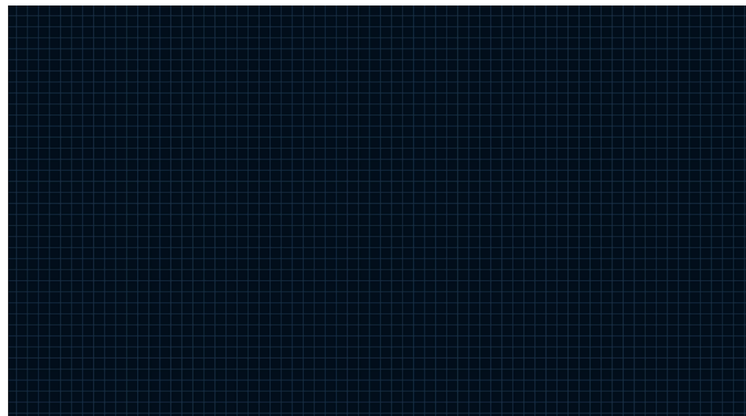
This is an exclusive list. All the actions that are not mentioned above won't be implemented.
I designed the user actions basically on two inputs, right click and left click. The right click is for the player to change things, basically making UIs appear. And the left click is for pressing buttons and drag and drop elements around the scene.

6. Technical overview

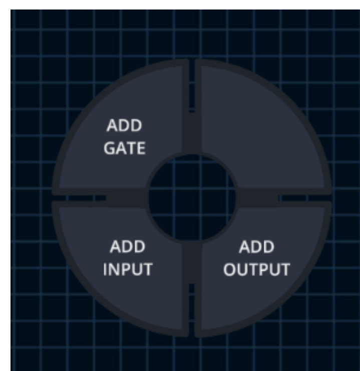
6.1. Usage example

In this section there is a simple explanation of an example of how to use the software. The objective of this step tutorial is to get the user through the main features of the simulator until we get a simple logic system. The made-up combinational circuit doesn't have a direct application. This explanation's objective is to go through the main feature, not to meddle whether a certain circuit makes sense for a specific application or not.

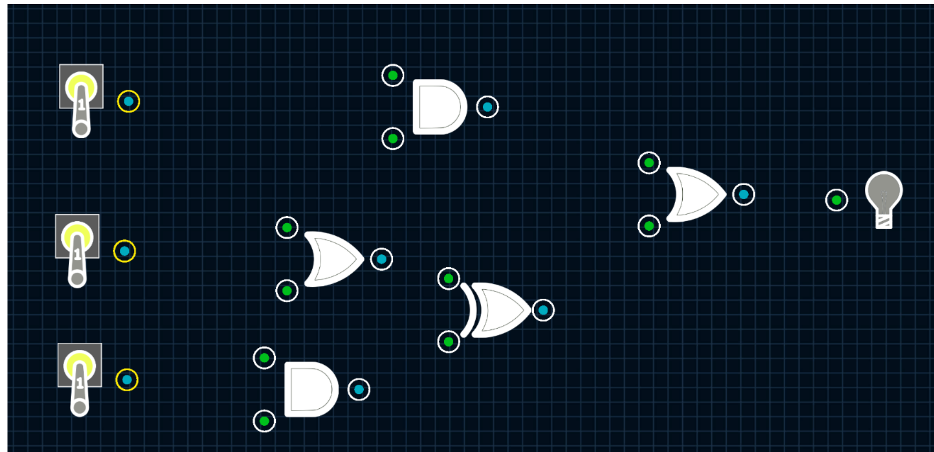
When starting, appears a blank canvas with lines representing the grid:



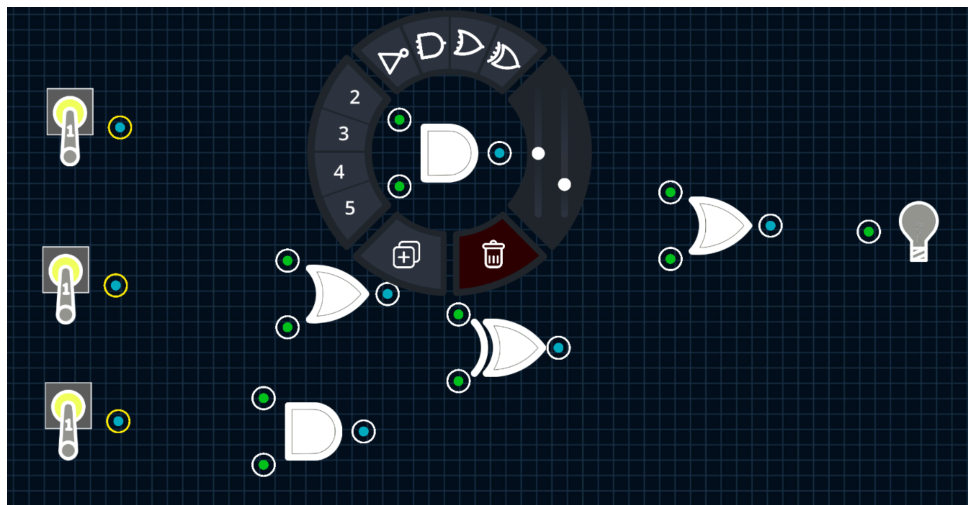
In order to start creating gates the user has to press right button on any part of the canvas. When done it will be presented with this UI. When pressed the user interface will disappear and will spawn the desired selected element.



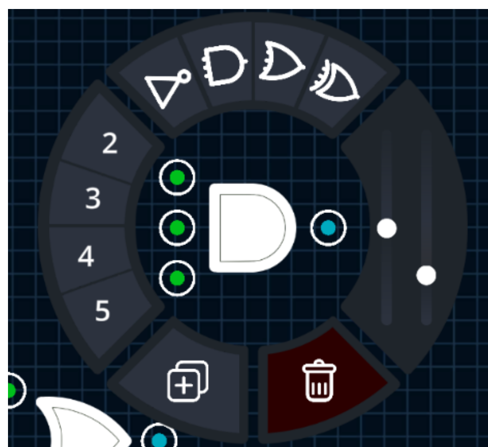
The user now knows how to add elements to the existing empty canvas, the user will add the following.



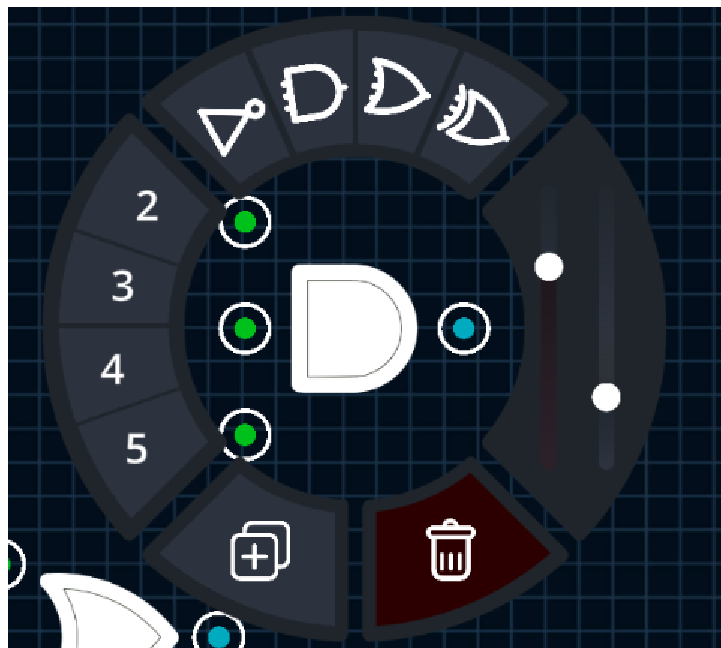
As we see, the gates that have been spawned have two inputs by default. We can change that by pressing the right button on the desired gate, and the button “3” on the UI pop up. The user will change the value to the upper AND gate and then on the right “OR”.



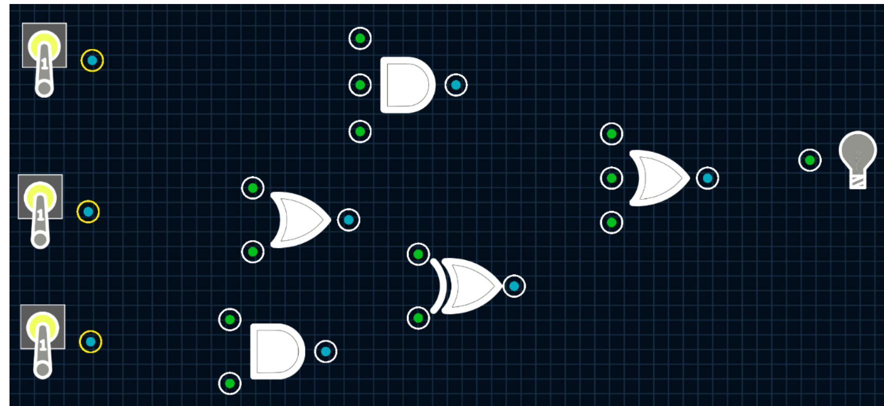
As we can see, after pressing the button, this time the UI doesn’t disappear and there is a third connection at the selected gate. We can proceed and delete the current gate with the red button, duplicate it or even change the type of the selected gate.



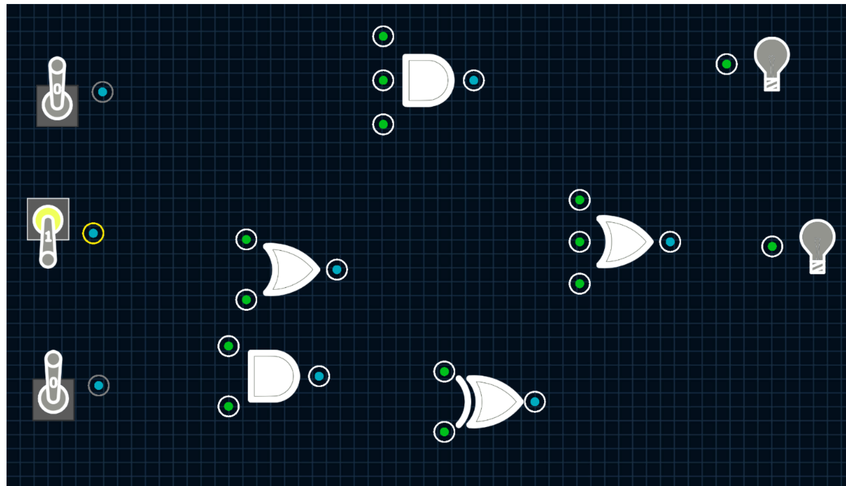
To make things easier to spot, we will modify the vertical distance of the connections to make it more visible and easier to operate. We will slide the left slider until we find the desired distance.



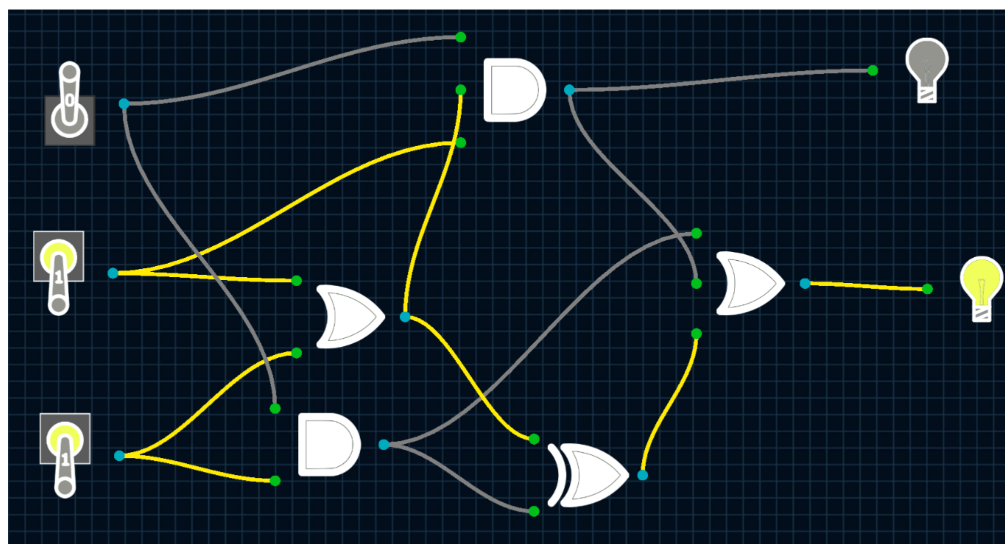
Now we have all the gates configured as mentioned earlier. The canvas should look like this now:



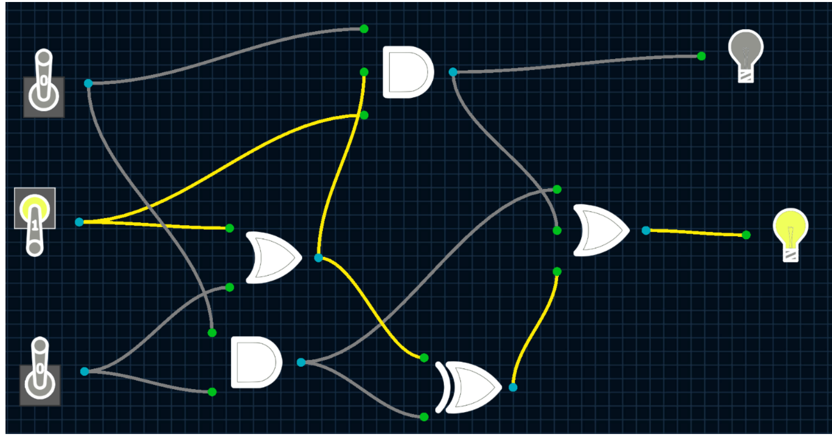
To move the elements around the canvas you drag and drop them with the pivot point on the symbol of the gates. We can try it with the XOR gate, for example:



The user will proceed to connect the wires in the following configuration. Keep in mind that the blue outputs let you drag more than one wire from them whereas the green inputs don't. In order to do so we will press the left button on the connections and drag it to the signal receiving connection.
The circle around the gates symbolizes whether they are connected or not. We will connect them all until they disappear. It should look like this:



Now we can proceed to interact with the inputs and see what happens with the combinational circuit we have created. To interact with the inputs you press the left button on the gate's symbol.



6.2. Early definitions

Explanations:

- When referring to a gate we will refer to it with capital letters to avoid confusion. For example when I talk about an “INPUT”, I mean that the gate itself is an input that you can toggle to give a desired output of high or low. When I talk about an “input” I mean about the attached input to the gate. A gate can have multiple inputs where you can drag wires from.
- A ScriptableObject is basically a data container that can be used to store large amounts of data, independently of script instances. It is a class that can be serialized and saved as an asset in your project. This means that you can create a ScriptableObject once and then use it in multiple scenes or Prefabs. (Unity Technologies, n.d.)
- An instance is a plain copy of a GameObject or ScriptableObject. When you create a GameObject or ScriptableObject, you are creating a template. You can then create multiple instances of that template, each with its own unique properties. (Unity Technologies, n.d.)
- GameObject, represented in the project with a gray square in the inspector, basically most the elements. A GameObject is the fundamental object in Unity. It is a container for components, which implements the real functionality. (Unity Technologies, n.d.)
- A sprite is a type of file that has a bit map of varying sizes. This information can then be used by a SpriteRenderer component on a GameObject to actually display the graphic. (Unity Technologies, n.d.).
- A collider in Unity is a component that defines the shape of an object for the purposes of physical collisions. It is invisible. (Unity Technologies, n.d.)
- A prefab. It is represented with a blue square. It is a saved asset that contains the complete definition of a GameObject, including its components, materials, and scripts. Prefabs can be used to create multiple instances of the same object in your scene, or to share objects between scenes. (Unity Technologies, n.d.)
- A Line Renderer is a component used on a GameObject to create line shapes with a finite number of points. (Unity Technologies, n.d.)
- A canvas is the main framework for UI creation, it holds all the elements that compose the user interface. (Unity Technologies, n.d.)

For future references, we establish that the color code for logic implementation goes for:

- White, empty state. No connections.
- Yellow, high state.
- Gray, low state.

6.3. Element hierarchy

In this section I justify my coding design and structure choices and how they have been achieved. The simulator has some core components that, in its entirety, make the simulator.

These core elements are:

- Gates
- Connections between elements.

- Logic.
- User Interface.

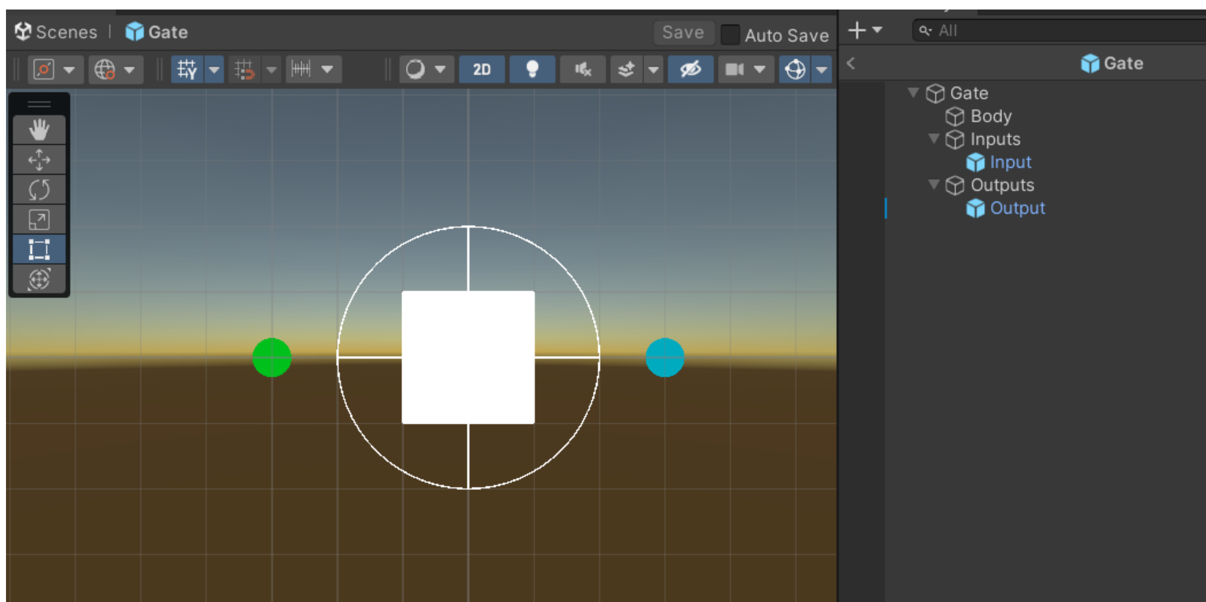
6.3.1. Logic gate object

All logic gates have a similar behavior. All of them include all types such as INPUTS and OUTPUTS. Primary logic gates have in common:

- Body and gate type representation.
- Connections to attach wires.
- Variable number of inputs and outputs to attach these wires.
- Logic.

Even though these are very similar to the core elements of the simulator, in this case I am referring to the specific connections of a specific element, and a specific piece of logic on this element.

This is the hierarchy in the inspector tab of a Gate:

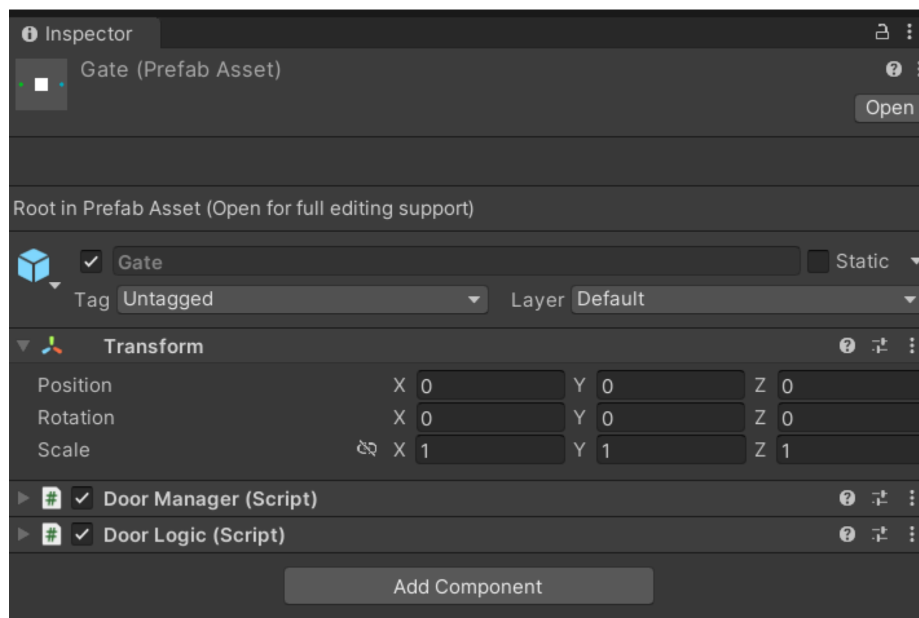


As you can see a gate is a prefab which contains other prefabs. It has three children, the Body, representing the white square, and Inputs and Outputs parents, each containing its corresponding prefab.

A Gate object is merely a generalization of all the gates, it carries the information of all the gates at the same time. It is much more time efficient to design this whole system around the Gate archetype due to the fact that it has more common ground than any other element, meaning that regardless of what kind of combinational gate we are dealing with, it will have inputs and outputs, and it will need other signals to get from or transmit to information. It also

has to be dragged around the scene, be able to be deleted, and created.
The difference between logic gate types is just merely a matter of parameter configuration.

At runtime, this component will just have a specific gate type selected, let's say an AND type logic gate, then with this information the prefab itself will change the input/output configuration. The white square will turn into an AND logic gate shape sprite, and there will also be an AND logic associated. When the object detects that its inputs are attached there will be an output signal with an expected AND behavior. If let's say now the configuration is set to an input, the gate will just change the logic from AND behavior to just give a high/low signal. It will remove it's input connection and will enable the toggle function to let the user change it's value at will.



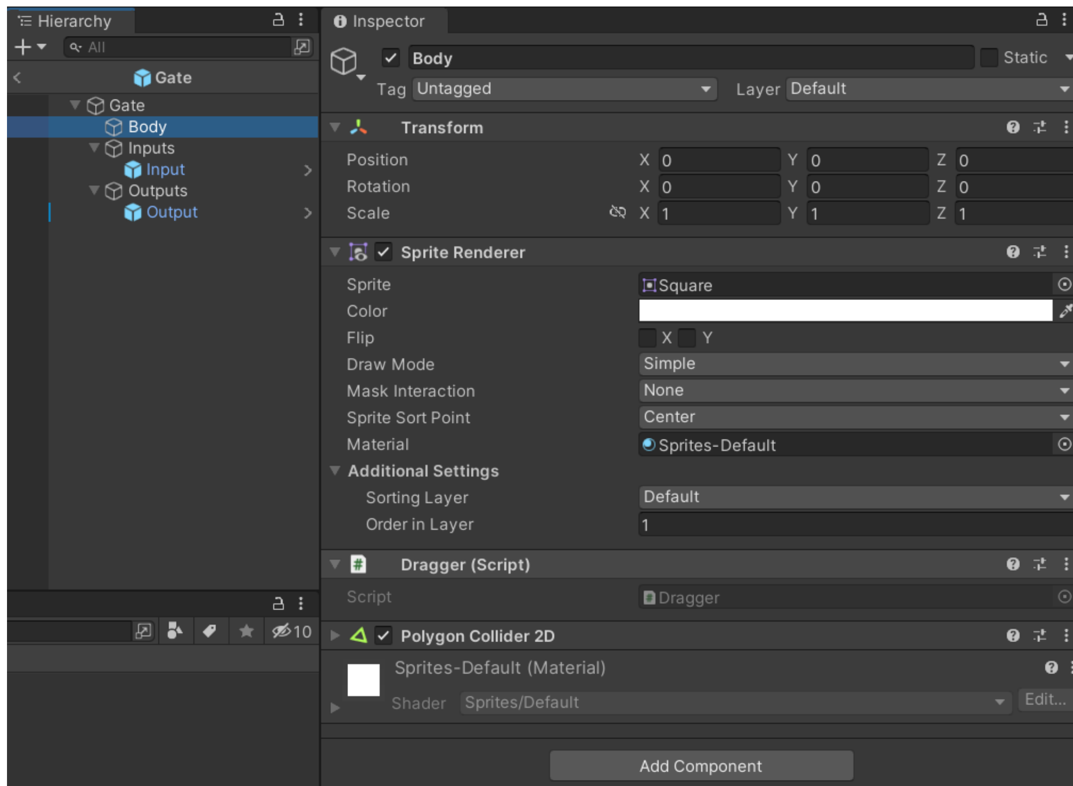
This prefab has 2 added scripts: Door Manager and Door Logic, I intentionally took the decision to refer the gate as Door on the scripts because it makes it easier to identify while coding what am I referring to. If it's a door, it means it's referencing the script and if it's gate it's referencing the object on the scene.
There is no need to nail the scrutiny of all the specific names and nomenclature of a digital circuit, due to that what's running behind the scenes. There are no high or low voltages that carry information through a wire to the next logic gate, there is just a series of scripts that work together to make it look like a digital circuit which is running.

DoorManager.cs and DoorLogic.cs are both a core part of the whole system, but to be able to explain them properly other more specific type scripts must be shown first.

For now let's say that DoorManager.cs manages gate configuration, input/output number, logic type selection and the changes of all these variables at runtime.

The gate's children are:

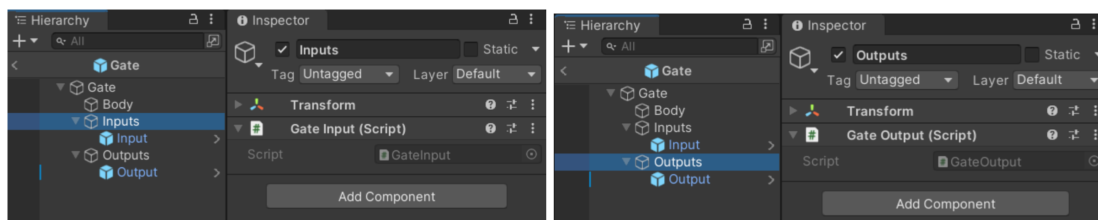
Body:



The most important part of the Body is that it has an added component which is a PolygonCollider2D, a Sprite Renderer, and a custom Dragger script.

A PolygonCollider2D will be used to detect mouse input, and the SpriteRenderer to display the type of logic gate.

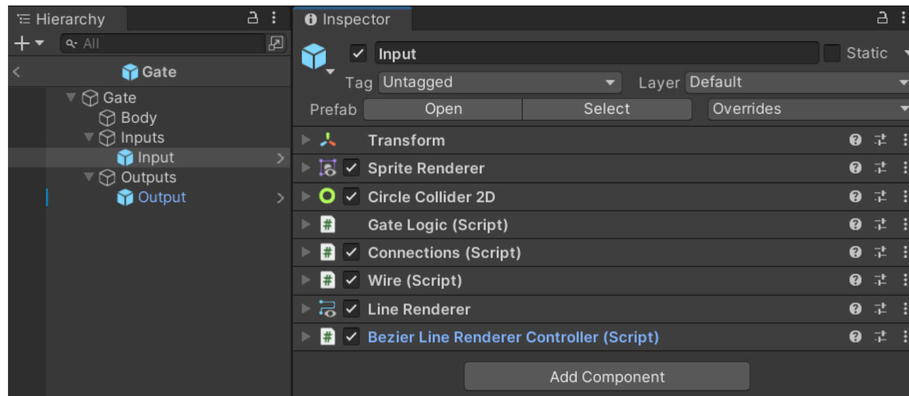
Inputs and outputs parent elements:



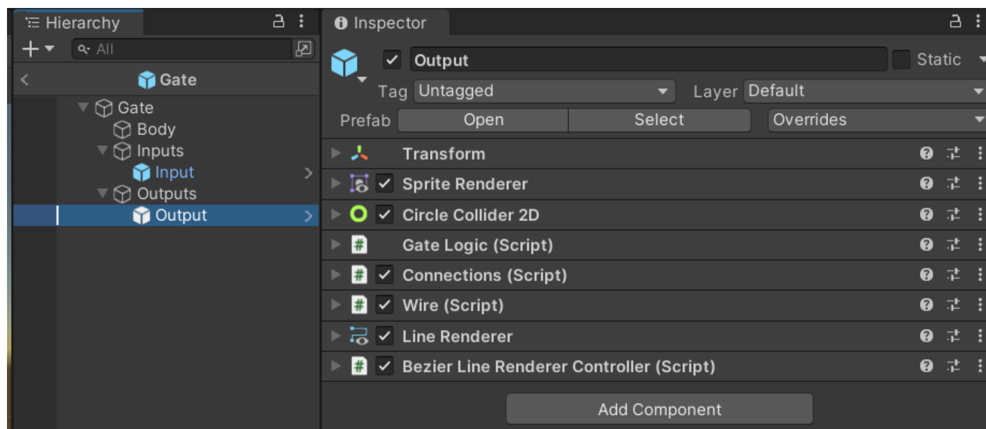
These input and output parent elements are only created to group the prefabs instead of having them gathered around. At runtime we might have more than one instantiated input so it's very convenient to have them being children of a parent object. The Gate Input / Output custom scripts are empty scripts whose only functionality is to be identified by other scripts.

Inputs and outputs children prefabs:

Input:



Output:



As you can see both input and output prefabs have the same components. In code, we play around the fact that an input can only have one wire connected, but an output can have multiple wires connected.

6.4. Script management

6.4.1. Connections custom script

As mentioned earlier, Connections.cs is a script attached to the Input / Output prefabs. These prefabs represent the anchor points for wire attachments.

This script is one of the most important ones. It's attached to every node available to be connected through a wire, meaning that each door will have more than one Input and Output elements, which also contain the Connections custom script.

This script behavior changes whether it's attached to an Output Prefab or to an Input Prefab. Let's start with the simplest of them, the input.

This is the explanation of how elements attach to each other, and how they keep the references of each other as well.

Inputs are represented with a green color dot. As I mentioned earlier they have a CircleCollider (used to detect mouse input). When you press the left button on the input, the script tries to detect if the click was close to the input. If it was (through a custom radius) then it starts drawing the wire as you drag the mouse around the screen. So basically this would be when you are trying to connect an input to an output, for instance.

It also passes the mouse and the input coordinates to the Wire custom script that is in charge of updating the Line Renderer component so there is visual representation of the wire.

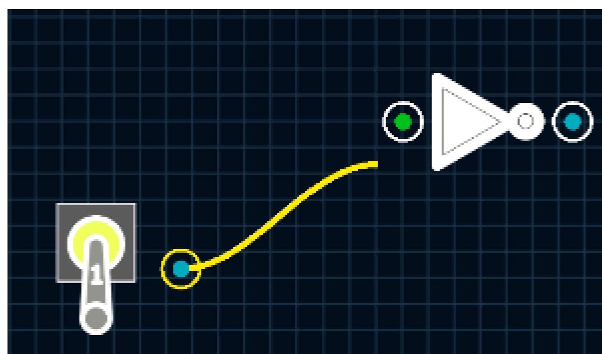
As you drag the mouse with left click around the scene, the wire updates its position to where your mouse is currently pointing at. When you stop dragging the mouse it detects if there was another eligible element that the input can be attached to. By eligible meaning that it wouldn't make sense that an input can be attached to another input, these restrictions are always checked. If there is none, the script resets its own values, but if there is and it's eligible then the input detects that you are trying to connect it to an output and performs the following actions.

- Updates its own references, so the input knows that it's connected to an output.
- Updates the references of the attached output, so the output knows it's also connected to an input.
- Raises a flag for logic check. This will be clarified later when we dive deep into the logic management of the simulator.
- Sets the LineRenderer values and enables it for the duration of the connection.
- Updates the LineRenderer color depending on the logic value.
- Updates the ring visual feedback.

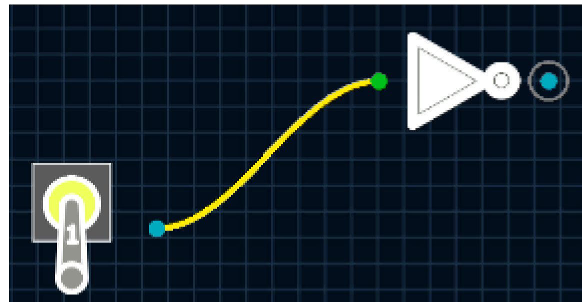
In case that it's an output, the situation is a bit more complex, because instead of having one reference it will have multiple of them. It makes sense that an input is connected to only another element, but not an output, output can be connected to as many elements as needed. So the script does all of what I mentioned earlier, but instead of updating the reference of its only connection, it updates it for all the connections that are kept in the OutputsList.

If the user wants to remove an already made connection, simply clicking the input again will delete it and reset all the values including whatever it's connected to.

Example:

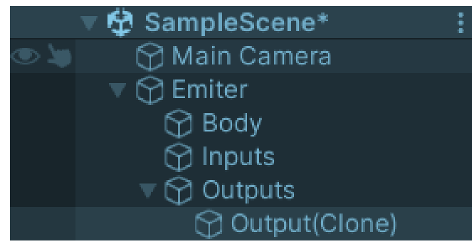


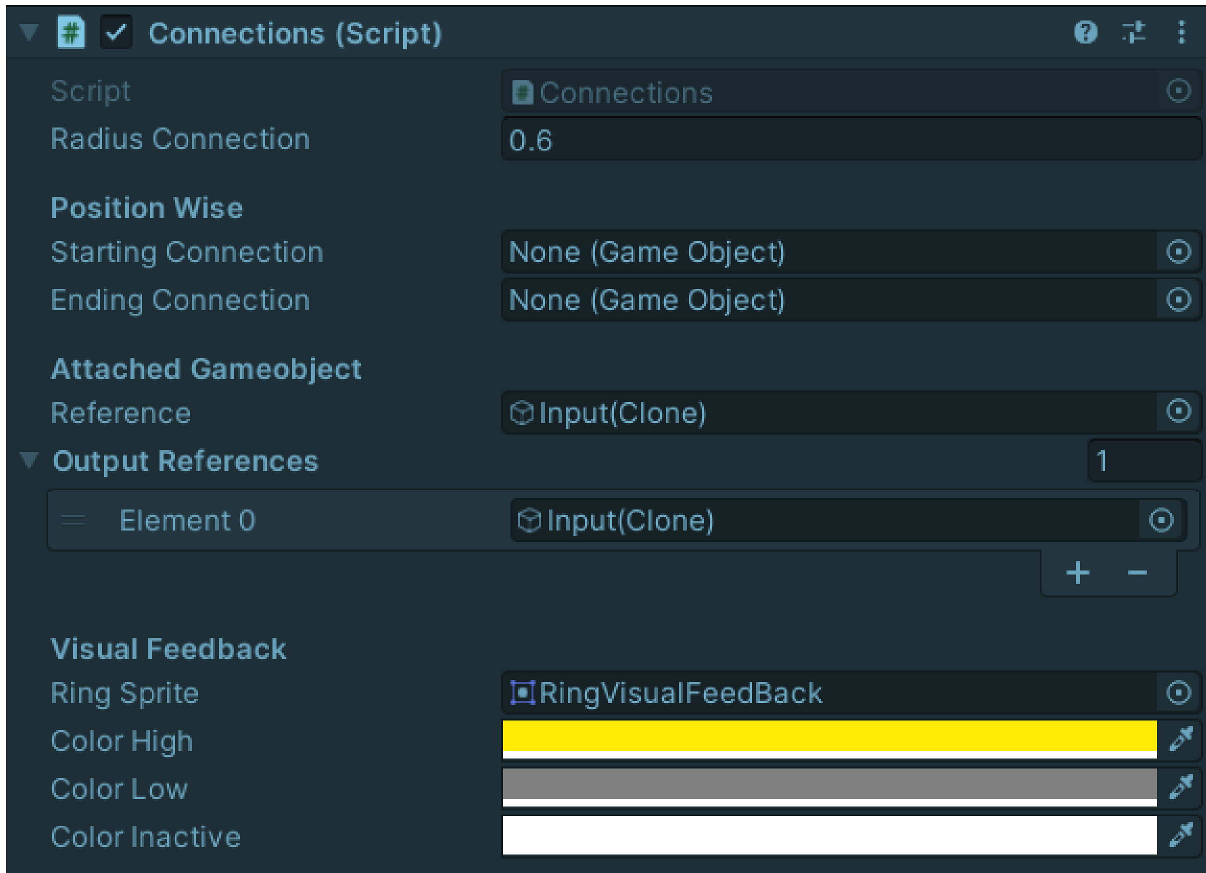
Here we are dragging from an INPUT gate in a high state to the input of a NOT gate. The ring visual feedback on the attachments is only there to show that the attachment is still not connected.
As shown the NOT gate has not been connected yet so its input and output are in the white color showing that there is no logic being passed/passing.



Now the connection is made and the rings have disappeared. Another change we can appreciate is that the color of the NOT gate output has been updated, from white to gray, which is correct, but the ring is still there because it's only showing the connections that have yet to be connected.

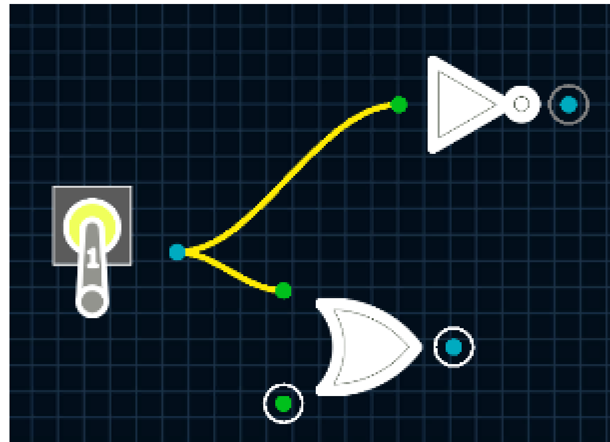
Now let's take a look at the inspector window of the INPUT switch gate output attachment.





We can see some parameters that we have already mentioned like the connection radius, or the visual feedback parameters. To give a detailed explanation of all the parameters we can see here, the StartingConnection and EndingConnection are just parameters to be shown for debugging porpuses.

The interesting part of the script is the Reference and the OutputReferences list. As we can see, what we are looking at is the Connections.cs of an output. So we must see that the list has one element, Input(Clone), meaning that is keeping the reference of the NOT gate input. Let's see what happens to the Connections.cs when we connect another gate to the same output.

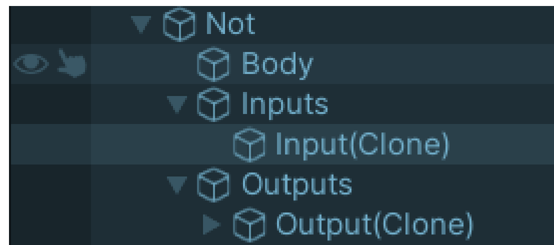


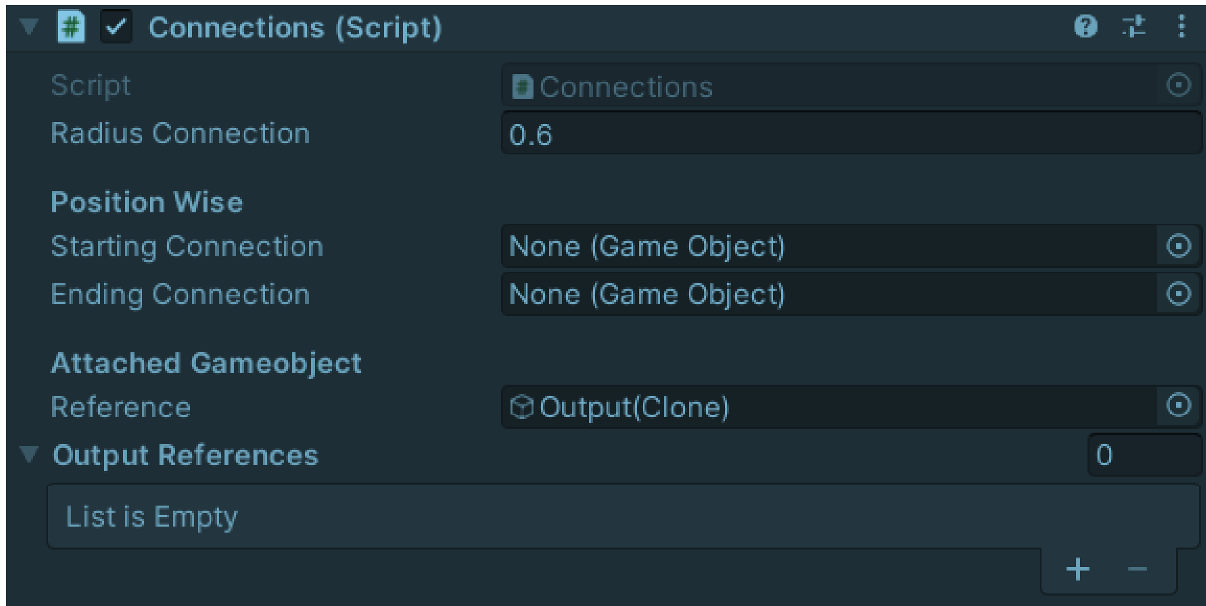
Connections (Script)

Script	Connections
Radius Connection	0.6
Position Wise	
Starting Connection	None (Game Object)
Ending Connection	None (Game Object)
Attached Gameobject	
Reference	Input(Clone)
Output References	2
Element 0	Input(Clone)
Element 1	Input(Clone)

As we can see, the OutputReferences list now has two elements, apparently named identically, but both these Input(Clone)s are one from the NOT gate that we had before, and the second one is the input from the newly created OR gate.

If we now check the Connections.cs from the NOT gate input we have:





In this script we can see that the Reference has an Output(Clone) which is the reference of the attached output on the INPUT gate. If we check the newly created OR gate, the script looks exactly like this one.

If we take a connected wire and make it disappear, its references would be updated at runtime as well. So we can see how to connect the wires, and how the system keeps track of these connections. So if we were to disconnect one of the connections, the Door we are taking from the connection would overwrite all the involved Connections.cs scripts and take itself out of the tracked attachments.

So far we have discussed how the Connections.cs manages the relationship between the gates. It's important to insist on the fact that this custom script doesn't have anything to do with logic. The script does all of the above, and of course triggers some flags for other scripts to identify, but that's all. This script is not responsible for logic management.

Now, after knowing what this script does, we can dive deeper into the other components that make an Input/Output Prefab.

6.4.2. Wire custom script

The wire script is in charge of managing the specifics of the LineRenderer component. It only holds methods that will be called through Connections.cs such as enabling or disabling, drawing a bezier curve between two given points, changing the color of the wire, the thickness, type, materials etc.

6.4.3. Gate Logic custom script

This script is just a class holder for the definition of the logic states, and a reset method, setters and getters.

6.4.4. Door Logic custom script

This script manages the logic part of the simulator. It's located on the root of the prefab. So it has both access to all of its inputs and outputs.

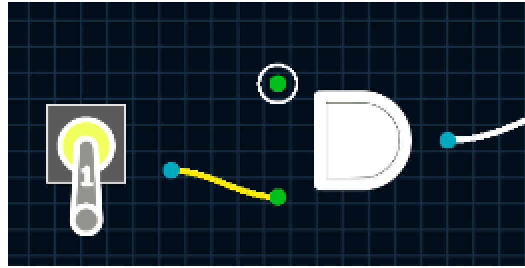
Its most important features are:

- Manage active/inactive simulation status. This feature is currently not enabled in the final project, but I thought it would be a nice idea in order to make this script be ruled by this condition in case in the future I want to expand the project to take into consideration sequential circuits.
- Keep track and assign the intended logic state of newly created inputs and outputs (by DoorManager.cs).
- Clear own and attached gate's logic values when a reset flag is triggered by DoorManager.cs.
- Respond to Connections.cs flag triggering, so that every time there is a newly made connection in its own gate, it will attempt to perform the logic of the gate. The conditions are that all the inputs of the gate must be connected, and have a state different from inactive. If that's the case, then the output will be updated with the high/low state.
  The information needed to perform the logic will be taken from DoorManager.cs.
- Update the visual representation,. For INPUTS and OUTPUTS gates it will be updating the sprites held on the Body. As well as updating the output's wire color and the ring visual feedback.
- Update the logic on the input of attached gates. So if the logic check is true, meaning that the door is conducting logic, it will update the logic of the attached elements of other doors.

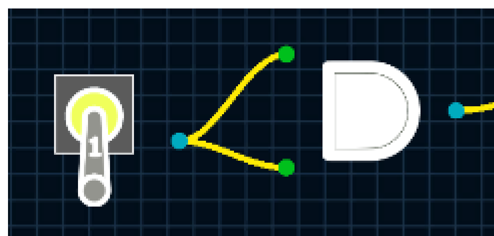
Sprite update example:

When toggling an input, its output value goes from low to high, never inactive. So when we toggle it with a left mouse button click, the sprite updates depending on the output state. We can see, by the ring visual feedback, that the blue output is "conducting" a low or high state.

In this other image, we see multiple features:

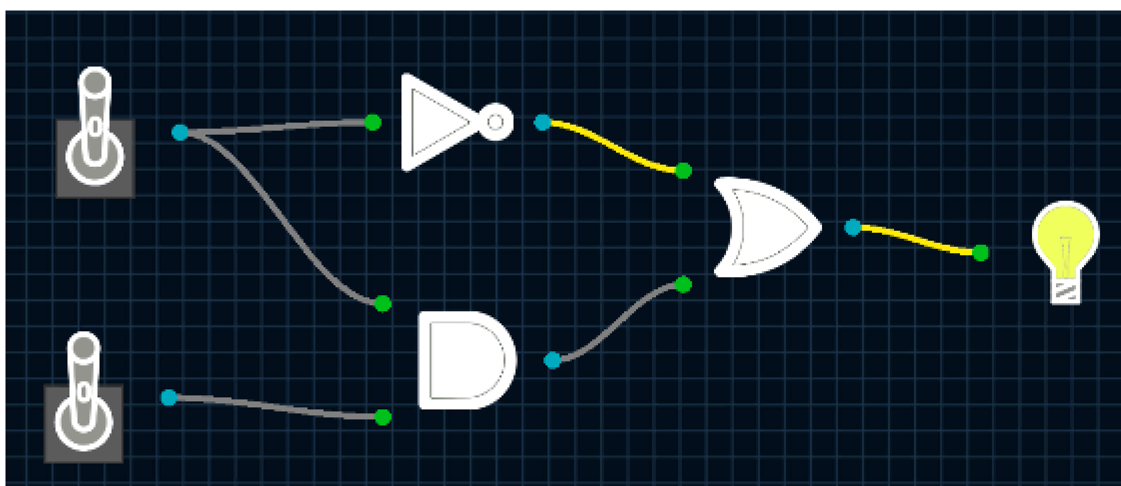


We see that the INPUT is giving off a high signal that is received by the input of an AND gate. But the other input of the gate is not connected, meaning that it shouldn't give off an active signal on the output. So even though the AND output is connected to something else we can't see on the picture, we see the wire white, meaning that there is no signal.



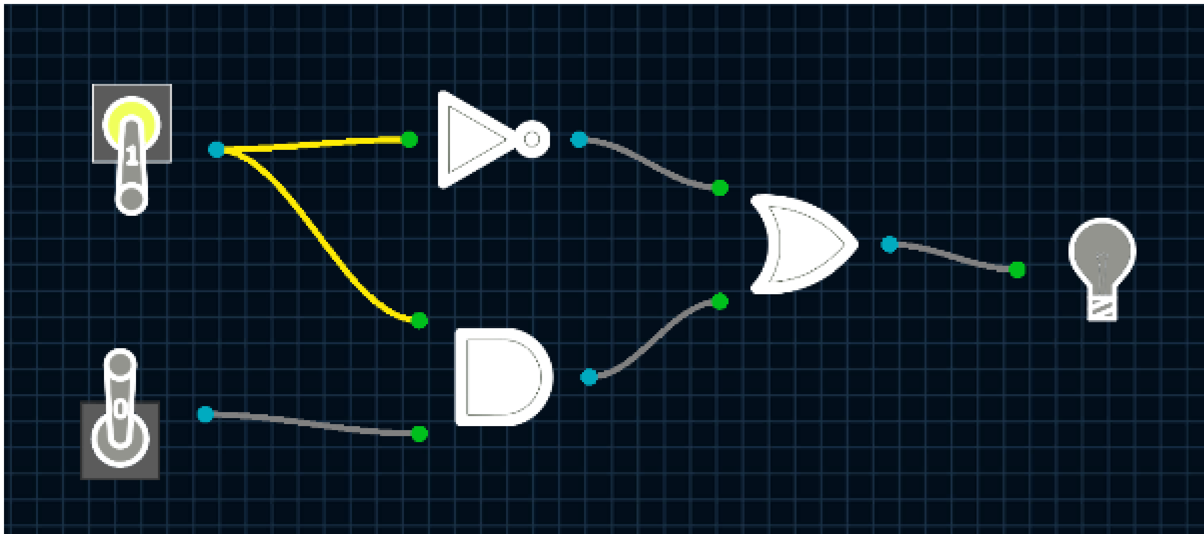
In this other image, after connecting both inputs, we see that the AND gate is now passing a high signal to the connected gate's input.

Gate overwrite feature:



In this picture, we can see a simple digital circuit. As we can see everything is connected and giving off a signal, either low or high, so all the colors we can see are either yellow or gray.

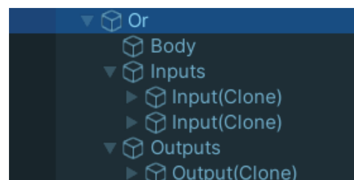
If now we toggle one of the inputs:

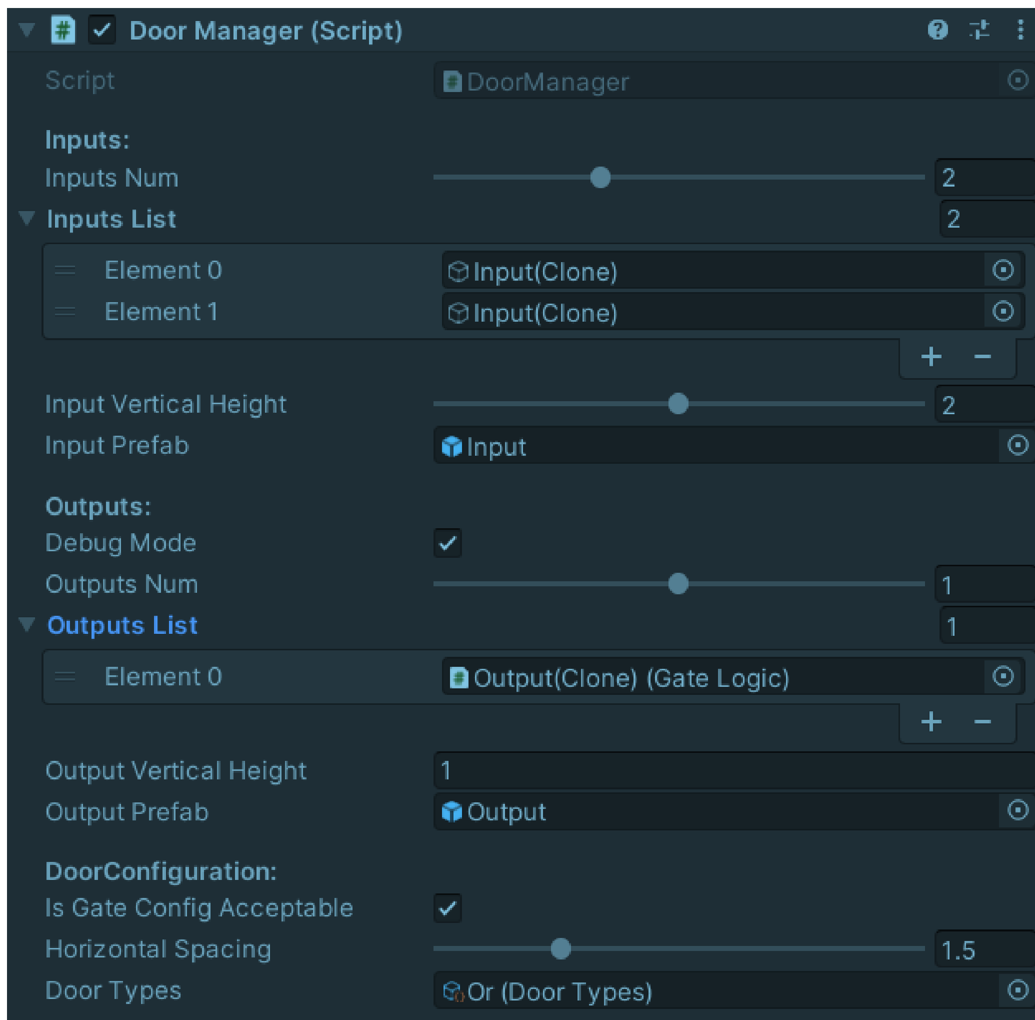


We can clearly see how just toggling one of the inputs, updates the color of the attached wires, this creates a chain reaction to the other gates. It might not be obvious so far but every single element we see on this scene is the same one, but with different parameters and conditions. So, as long as the same object is designed to be able to overwrite other object's logic states, the chain reaction will occur so we can make the digital circuit as complex as we need. The logic behind it will be as complex with two elements as with fifty.

6.4.5. Door Manager custom script

This script is the most complex script of all. Instead of giving a generic explanation of its features it's better to start with the parameters set on the inspector. This script is located at the prefab root, just like DoorLogic.cs:





- We start with the number of inputs, which is set using a slider. You can set it up to a limited number. By default, the configuration of the OR gate is set to 2 and its maximum 5.
- Inputs list holds the references of the Input objects of its own gate.
- Input Vertical Height stands for the difference in height between the inputs. There was initially an undesired behavior when having too many inputs in the same door, when you were about to click on one of them, the detection wasn't too great. To make the simulator user friendly you need to have the parameter on the Connections.cs called Radius Connection, bigger than its dot radius, otherwise it would be too difficult to click. So the circle of how the input detects the mouse click is bigger than the actual dot representation. When two of these inputs are too close together there is a conflict that the script resolves taking the connection closer to the point. But in practice this is not the best user friendly, so to solve this undesired behavior this parameter got introduced. So by timing with this parameter you can set it to the optimal distance between connections so the mouse click doesn't overlap with each connection.
- DebugMode is used for debugging and Outputs Num is currently not enabled in the project. It's there in case I want to expand the project to accept sequential circuits so this parameter will start becoming relevant when adding flip-flops for instance. By now it's disabled and set to one.

- Outputs List is very similar to the inputs list. It keeps track of the elements of its own gate. So it holds the reference of the output. The fact that instead of keeping an object keeps a script is just a matter of compatibility. It holds no relevance.
- Input Prefab and Output Prefab holds the references to the Prefabs (not in scene) needed to be instantiated.
- Is Gate Config Acceptable is just a Debug parameter.
- Horizontal spacing is just a custom offset and moves the inputs/outputs X coordinates within the door further or closer.
- DoorTypes keeps the reference of the selected door type that the gate will be at runtime. It holds the type of gate that the object will be. This type of information holder is called ScriptableObject.

Most of the values mentioned above will be modified at runtime. So most of them are just shown in the inspector for debug purposes.

The process of how this object called “Door” transforms to an actual gate is a bit complex. To explain it easily we’ll go through the features that the script holds and makes use at runtime.

When this object gets enabled in the scene, which means that it is instantiated whenever it’s added or whenever the simulator starts, it will first check for incompatibilities of the parameters in the inspector. In this way that it will prevent it from being enabled without proper gate configuration, meaning it will check impossibilities like a NOT gate being instantiated with two inputs for instance. Then it will assign the default set up values specified in the selected DoorTypes ScriptableObject. It only holds methods and parameters that will mostly be read by this DoorManager.cs.

This was just the set up, then every frame it will check whether one of those parameters has changed. Let’s say that the number of inputs has now changed, or it’s not a NOT anymore and it has become an OR gate, or maybe a change in the vertical or horizontal spacing.
If there has been any change regarding the configuration, the door will reset itself to adapt to the new values.

This “reset” is also the one made when I instantiate the gate and by order will:

- Get the existing inputs and outputs. Then it will read if these inputs or outputs have any references attached (if these are attached to another gate). So basically if it’s an input nothing will happen but if it’s an output, it will get the OutputsList of its own output, then it will reset the logic and connections of that other attached gate, so if there was an attached gate passing logic, it will get updated,
- After the reference reset is done, it will destroy its own inputs and outputs.
- Next it will instantiate new inputs and outputs depending on the new gate configuration. If it’s a NOT it will spawn one input and output but if it’s an EMITTER gate it will only spawn one output. Then raise a flag for the DoorLogic.cs to read and set up its logic state to high, and update its sprite.
- Then it will update the Body sprite matching to the selected gate type. So if it’s an OR it will update its shape with a new sprite.
- Then it will update the existing collider to the new shape. We mentioned earlier that what lets the user click stuff is the collider. So it wouldn’t make sense to have the

area of where you click a NOT shape, a triangle when you are actually trying to click an OR for instance. So it updates the PolygonCollider2D of the Body to match the new shape.

6.4.6. DoorTypes ScriptableObject custom script

This type of script derives from ScriptableObject which is a base class that is used to keep information. This information can be accessed regardless of the scene state. So basically in this script we will keep most of the possible configuration and default parameters that are accepted depending on the gate type.
This class defines each kind of door type that exists in the project. It also keeps information like the sprite that is used to represent it and update its body, the acceptable input and output range number, the default number, etc.
It's important to keep in mind that it doesn't hold parameters such as input / output references. This script will be accessed by every single other gate in the scene. So its information must exclude parameters like a buffer, or that holds information about instantiated objects.

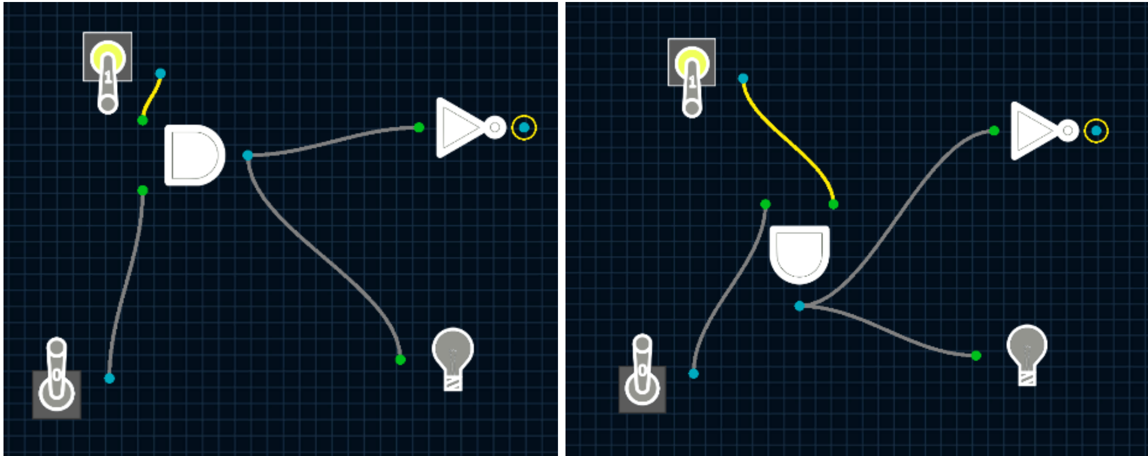
This script also holds the code used to perform logic checks. Meaning that the already mentioned DoorLogic.cs script takes the information from this script to run its logic checks. Basically it will read that if in that moment it's a NOT gate, it will take this scriptableObject, read through the NOT gate logic method, pass it certain parameters, run it, and get a result.

6.4.7. Dragger custom script

This script is attached to the body of the gate. It gets triggered when we drag and drop the gate around the workspace. So all its features will be surrounding that event.
Most important features:

- First and most obvious, it changes the position of the dragged gate around the scene. To achieve the desired behavior there is also the consideration of an offset (between the actual gate position and the mouse click). So the updated position takes into consideration that you are dragging the gate from a specific spot on the collider.
- When we press on a gate body to drag it around the scene, the script reads through all of its own inputs and outputs looking for attached references. Then when we drag it around, the wires of those references also get updated according to the new position. Otherwise we would be able to move the gate around but one end of the attached wires would remain in the initial position, and that's not a desired behavior.
- Also when we press R while dragging, the object turns around with all of its wires to a new rotation and position as well.

By means of a graphic demonstration of dragging and dropping the gate around the scene, we can see how the connection's positions get updated with the new relative position to the body and how after pressing R (second image) we see the rotated object with its wire also responding to that same rotation and updating its wire state.

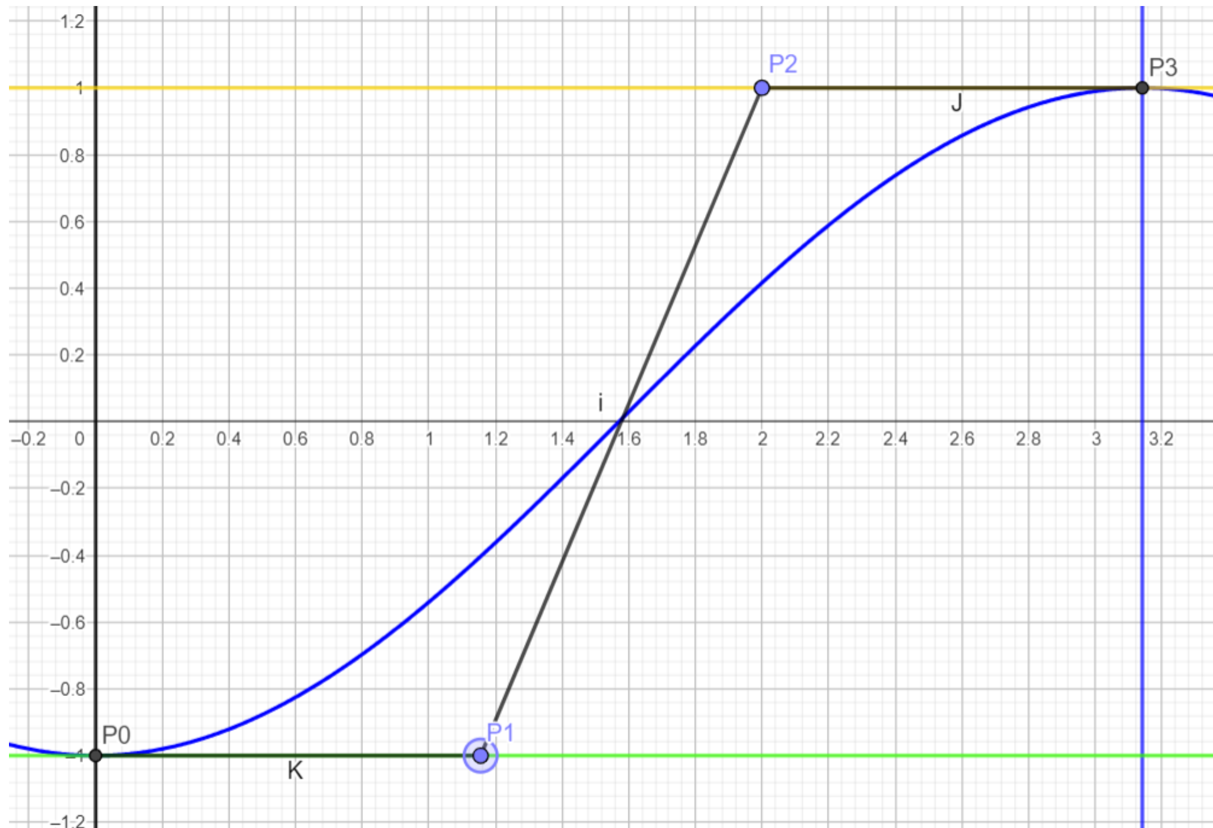


6.4.8. Bezier Line Renderer Controller custom script

Bezier curves are parametric curves used in computer graphics, image editing and vector-based drawing programs. They are defined by a set of control points that determine the shape of the curve, the most commonly Bezier curve used is the quadratic one, defined by three control points. After a few trials I realized that I wasn't getting the shape that I desired and it didn't look natural. It was too complex and it really didn't give off the impression that the line was emulating a wire. So I decided to use the cubic one with two control points (without counting the ends of the wire).
The wire needs two points to be created, but I wanted a half sinus behavior for the bezier curve to make it feel like an actual wire instead of just a straight line between two points.

To represent a bezier curve we need four points P0, P1, P2 and P3 on the plane or in a higher-dimensional space define a cubic curve. The curve starts at P0 going toward P1 and arrives at P3 coming from the direction of P2. Usually, it will not pass through P1 or P2; these points are only there to provide directional information. The distance between P1 and P2 determines "how far" and "how fast" the curve moves towards P1 before turning towards P2.

We want a cubic Bézier curve made out of points P0(0,0), P1(0,K), P2(1-K,1), P3(1,1) that approximates the shifted sine curve ($y = a \cdot \sin(bx + c) + d$) which has its minimum at (0,0) and maximum at (1,1).
So basically P0 and P3 determine the end points of the curve.



We need to find the value of K for which the derivative of our Bezier curve matches the one of the sinus at t=0.5, which would be π/2.

The derivative of the cubic Bézier curve with respect to *t* is

$$\mathbf{B}'(t) = 3(1-t)^2(\mathbf{P}_1 - \mathbf{P}_0) + 6(1-t)t(\mathbf{P}_2 - \mathbf{P}_1) + 3t^2(\mathbf{P}_3 - \mathbf{P}_2)\,.$$

And we know all points except the K parameter.

$$\frac{dy}{dx} = \frac{dy/dt}{dx/dt} = \frac{d(3(1-t)t^2+t^3)/dt}{d(3(1-t)^2tK+3(1-t)t^2(1-k)+t^3)/dt}$$

We know that at t = 0.5 the expression simplifies to:

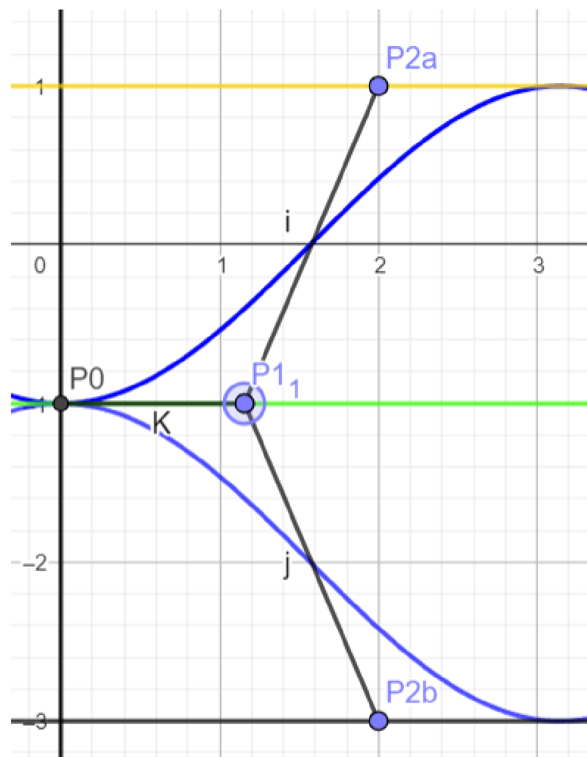
$$\frac{dy}{dx} = \frac{1}{1-K}$$

$$K = \frac{\pi-2}{\pi} = 0.36338$$

So now we have the values to create this half sinus wire. All we need to do is to create a sample and give the line renderer each individual point. I decided the number of points to be 20, so the script will spread those 20 points through this half sinus behavior. Knowing K it's just a matter of creating a bezier curve through code, giving it the custom points and calling that method 20 times per frame with an increment that makes up for the total distance between P0 and P3.

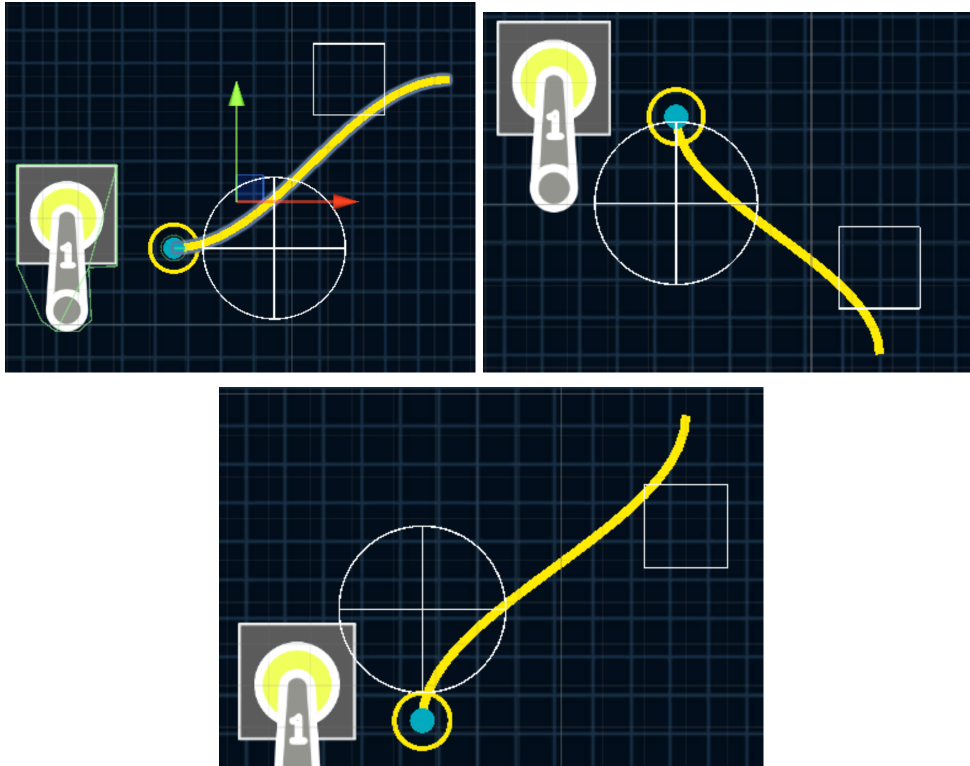
There is another issue that comes up after and needs to be solved. The issue is that the wire doesn't take into consideration its relative position to the gate we are dragging from. This issue causes undesired behavior. If we are dragging a wire from an output (meaning the gate is on the left) and we drag it to the left, the first control point will be located on the -x axis overlapping the gate. To avoid this behavior we have to establish that the control points must be different depending on where the P3 (mouse position) is and the P0 (element dragging from) are.

So basically to make it look symmetrical I divided the 2D plane into 4 sectors and calculated the angle between the points. The control points will be placed depending on the relative position of P0 and P3.

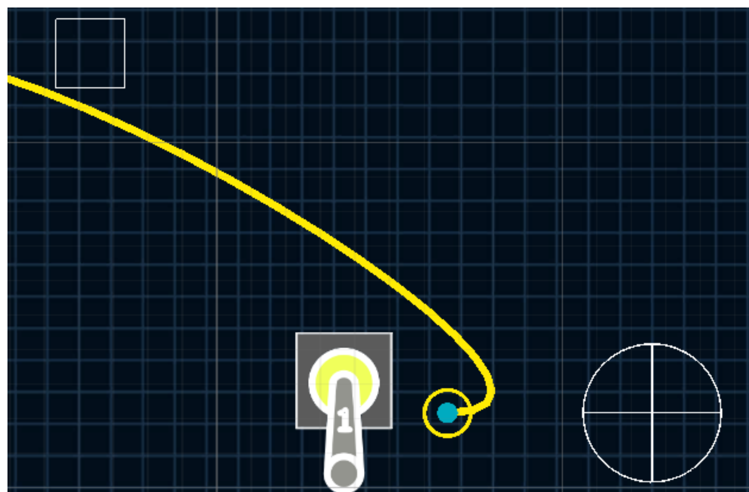


In this picture there is a visual representation of one of the four sections I divided the space into. This section goes from -45 to 45 degrees. In this section, it doesn't matter where the mouse position is. If its y value is greater than 0 then it will use the second control point P2a, and if it's below than 0 it will use P2b. Regardless of this, the control point P1 will remain at the same positive x axis. But when the angle between the mouse position and the P0 (attachment point) goes beyond these limits then the control points will be defined by another sector's parameters.

In the following pictures, the blue dot is P0, representing the output, the circle represents the control point P1, the square represents the control point P2, and the end of the drawn white wire is the P3 (mouse position).



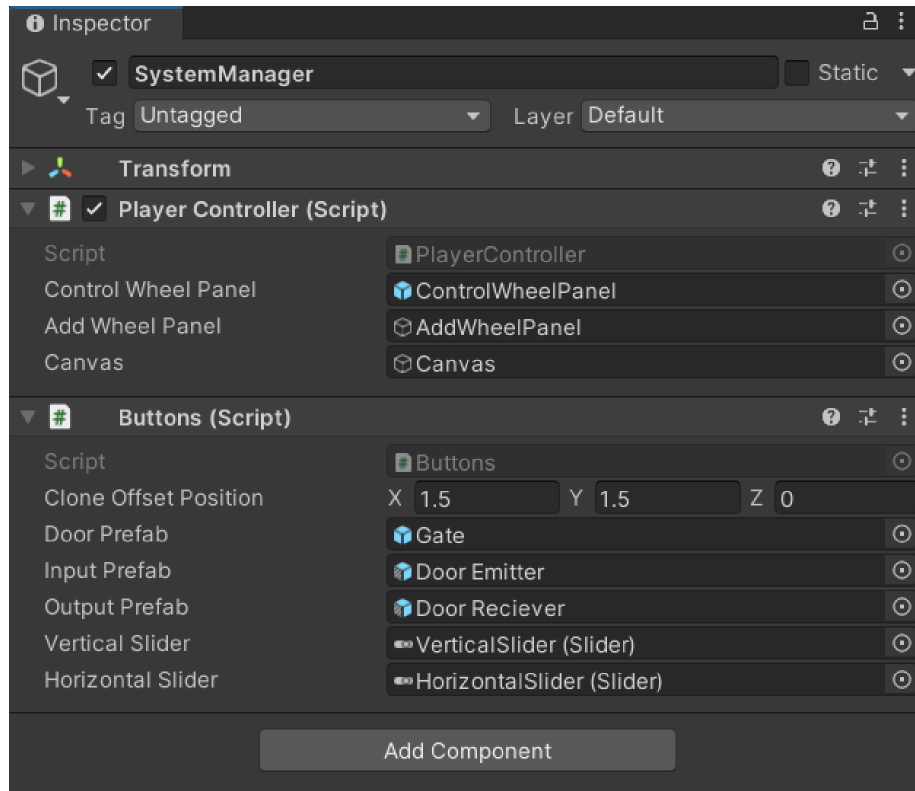
As we can see, the circle (control point P1) is located in different places in relation to its P0. So if we drag it to the left, the gate is on the left side of the output, so it wouldn't make sense the control point 1 to be located on the left side of the output, because the gate body is there and it would overlap. This is controlled through code so even if you rotate the object it wouldn't overlap anyway. In this position the control point P1 will always be on the right side of the output P0.



As we can see here, the P1 is on the correct side, so the wire bending looks as initially intended. If the wire too low resolution, we can just add more simulation points. The current value is set to 20.

This has been an explanation of how the gates operate in the existing scene at runtime. It shows all the implemented features as well as how they operate behind the scenes. But another important part of the project is the user interface.

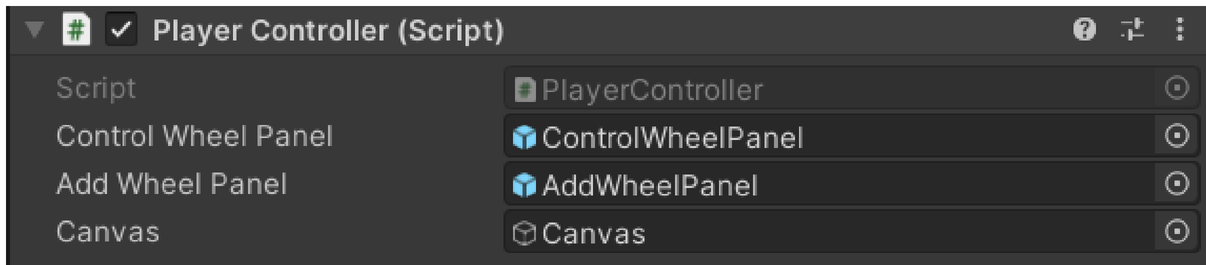
To make the user interface work, there is an empty GameObject in the scene that is only there to be able to respond to user input. This element holds two key custom scripts.



6.4.9. Player Controller custom script

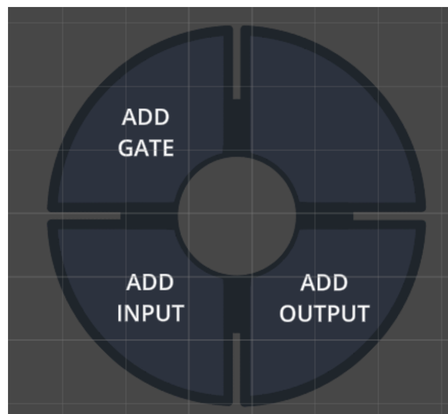
This script directly triggers flags for user controls. When pressing the left or right button, it triggers different flags. To make the user experience I earlier mentioned in the UX design that the simulator should be easy to operate. So basically when the user clicks the right button, it triggers either the “Add Wheel Panel” when clicked on the workspace background or the “Control Wheel Panel” when clicked on an existing gate. So if the player wants to add a gate, it will right click on the background, and select either an input, output, or a standard gate, which will be a NOT by default.

When any panel is enabled, it will always be located right on the mouse position.



- “Add Wheel Panel” behavior. When triggered, any other existing UI element will get disabled, and it will be rendered on top of any other existing element. It will have three buttons. Each of them will add the same gate element but with a different configuration.

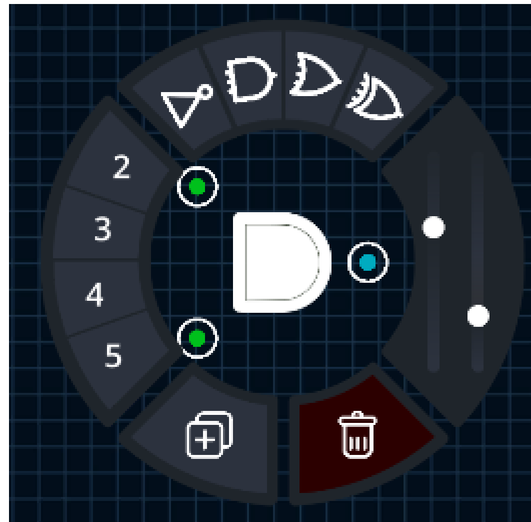
  Visual representation of the “Add Wheel Panel":



When clicked at one of these three buttons the desired behavior is performed (gate instantiation) and then the wheel will get disabled.

“Control Wheel Panel” This wheel has different buttons that perform different features. It is divided into four different segments that hold different sets of functions:

- The top segment holds the buttons for the basic door types. So it will be used when the user wants to change the current gate type.
- The left segment holds the number of desired inputs. So it will reset the door to match its desired configuration.
- The right segment holds two sliders in charge of changing the height between inputs and the width between body and inputs / outputs.

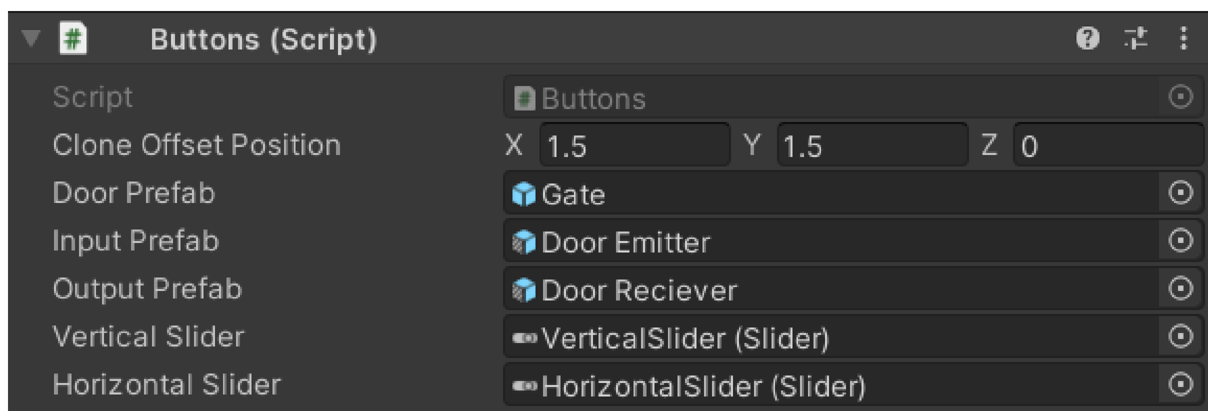


The left, top and right segments won't make the wheel disabled when pressed. This is important because if the user wants to change more than one parameter at once he can do it all at a time triggering the appearance of this user interface.

The bottom segment is divided into two smaller segments, one is to duplicate the selected gate on the scene. The other one is to delete the current gate.

After pressing one of these buttons, the wheel will get disabled.

6.4.10. Buttons custom script



This script holds all the methods that will be called when one of the buttons is pressed or when one of the sliders' values are changed on both wheels. The "Add Wheel Panel" and the "Control Wheel Panel".
Basically, the methods from DoorManager.cs just raise a flag when the buttons are pressed and then the script just behaves similarly than when it resets a door, but with the new parameters.

7. Conclusions

After creating this combinational logic simulator from scratch there are certain topics I would like to address:

- Regarding the capabilities of coding and learning the engine from scratch, I must say that with the help of many free resources online and university books and with the help of my advisors the journey hasn't been as rough as I had imagined. Unity has high quality documentation available for everyone and backs it up with practical examples of the features that it presents.

- Regarding scalability, this project was initially thought to be only for combinational circuits, but taking into consideration the possibility of expanding it in the future to sequential circuits. With the radial user interface that I have currently designed, it's not possible to keep up with that expansion. The reason is that a radial UI is not suitable for an ever expanding number of elements. So if this project gets revisited in the future, this issue will need to be addressed.

- Regarding the coding learning curve, as I mentioned earlier, the journey hasn't been as hard as I thought, but I was learning to code while I was coding the simulator. There are a number of things I would definitely do differently now, but the most important is the dependencies in code. From my little experience I can tell that it can get out of hand very quickly and it's important to keep your code only doing and accessing the only and necessary classes to operate. The scale of this project is only of one person, but if this project had to be done in a team, much of the code would need to be refactored.

- Regarding the capabilities of the Unity engine used, there are many features that I still don't know how to use. The tool is great to create any kind of virtual experiences, from a wide range of anything you can imagine. It works well, and has great online assistance by the team and is ever expanding with new features. As I am writing this document, there have been many updates of the software. I started on version 2021.3.15 and now it's at 2022.3.3. The new versions expand its features even more.

- Regarding functionality, the software doesn't disappoint. It's quick to learn and use, visually coherent and responsive. After a process of debugging I got rid of all the major bugs. But since I lack any kind of feedback from users I can say for certain that there must be some minor bugs. The resulting product matches and fulfills all of the objectives that I had set in the beginning of the product design. It's quick to use, meaning that it's very easy to navigate the tool, get a hang of its main features and make it work. It's also intuitive, and it's also working accordingly.

8. References

Bowler Hat LLC. (n.d.). *Logic Circuit Simulator*. Logic.ly. Retrieved August 2, 2023, from https://logic.ly/demo/

CircuitVerse. (n.d.). *Digital Circuit Simulato*. CircuitVerse. Retrieved August 3, 2023, from https://circuitverse.org/simulator

Cristian Born. (n.d.). *Logic Circuit Simulator*. simulator.io - Build and simulate logic circuits. Retrieved August 3, 2023, from https://simulator.io/

LogiJS. (n.d.). *Logic Circuit Simulator*. LogiJS: Logic Circuit Simulator. Retrieved August 3, 2023, from https://logijs.com/

Neusesser, T. (2023, March 26). *UX Basics: Study Guide*. Nielsen Norman Group. Retrieved September 5, 2023, from https://www.nngroup.com/articles/ux-basics-study-guide/

Unity Technologies. (n.d.). Plataforma de desarrollo en tiempo real de Unity | Motor de VR, AR, 3D y 2D. Retrieved September 5, 2023, from https://unity.com/es

Unity Technologies. (n.d.). *Prefabs*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/Manual/Prefabs.html

Unity Technologies. (n.d.). *ScriptableObject*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/Manual/class-ScriptableObject.html

Unity Technologies. (n.d.). *Scripting API: Canvas*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/ScriptReference/Canvas.html

Unity Technologies. (n.d.). *Scripting API: GameObject*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/ScriptReference/GameObject.html

Unity Technologies. (n.d.). *Scripting API: LineRenderer*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/ScriptReference/LineRenderer.html

Unity Technologies. (n.d.). *Scripting API: Object.Instantiate*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

Unity Technologies. (n.d.). *Unity - Scripting API: Collider*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/ScriptReference/Collider.html

Unity Technologies. (n.d.). *Unity - Scripting API: Sprite*. Unity - Manual. Retrieved September 5, 2023, from https://docs.unity3d.com/ScriptReference/Sprite.html