

VR Logic Gates

Laura Choy

Supervisor: Dr Parisa Eslambolchilar

May 2022

CM3203 Individual Project – 40 Credits

Abstract

The main goal of this project was to create a simulation of Logic-Gates within a VR environment that could update in real time and used modular components. Its concept was designed to give visual representation of logic circuits as well as provide spatial awareness to components that take physical space. The project is a single user application that can be used on both tethered PC VR headsets and standalone VR headsets.

Acknowledgements

I would like to acknowledge my supervisor, DR Parisa Eslambolchilar for her guidance and suggestions. The project would not have gone ahead had she not believed in its credibility and design.

Secondly, I would like to thank Unity and its community for its documentation and tutorials. A lot of skills were required to complete that project that I did not possess and it was thanks to the learning tutorials and videos that the project was able to be completed.

Table of Contents

Abstract	2
Acknowledgements	2
Table of Contents	3
Table of Figures	4
Introduction	5
Background	7
<i>Existing logic circuit simulations</i>	<i>7</i>
<i>Core design merits</i>	<i>10</i>
<i>Examples of existing VR logic circuit simulation</i>	<i>11</i>
<i>The Unity engine</i>	<i>13</i>
Approach	15
<i>Requirements</i>	<i>15</i>
<i>Design aspects</i>	<i>16</i>
<i>System requirements</i>	<i>17</i>
<i>Project architecture</i>	<i>17</i>
Implementation	19
Video demonstration	30
Results and evaluation	30
<i>Project outcome</i>	<i>30</i>
<i>Description of testing procedures</i>	<i>31</i>
<i>Testing against requirements</i>	<i>31</i>
Future work	33
Reflection on learning	35
References	37

Table of Figures

Figure 1 – Logic Gate Simulator: Inserting nodes	7
Figure 2 – Logic Gate Simulator: Input / Output visualisation	8
Figure 3 – Logic.ly: Inserting components	8
Figure 4 – Logic.ly: TRUE value visualisation	9
Figure 5 – Logic.ly: Dependency error checking	9
Figure 6 – Logic.ly: Truth table generation	10
Figure 7 – VR Logic Circuits Laboratory: Screenshot of simulation from paper	11
Figure 8 – VR Logic Circuits Laboratory: Extract of conclusion from paper	11
Figure 9 – Zen Marmot Digital Logic Simulation: Video demonstration 2 NAND gate	12
Figure 10 – Zen Marmot Digital Logic Simulation: Video demonstration 2 scenario queue	12
Figure 11 – Zen Marmot Digital Logic Simulation: Video demonstration 2 project description	13
Figure 12 – Project Diagram: Scene diagram	18
Figure 13 – Project hierarchy scene template: XR Origin	19
Figure 14 – Locomotion system: Continuous movement	20
Figure 15 – XR Origin: Virtual hand placeholders with colliders	20
Figure 16 – XR Origin: XR Ray Interactor	21
Figure 17 – Empty button prefab: Button game-objects	21
Figure 18 – Button controller script: Event invoking	22
Figure 19 – Straight Wire Prefab: Child game-object power components	23
Figure 20 – Output script: Refencing connection based on collision detection	23
Figure 21 – Output script: Updating power output to connected inputs	24
Figure 22 – AND gate power controller script: Evaluating signal output based on inputs	24
Figure 23 – Snapping script: Rotation and position rounding	25
Figure 24 – Snapping script: Executing on select exited for XR Grab Interactable	25
Figure 25 – Active emission script: Enabling keyword _EMISSION	26
Figure 26 – Wire input script: Broadcasting glow activation function to child game-objects	26
Figure 27 – Straight wire: Glow effect comparison	26
Figure 28 – AND gate puzzle evaluation: Checking state passing	27
Figure 29 – Truth table updater script: Setting states then invoking for buffer	28
Figure 30 – Truth table updater script: Checking output states and changing cell values	28
Figure 31 – Marker script: Drawing to a wall with tag “Board”	29

Introduction

This section explores the following:

- Overview of the project design
- Explanation of project uses and target audience
- Justification and elaboration of design choices relating to scope and function

Logic gates are one of the fundamental concepts of computer science and is taught in many courses such as Computer Science, Software Engineering and Electrical Engineering. Within these courses, the theory behind logic gates is presented through truth tables and diagrams. Whilst other topics such as programming have exercises and tutorials for the application of programming theory, logic gates cannot have application unless working with physical components.

The visual representations of logic circuits through diagrams, tables and logic statements all operate on the assumption that the users of the visuals can recreate the state of the information mentally. The diagrams provide no output and create no information unless a set of inputs is provided. Similarly, a truth table only represents the states of a given circuit, not necessarily the configuration required to reproduce the set of states.

Extensionally, components always take physical space. Central processing units are ultimately comprised of many logic gates constructed in a way that allows processing of instructions. As each logic gate is technically an electrical component and takes up space within the unit. Part of the design problem with central processing units is the limited space as well as the requirement of non-intersecting components.

The project aims to provide a 3D visualised representation of logic gates within a VR environment. The simulation is interactive and shows state of the circuit through lights and arrows using mesh representations of logic gates and wires. This is coupled with a modular design system which allows the user to re-orient the wires and gates at their will to restructure circuits and experiment with designs. With the 3D representation, wires and gates cannot intersect and thus the simulation provides tangible layouts of logic circuits without paradoxical use of space.

The target audience for the simulation are students studying computer science or similar subjects in the GCSE curriculum or above. This project is designed to be a visual learning aid for students learning the concepts of logic gates. It is not a substitute for theory education but serves as a hands-on approach to learning and developing skills such as spatial awareness and logic engineering.

Scope of the project is narrow but allows for constructive use rather than explicitly guided use. The project has two core areas, a sandbox area, and a construction puzzle area.

The sandbox area provides the user with all the gates and wires and a construction area as well as a few inputs and outputs. It is designed to allow users to experiment with configurations at their own pace and visually see how their changes affect the circuits. The area also provides a self-writing truth table that can be updated from user input. Once prompted, the truth table will adjust the inputs to align with the first state of the truth table. Then it will read the outputs of the simulated circuit and update the output values in the truth table. This provides the user with the ability to experiment and construct a circuit and then see exactly what the truth table looks like for that specific configuration in all its states.

The construction puzzle area is a similar but inverse section to the sandbox. Each puzzle room provides a truth table, and the goal is to reconstruct the associated logic circuit. The puzzle provides a strict boundary area so that users must reconstruct the circuit to fit within the bounds whilst satisfying the truth table. To complete each challenge, a user requires knowledge of the components they are working with and be able to orient them in a way that satisfies the area boundary.

The most important outcome of the project is that the simulation of all basic gates and wires function as expected and that they can be modularly arranged with VR input. Logic gates are small building blocks for circuits and other gates. As long as the functionality of the basic gates and wires is correct, the simulation can construct valid circuits and complex gates. An example of this would be a NAND gate being a combination of AND and NOT gates. The NAND gate can be seen as a configuration of basic gates and thus will function correctly given that the AND and NOT gates function correctly within the simulation.

The focus of the project is the simulation and components. To ensure that users are not caught up in any complexities of VR, the project will have minimal controller input consisting of locational joystick movement, grabbing button of components, and rotation joystick control of components. The rest of the project is interactable through buttons that users will use their controllers as “hands” within the virtual environment to activate. Pressing these buttons mimics how one would press a button in real life and provides distinctive sound effect feedback.

The project was also expanded to host a variety of quality-of-life features for the simulation. This included the following:

- Bin that destroys components that enter it
- Marker and eraser for writing notes on the walls
- Highlighting colours for values in truth tables during evaluations of logic configurations

Finally, the program has a tutorial section that introduces new users to the program. It involves instructions on the following:

- Picking up and fitting components together
- Spawning and deleting components
- Writing and erasing on walls
- Turning on input boxes

Whilst there are similarities between the project and a puzzles game, it has not been designed for game polish. As such, it lacks the following attributes within its design:

- Animations and particle effects
- Soundtrack or musical elements
- Polished 3D models with textures

Overall, the project is designed to be simple tool that focuses on the logic circuit simulation as opposed to a game however in future development it could be expanded into a puzzle game.

Background

This section explores the following:

- Existing logic gate simulations and their core design merits
- Examples of VR use with similar simulations
- Basic explanation of the Unity Engine
- C# scripting with the Unity Engine
- VR capabilities in the Unity Engine

Existing logic circuit simulations

Computerised simulations of logic gates are not a new concept and have been implemented as learning tools many times before. Examples of logic simulations can be found on the internet through searches such as “logic gate simulation”. An analysis of existing simulations and their similarities provided insight for key design points on which the basis of this project is founded on. Two of these online tools that this report analysed were:

- Logic Gate Simulator by Academo
- Logic.ly by Bowler Hat LCC

Logic Gate Simulator [1]:

Logic Gate Simulator is a very basic visual representation of logic circuits. It provides the ability to add nodes through a drop-down selection, which are classified as components within the circuit excluding wires. The user can re-arrange and connect the nodes by dragging them or the wire connection points.

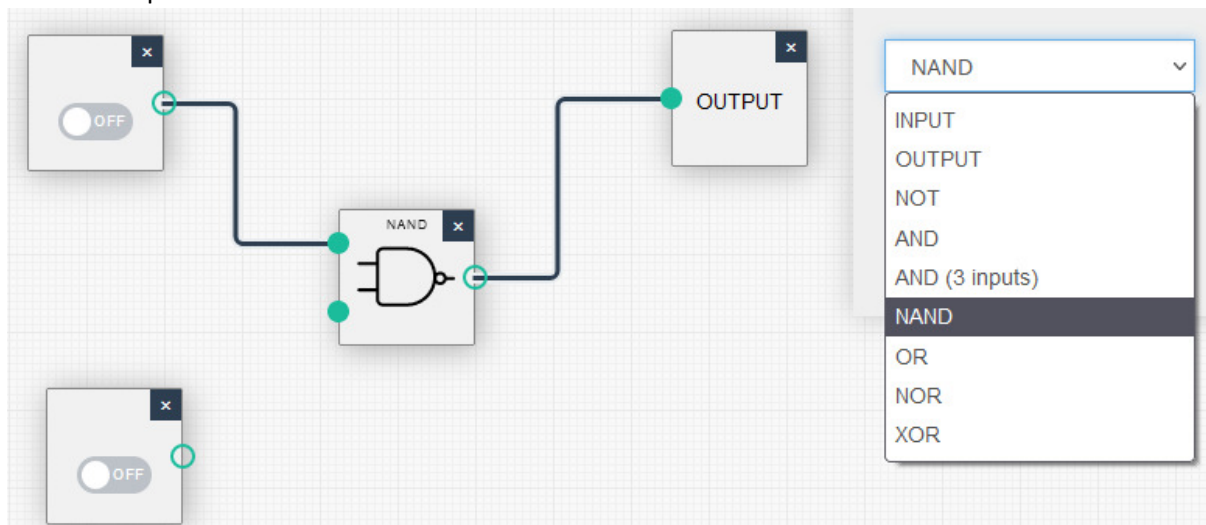


Figure 1 – Logic Gate Simulator: *Inserting nodes*

The user can only insert nodes from the list in the drop-down selection. Each node has a set of inputs depicted as circles on the left of the node and a set of outputs depicted as circles on the right of the node. Nodes can only be connected from the output side of a node into an input of another node. This limits the direction of circuits to be constructed in a left to right manner.

In addition to this, all nodes are characterised by their visual representation. Logic gates are labelled using their academically assigned symbol, inputs are shown as their state of on or off using a Boolean slider, and outputs are given a simple text label.

When an input or output has its state determined as “ON” or true, the node has a distinct colour change from its “OFF” state. However, this does not apply to the gate nodes. These nodes as well as the wires and connecting circles exhibit no visual feedback when states change.

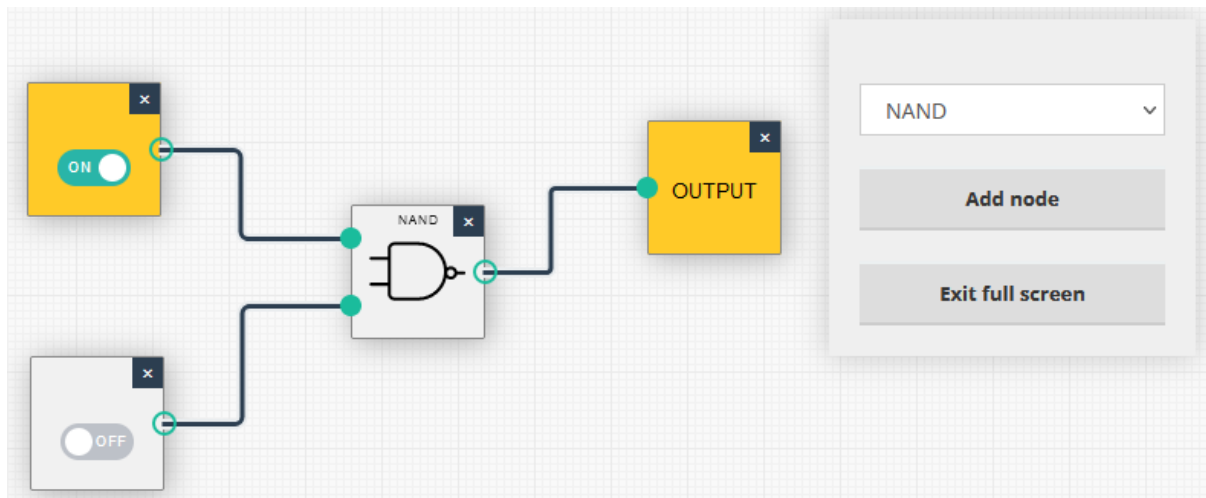


Figure 2 – Logic Gate Simulator: *Input / Output visualisation*

Overall, Logic Gate Simulator has good basic functionality but lacks depth for more uses than simple visualisation. The program lacks features such as truth table representation of logic circuits, visualisation of feedback for wires and gates, and problems to engage the user and facilitate task-based learning. All of these would aid the program more as a learning tool however this program was designed as a visualisation tool rather than a teaching application.

Logic.ly [2]:

Logic.ly is a more fleshed out simulation of logic circuits that include multiple extra features compared to Logic Gate Simulator. Logic.ly has an expanded set of available components including flip-flops, time-sensitive components, and signal-strength components. From inspection, Logic.ly is designed with electrical engineering as a consideration.

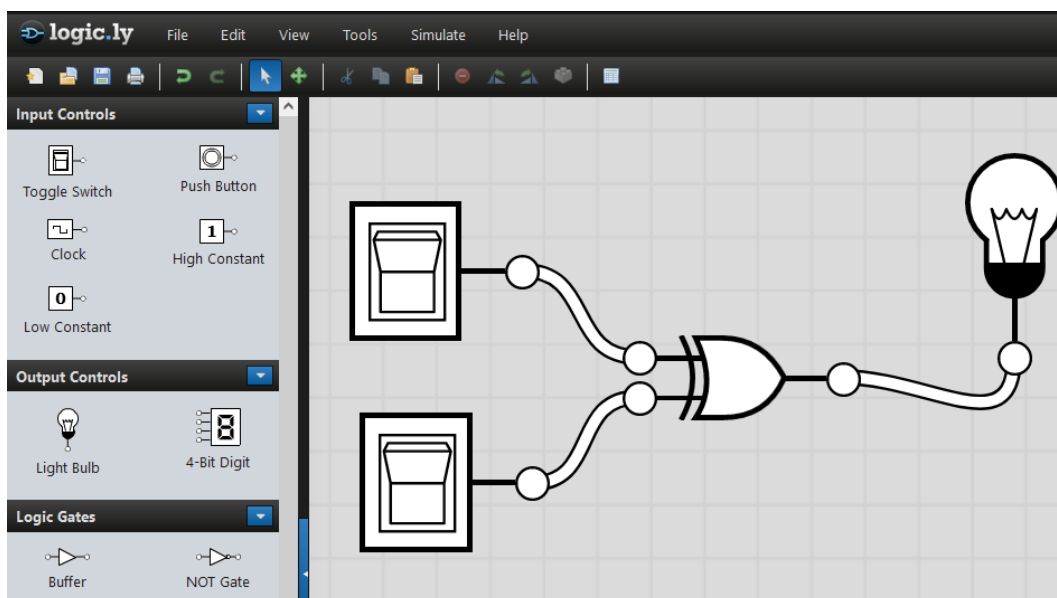


Figure 3 – Logic.ly: *Inserting components*

The program provides a drag-drop system for inserting components into the simulation space and connects components through a dragging wire system. Similar to Logic Gate Simulator, Logic.ly uses an input and output system for each component of the logic circuit. Outputs can only be fed from inputs and follows the same left to right pattern as seen before. The gates follow the same academically assigned symbols however the inputs and outputs are presented with more physical component-based representations. Inputs are provided using switches or buttons and outputs are shown using a light bulb or a 4 bit to 10-digit segmented display.

Logic.ly uses a highlighting system comparable to Logic Gate Simulator however also includes wires within the visual feedback. Every component that provides a value of TRUE is highlighted with a blue colour except for the gate components.

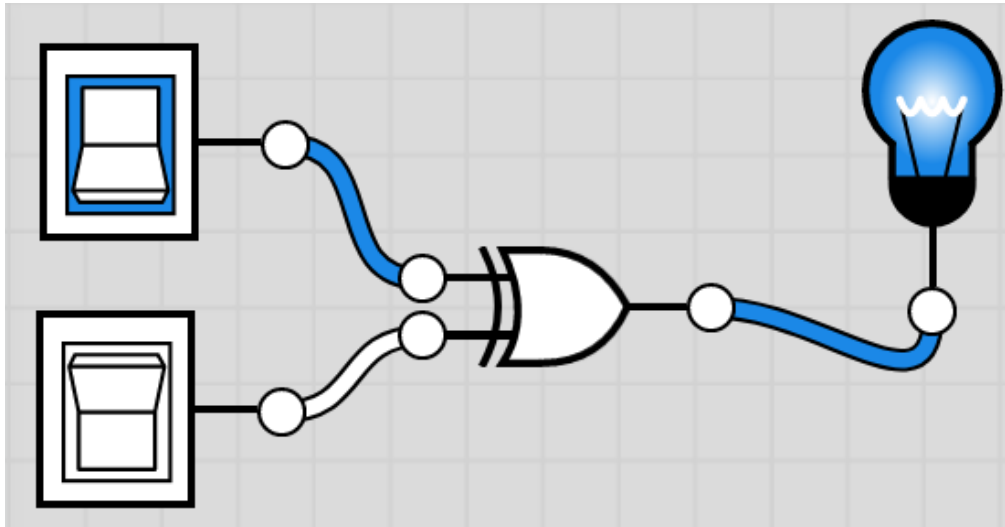


Figure 4 – Logic.ly: **TRUE value visualisation**

In addition to this, Logic.ly also checks for components that do not have all their inputs satisfied. Any gate that has incomplete dependency connections will have its output connections highlighted red. It also highlights any other components further from the gates connection with the red highlight. In this figure the light bulb filament is shown as red compared to the earlier figures where the filament was white or black.

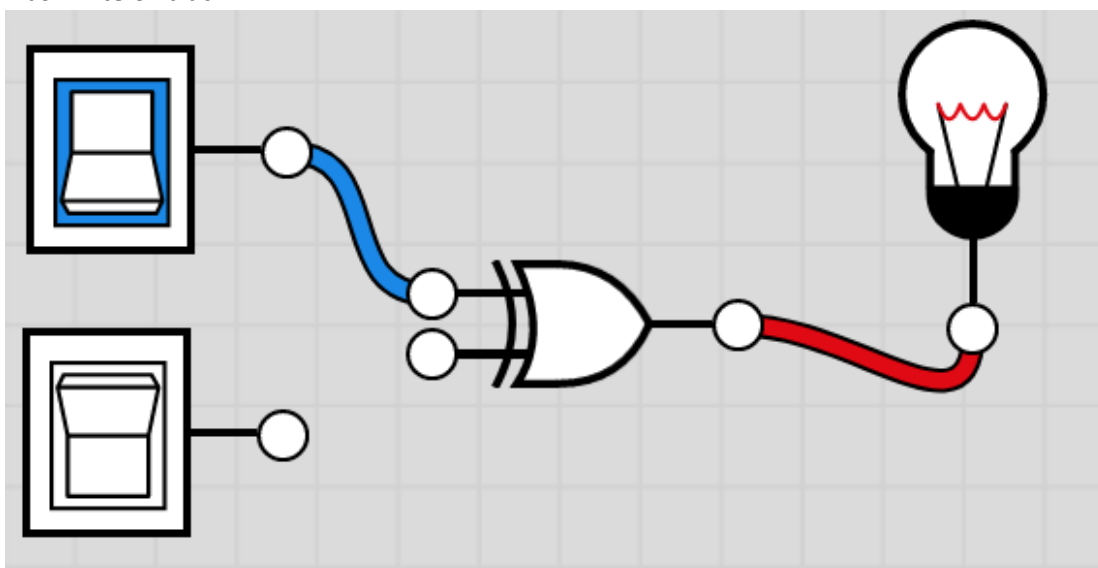


Figure 5 – Logic.ly: **Dependency error checking**

Logic.ly also provides a truth table generation tool that evaluates the current logic circuit and provides a table that encompasses all inputs, outputs and states of the circuit. It evaluates based upon the components present even if they are not connected up to each other.

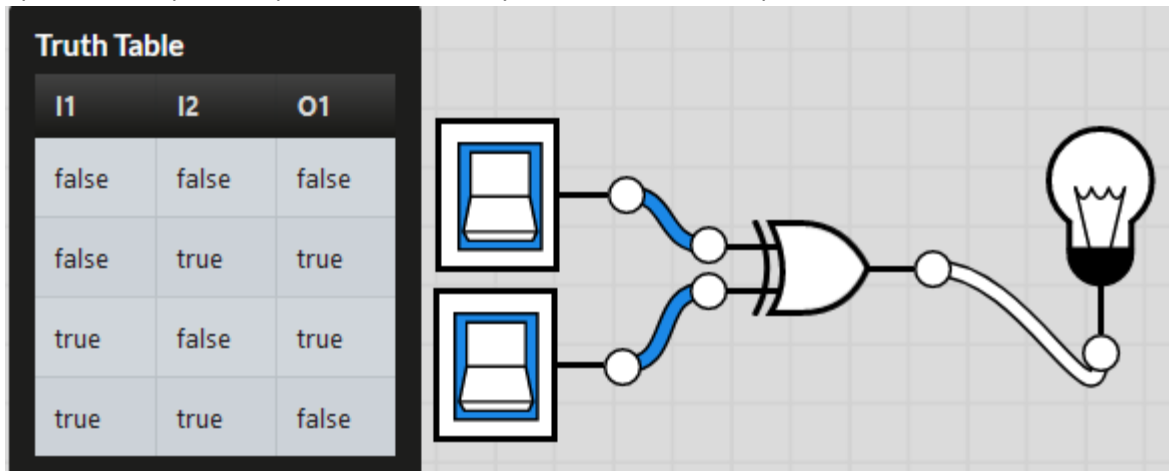


Figure 6 – Logic.ly: **Truth table generation**

These features make up the core of Logic.ly and provide more informed displays of logic circuits compared to Logic Gate Simulator. The program is similarly based as a visualisation tool and provides no problems or learning tasks but provides a wider range of components and clearer visualisation.

Core design merits

Both Logic Gate Simulator and Logic.ly provide the tools necessary to represent and create logic circuits from modular components. Their use is based upon the necessity to visualise a circuit. This could include visualisation of factors such as the states of the circuit, the configuration of gates, and the travelling of signals. In all cases, the visual component of both programs is specifically targeted to aid through visual learning. With the visual and other factors considered, the following design principles were taken and applied throughout the development of the project:

- Visual feedback for signal in components and wires
- Visual feedback for the validity of a component's connections
- Creation and deletion of selected components
- Ability to specify Boolean value of an input
- Ability to generate a truth table for a given configuration

In addition to these points, certain features were not included in either program. These design principles were created and applied to enrich the experience and align the project closer to its goals:

- Puzzles that engage problem solving with logic gates
- Specified area constraint for spatial awareness thinking
- Ability to orient components

Examples of existing VR logic circuit simulation

Existing simulations for VR logic gates have been developed in the past. This paper has a brief overview of two different simulations developed within the Unity game-engine. Both simulations were created for different goals but use an underlying Logic simulation for their respective applications.

Philippine VR university laboratory [3]:

The first simulation was used to recreate lab experiments for virtual university laboratory education sessions. A supporting paper [3] was published to evaluate whether the virtual lab simulation could be an alternative for students when attendance is unavailable. Unfortunately, due to the nature of documentation, a copy of the simulation was unobtainable. However, small screenshots of it are present within the paper. Here is one of the screenshots demonstrating a student using controls to manipulate the virtual components.

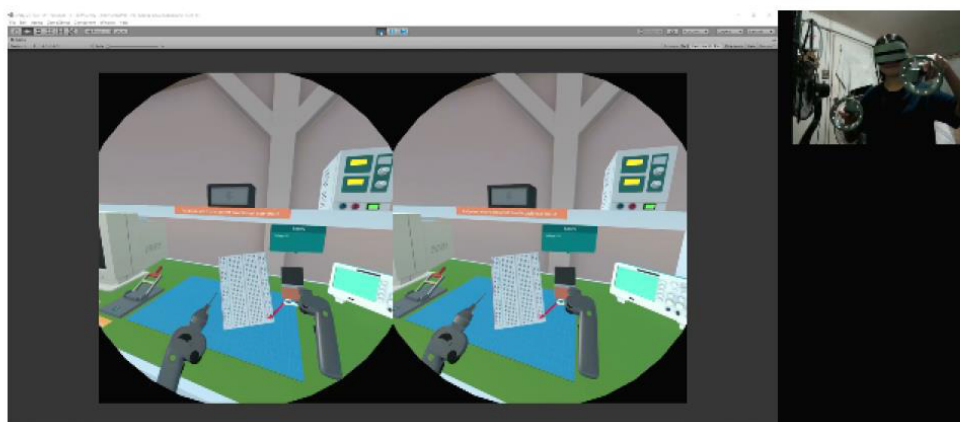


Figure 4: A participant performing VR Logic Circuits Laboratory 1

Figure 7 – VR Logic Circuits Laboratory: Screenshot of simulation from paper

A sample group of students completed laboratory exercises for one module within the simulation. It provided the necessary tools and components for students to complete the learning tasks. After the module of learning exercises, the students were then tested using the same evaluation assessments that the standard module uses. The following conclusion was that students were able to learn from the tasks provided within the simulation and pass the test with the lowest score being 70%.

means that the characteristics of the IC in the VR Laboratory simulate the actual IC. Base on the participants' score, the lowest score is 70% which proves that the participants have learned from the VR Laboratory. The use of Virtual Reality (VR) to simulate the Logic Circuits Laboratory 1 course from Mapúa University has proven to be successful.

Figure 8 – VR Logic Circuits Laboratory: Extract of conclusion from paper

This paper demonstrated the virtual environment can be utilised for learning and does not impact the ability to apply skills learnt.

Zen Marmot Digital Logic Simulation [4]:

The second simulation was used as an abstract view of logic gates within VR. Similar to Logic Gate Simulator and Logic.ly, the simulation used moveable components and expanding dragging wires to connect gates freely. Video demonstrations of the simulation were documented on Youtube [5] however no source code or version-controlled copy have been provided.

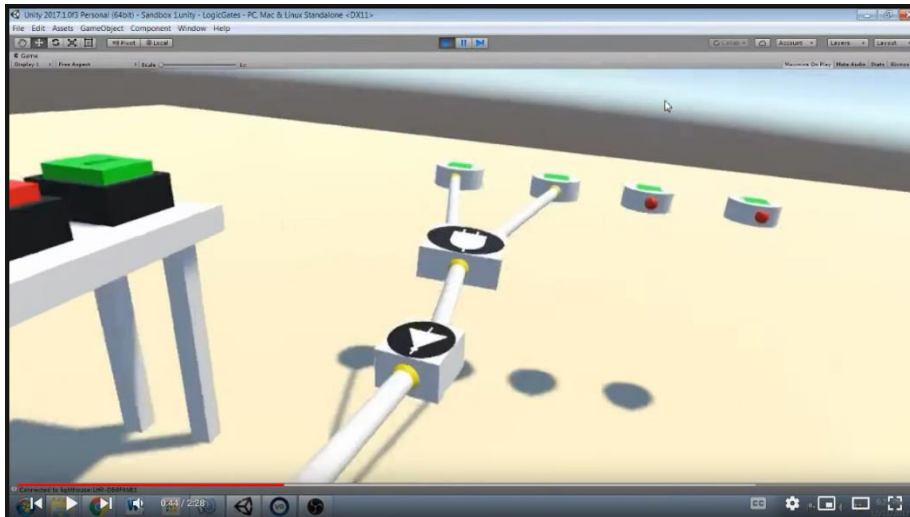


Figure 9 – Zen Marmot Digital Logic Simulation: **Video demonstration 2 NAND Gate**

The video demonstrations of the simulation [4] show that the logic system was integrated as part of a larger application based around sorting and feeding of datasets. The logic gates and wires produced outputs and were tested based upon the values in the queue of scenarios provided. Each set of inputs would correspond to a set of capsules that highlighted blue when the output should be ON and white when the output should be OFF.

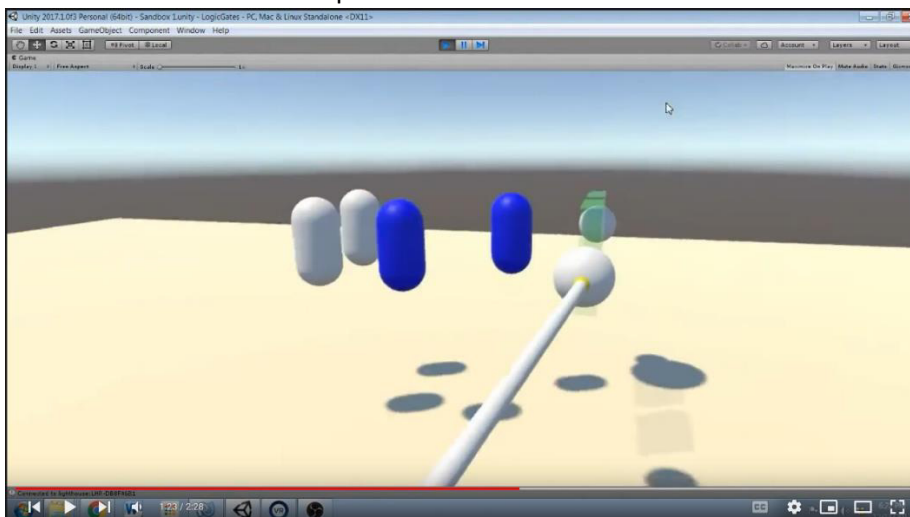


Figure 10 – Zen Marmot Digital Logic Simulation: **Video demonstration 2 Scenario Queue**



Zen Marmot Digital

154 subscribers

Logic Gates is a VR puzzle game in the early stages of development. In this video I test wire placement, connecting wires to gates, saving/loading data, and playing through an early level.

The white capsules represent conditions the player wants the output to be ON, the blue capsules represent OFF conditions. When the capsules hit the transparent boxes, they turn the inputs on and off. The player's goal is to connect the inputs to the output with the right digital logic in between to create the desired outcomes.

Figure 11 – Zen Marmot Digital Logic Simulation: **Video demonstration 2 Project description**

The project shows a freeform circuit builder which utilises circuit simulation and incorporates it into a learning puzzle. The simulation offers the same core functionalities as before but excludes features such as spatial constraints and truth table generation. Its main function is a puzzle game and therefore assumes that the player understands the functions of the gates before playing. Without a tutorial or explanation of the gates, it cannot be used to teach logic gates, only demonstrate them, or provide consolidating learning exercises for students who already understand logic gates.

The Unity Engine [6]

Engine Overview:

Unity is a virtual engine capable of creating 2D and 3D simulations and is developed by Unity Technologies. The engine is designed for editor side construction and alterations of projects, with an API that allows programmers to create scripts that execute during runtime. The engine is primarily used for construction of games due to its large variety of tools that streamline the game-making process. As such, Unity is a highly regarded engine with many casual and professional developers and even won “Best Engine” at the Develop Awards in 2012 [7].

Unity uses an object-oriented method that encapsulates game-objects within the simulation. Similar to how objects in object-oriented programming hold functions and variables, game-objects within the Unity engine hold engine specific components and attributes. These can include components such as colliders that use polygon meshes, and renderers that use textures. All components of a game-object can be viewed within the editor and enable developers to alter values and components without having to alter code. This allows developers to view the options for changes they make as well as making it very difficult to force the engine to do something it is incapable of. This limitation is also bypassed when using scripts which allows for more expansive alterations and runtime execution.

With Unity being a highly maintained engine and the project being similar to puzzle-based games, Unity was the best choice of engine available. Whilst the program is primarily a visualisation tool, the engine provided built-in support for 3D models, lighting, and input control that the project scope could not have accounted for. With these features already implemented, the focus of development would be on the functionality of the gates and wires rather than the implementation of user input and other engine-related features. The version of Unity which the project is based in is 2020.3.23f1.

More details about the Unity Engine can be found here [8]

<https://unity.com/products/unity-platform>

C# Scripting [9]

Despite the emphasis on in-editor development, certain aspects of a Unity application still have to be developed using code. C# is the language used to create scripts that interact with the engine's API. The API allows access to all the components and public variables set within components and scripts. A C# script attached to an object will use derive the base Unity functions through MonoBehaviour inheritance. This allows the script to communicate to the engine which gives it the capability of running functions on events that the game engine generates.

Scripts allow for sections of the project to be programmed individually within their respective classes. When assigning a script to an object, the script becomes part of the collection of components within the game object. Similar to the components such as the renderer, a script can have interactions with multiple other components or objects based on what the script's function is. By attaching scripts to objects, prefabricated game objects (or prefabs for short) can be defined and then instantiated.

VR with Unity [10]

Unity has had support for VR for many years in the form of the Unity XR packages. Unity's Extended Reality (or XR) packages hold core implementations for Augmented Reality and Virtual Reality. The XR Plug-in framework allows locational tracking data from controller and headset inputs to be mapped onto Unity game-objects. This leads to the ability to control in-game objects with inputs from controllers as well as reading of inputs from joysticks and buttons.

VR "hands" within Unity are just game objects with the necessary XR controller scripts that control their location and rotation. With the addition of meshes and colliders, the controllers can be turned into "physical" objects within the simulation and are able to collide with and interact with other objects.

Unity also has packages centred around VR applications such as the XR Interaction Toolkit [11]. This specific toolkit allows for most basic VR inputs including grabbing and moving objects as well as extra tools such as visuals and UI interaction. XR Grab Interactable [12] is the specific type of component that gives game objects the ability to be held by an XR controller. This allows any XR setup project to have access to "physical" objects that can be manipulated in 3D space using the controllers as hands. The XR Grab Interactable is the primary component that allows the project to have complete modular construction for combining gates and wires.

Unity includes Android build support as an optional package that allows for the building of .APK files. These files are used to install applications onto Android devices. A common headset used for VR development is the Oculus Quest 2 [13] due to its ability to run applications installed through .APK files as well as its independence from external processors. Development for the project initially took place on a Valve Index [14] which is a wired PC VR headset that used a direct connection between Unity and the headset. However, later development builds were created for the Oculus Quest to allow for remote deployment.

Approach

This section explores the following parts of the project:

- Functional and non-functional requirements
- Design aspects
- System requirements
- Project architecture

Requirements:

This section focuses on the required goals that the project set out to achieve. For clarity of which section these requirements focus on, this paper has split the requirements into two distinct sections.

1. Simulation
2. User interaction

The simulation section focuses on requirements specific to the functionality of components within the simulation which includes parts such as gates and wires. The user interaction section focuses on requirements specific to how the user interacts with the application as well as what feedback the application gives the user. Some of the requirements given were derived from design merits found within the background and research section explored earlier.

Simulation:

Functional Requirements:

1. *Four basic gates (AND, OR, XOR, NOT) must be implemented and provide the correct output for their assigned type given a set of inputs*
2. *Three wires (straight, 90-degree, splitter) must be implemented and carry signal to its connected components when given signal*
3. *All gates and wires must have 3D meshes representative of their 2D counterparts*
4. *All gates and wires must have the same centre of origin and have the same diameter connections when lined up*
5. *All gates and wires must only be active when inside a boundary cube*
6. *All gates and wires must snap to the closest 90-degree rotation and align to the Unity world space grid when placed*
7. *All gates and wires must misalign themselves when a user attempts to make them occupy a space that is already taken to avoid overlapping components*
8. *Input signal boxes must be able to be toggled ON and OFF*
9. *Input signal boxes must carry signal to its connected components when turned ON*
10. *Signal travelling through components must update every time there is a change in connection or change in signal*
11. *The sandbox section must include a truth table that appends output values to itself based upon the circuit constructed in the sandbox section*
12. *All puzzle evaluations must test a provided circuit using all state configurations and verify based on predicted output values*

Non-functional Requirements:

1. *All gates and wires must be visually distinct when not inside a boundary cube*
2. *All gates, wires, inputs, and outputs must visually show the direction they carry signal*

3. *All gates, wires, inputs, and outputs must visually show when they are carrying signal*
4. *All puzzle evaluations must show each individual state that is being tested in the truth table*
5. *All puzzle rooms must visually indicate whether an evaluation has passed or failed*
6. *All boundary cubes must be visually distinct from the gates, wires, inputs, and outputs*

User Interaction:

Functional Requirements:

1. *The user must be able to move within the 3D environment using joystick input*
2. *The user must be able to control the position of virtual hands using controllers*
3. *The user must be able to grab and pick up gate and wire components using grip input*
4. *The user must be able to press buttons to interact with the environment*
5. *The user must be able to spawn and destroy components*
6. *The user must be able to create notes within the environment*

Non-functional Requirements:

1. *All buttons must provide audio feedback for when they are pressed*
2. *All buttons must provide visual feedback for when they are pressed*
3. *All labels for buttons must be distinct from the background and simulation*
4. *All environments must be mute and simple colours and distinct from the simulation*
5. *All text information must be large and in a clear font*

Design aspects:

Whilst the requirements of the system provided a strong foundation and helped set goals in development, design aspects provided a set of soft targets that would aid the project. These design aspects encompass the whole project rather than tackling a specific section of functionality. They served as reminders of good design philosophy and helped streamline the development process in some areas. These were the following aspects and their respective justifications.

1. **Minimise the number of functions running within the Update() method**
The Unity engine runs functions during Unity events. The most common event is Update [15] which runs once every time the engine displays a new frame. Using this event frequently can increase the time between frames which decreases performance.
2. **Minimise the strain for processors**
Virtual Reality headsets come in two forms, tethered and standalone. Tethered headsets require an external graphical processor to render the frames and then the headset provides the display. Standalone headsets are responsible for both the rendering and displaying of the frames. By simplifying parts of the project such as textures, mesh polygons, and lighting, the project should perform well on both tethered and untethered headsets.
3. **Minimise eye strain**
Virtual Reality headsets are digital screens that sit close to the users' eyes. When a user puts on a headset, their eyes focus on the screen display and are not able to see the environment around them. To reduce eye strain and fatigue, the project minimised complex shapes and colours which leads to a more enjoyable experience.
4. **Simplify control inputs**
Controls are how users interact with a given system. With Virtual Reality, some users may not be as familiar with the platform and may not be comfortable using controllers and a

headset. By simplifying controls wherever possible, users can focus more on the logic simulations than trying to get acclimatised to the control scheme.

5. Provide freedom wherever possible

By creating limitations such as locked rooms or puzzles, exploring the project becomes difficult. Giving users the freedom to look at the tutorials and experiment with the sandbox aligns more with the project brief rather than firmly guiding users through an experience. Different users learn at their own pace and whilst this project is more of a learning reinforcement than a learning tutorial, it allows users to engage at their control.

System requirements:

Virtual Reality applications are more graphically intensive than their non-VR counterparts. This is due to VR applications being rendered twice, one for each eye. In order to create a cohesive 3D effect from flat images, each eye is presented their display in relation to where the other is displayed. This mimics the difference between vision in both eyes and allows the brain to perceive the experience as a 3D environment rather than two different images. Due to this increase in rendering requirement, the graphical and central processors running the application must have suitable speed to ensure a high-quality experience.

With standalone headsets, the Oculus Quest 2 is a commonly used headset and was used to test builds for the project. The Quest 2 is the unwritten community standard due to it being the lead selling headset in 2021 making up 78% of all headsets sold that year [16]. It has sufficient power to render VR games and applications and any standalone headsets with similar performance should be able to run this project's builds.

With tethered headsets, the processing power must come from a PC's Graphical Processing Unit (GPU) and Central Processing Unit (CPU). PCs have much more customisability with components and a wide variety of suppliers. The development machine used to create the project had an Nvidia RTX 2070 GPU and an AMD Ryzen 7 2700X CPU and had good performance when building and running the application. PCs of similar power can run the application easily however it would be recommended to run it on the most powerful machine one has on hand to ensure consistently strong performance.

Project architecture:

To understand how the architecture of the project was conceived, a basic understanding of how the Unity engine represents its environment. Much like the object-oriented organisation of game-objects within the engine, scenes are Unity's packaged environments that can be developed and used. Each scene contains a list of all the game-objects which in turn have all their components and scripts. As such, a project in Unity can be represented through a scene diagram with an entry point and branches representing different scenes and their dependencies.

In addition to this, scenes can also be built from prefabricated lists of game-objects. Scene templates encompass the necessary components required for every scene and allow for consistency when developing. An example of this would be the lighting model. A scene template specific to a project will often include lighting information to allow the skybox and general lighting of each scene to match one another. This reduces the time taken to set up a scene and ensures that values are exactly as the project requires.

The following is the project diagram with the layout of scenes:

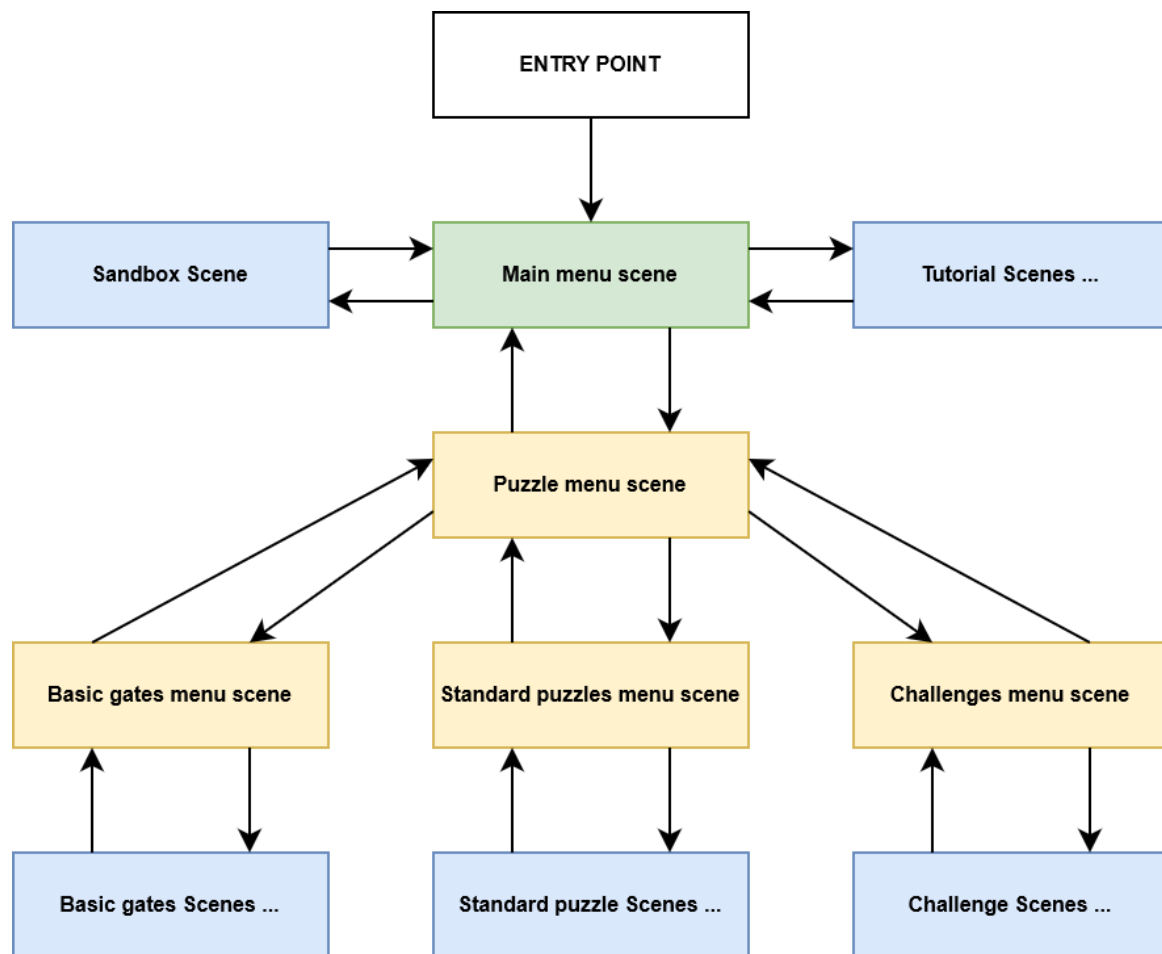


Figure 12 –Project Diagram: Scene diagram

Puzzles are split into three categories, basic, standard and challenge. Each set of puzzles is selected through its category's menu. The Sandbox and Tutorial sections can be accessed through the main menu. Any scene in blue has connections back to its respective menu that can be activated at any time.

The following is the project scene template components with detail about their functions:

- **XR Interaction Manager**
Manages input commands from the Virtual Reality controllers
- **Locomotion System**
Turns input commands into movement of the extended reality rig (XR Origin)
- **XR Origin**
Holds game-objects tied to the player including hands and camera for the head position
- **Room Parts**
Holds all walls, lighting information and scripts dealing with walls and the room
- **Text**
Holds all game-objects that have text on them such as menu labels and button labels
- **Buttons**
Contains all buttons in the room including buttons required to move between rooms

Implementation

This section explores the implementation of the following parts for the project:

- User input and XR rig
- Button functionality
- Component power controllers
- Component orientation
- Component visual changes
- Puzzle evaluation
- Truth table writing
- Marker and eraser

Note: during development a lot was learnt about the Unity engine. Many of the solutions created for problems are now known to be worse (inefficient, unnecessary, bad practice) than they could have been. As such, many notes will be scattered around this section highlighting how a better approach would have improved the program.

Notes will also be made to highlight certain sections of code that are reused in other sections of the project to cut down on the number of sections within the implementation. For example, code used for detecting the tag of a game-object was used in both Input/Output detection and deletion of objects using the bin. The bin was set to only destroy wires and gates and could not be used to destroy the marker or any part of the XR rig (player controllers and head)

User input and XR rig

The XR rig (extended reality rig) is a set of game-objects used to enable virtual and augmented reality input and display for Unity projects. The project uses an XR rig with one head camera and two hand controllers in each scene. As a scene changes, the XR rig in the new scene assumes control from the inputs of the user as the old scene is unloaded.

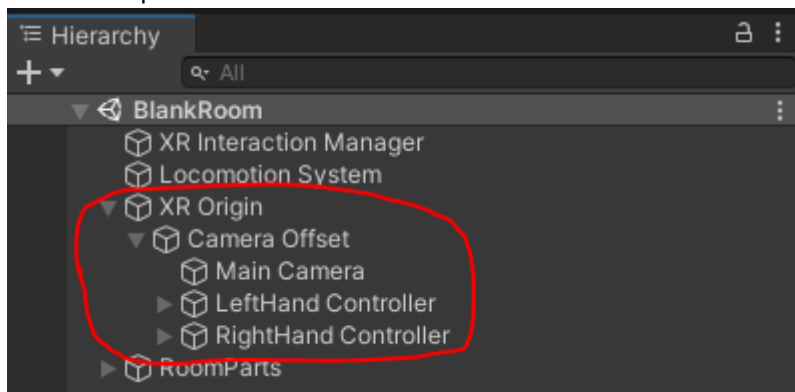


Figure 13 –Project hierarchy scene template: **XR Origin**

To enable the user to move around the project, the locomotion system [17] was used to turn joystick input from controllers into translation of the XR rig. This produced a gliding movement effect from inside the headset as the rig moved smoothly and continuously in the direction of the joystick.

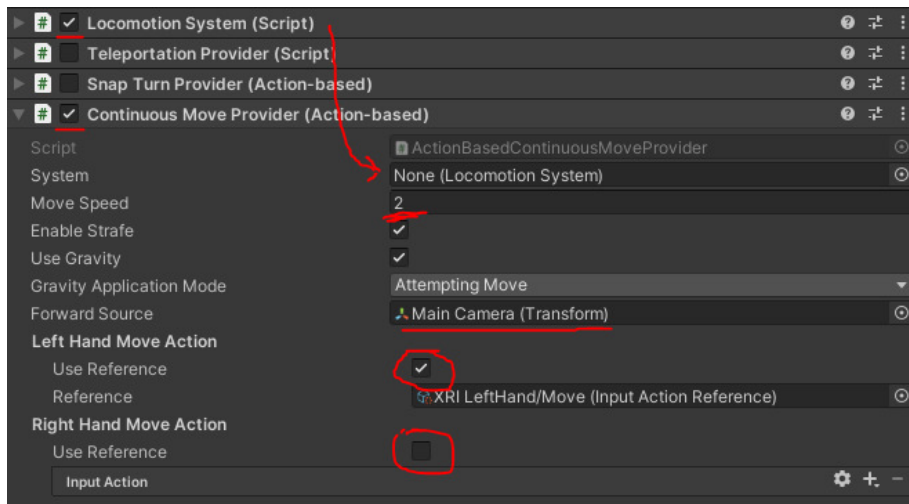


Figure 14 –Locomotion system: **Continuous movement**

Note: to simplify user input as much as possible, only the left-hand joystick was used for movement of the XR rig. This was so that movement would be strictly on the left hand and grabbing/orienting of components would be on the right hand. This would separate the movement of the rig and the rotation of grabbed components which reduced the complexity of the control scheme.

To enable the user to interact with buttons, each controller was assigned a child game-object consisting of a mesh sphere that could collide with buttons to simulate a button press. The spheres were created using Unity’s default sphere mesh and were coloured purple/pink to stand out against the background. They were also tagged with “Player” so that collisions with the button could be identified.

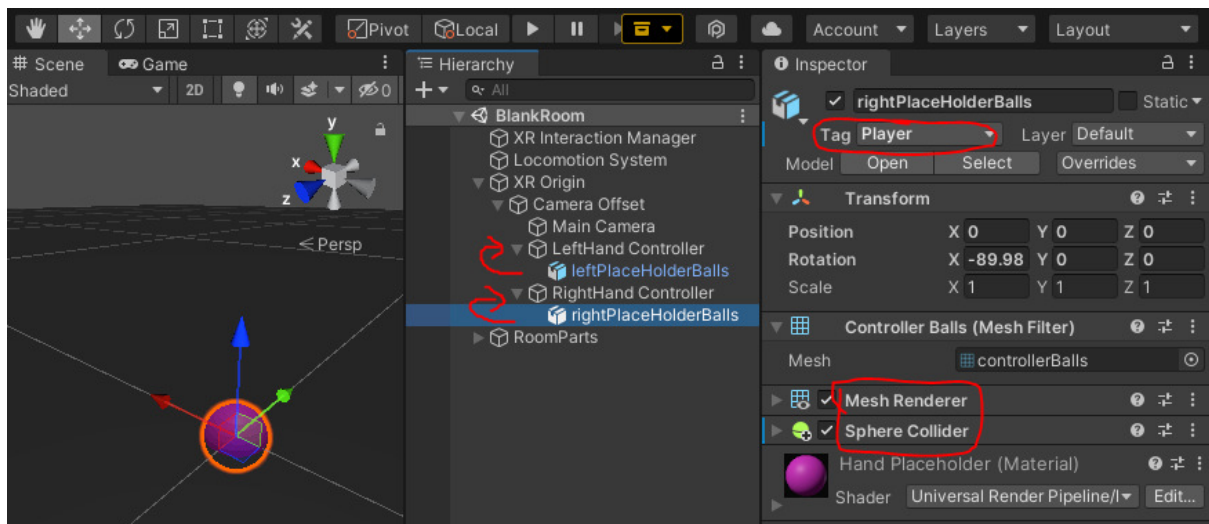


Figure 15 –XR Origin: **Virtual hand placeholders with colliders**

Note: the option to have a fully scaled and grip-controlled hand model was an option and would improve the visuals of using a finger to press a button. This was opted against however due to its complexity in animating and controlling as well as the larger collider size which could cause unwanted collisions. The decision to use a placeholder sphere for a hand was for simplicity in the collisions and visual aesthetics.

To enable the user to grab game-objects, the XR Ray Interactor [18] was added to the right-hand controller. The XR Ray Interactor uses a ray to identify game-objects it intersects. If the intersected game-object has an XR Grab Interactable [19] component on it and the user is pressing the grip button on the controller, the game-object will follow the user's hand as if they had picked up the object.

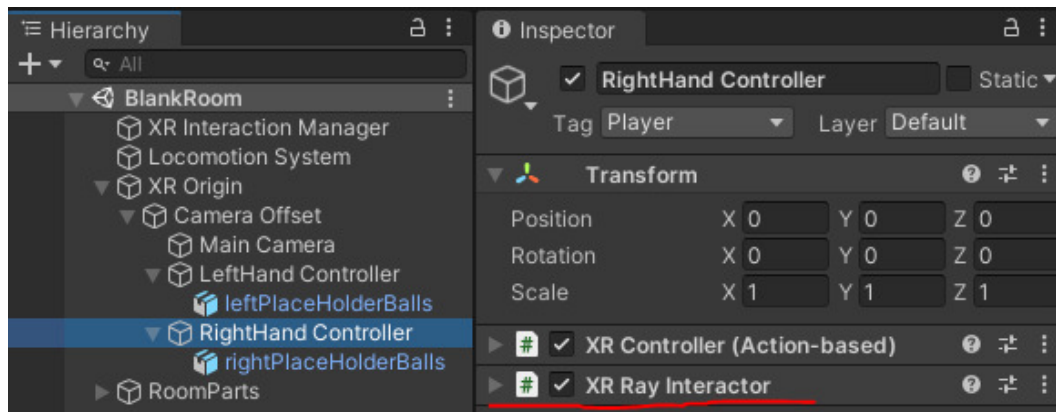


Figure 16 –XR Origin: **XR Ray Interactor**

Note: the XR Ray Interactor component has a lot of adjustable values including factors such as how far the ray is projected and activating events on hovering etc. None of these values were changed except the distance which was set to two meters.

Button functionality:

For the development of buttons, a prefabricated empty button was created. The button prefab consists of three child game-objects:

- Base
- Press
- Collider

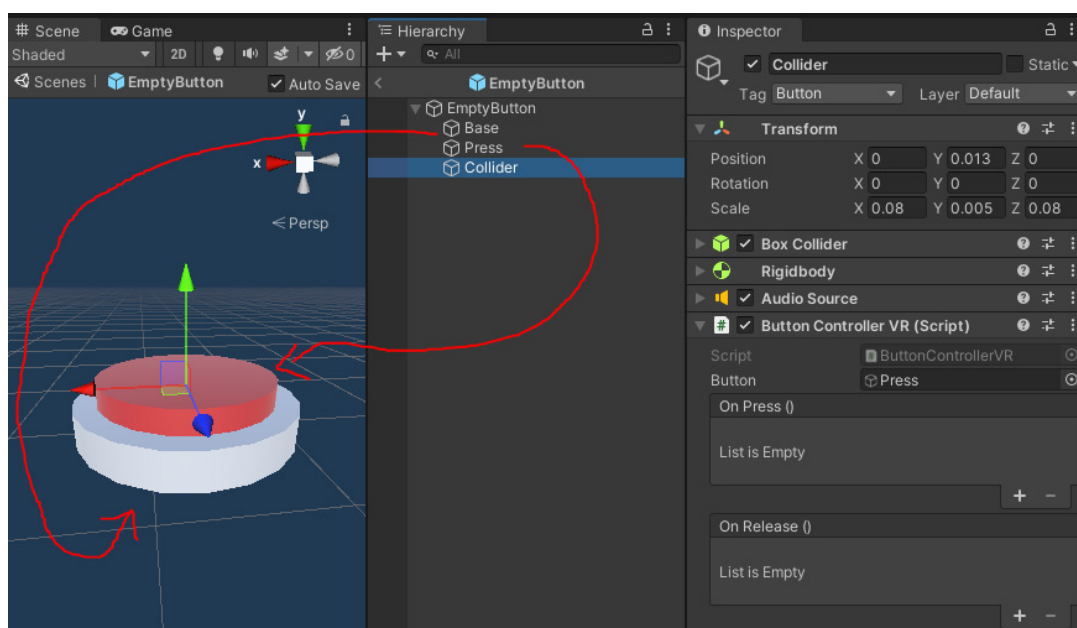
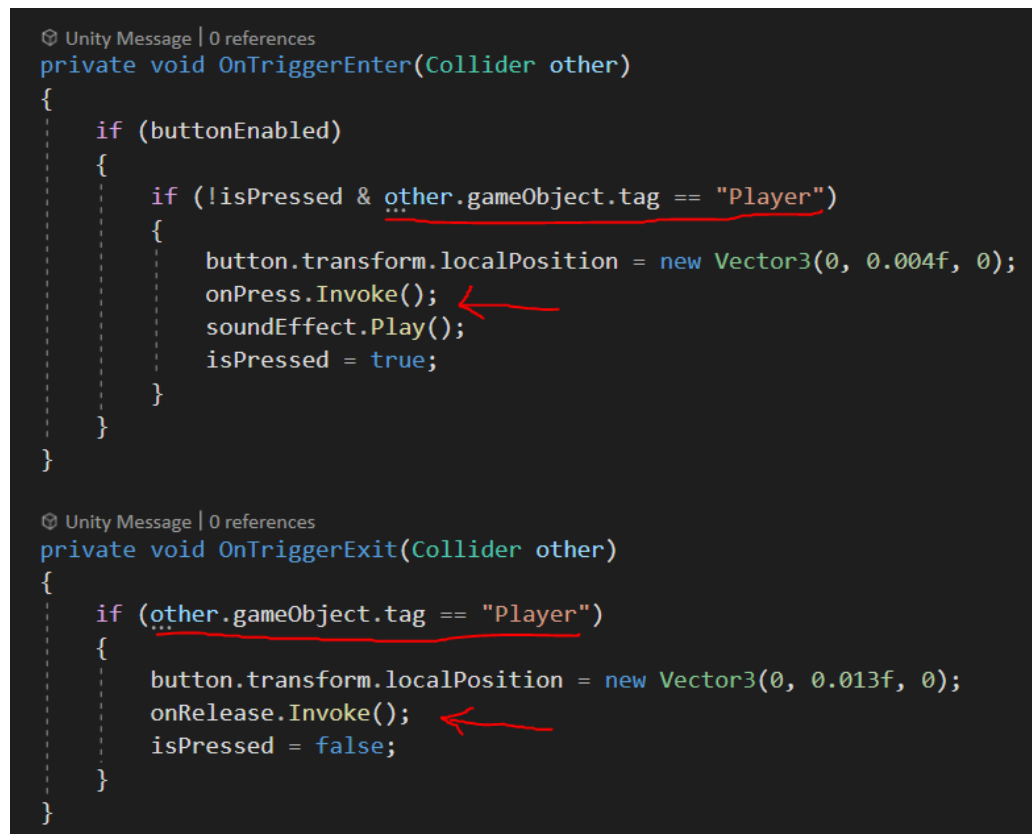


Figure 17 –Empty button prefab: **Button game-objects**

The base and press parts of the button are two separate polygon meshes that make up the 3D shape of the button. They do not have colliders attached to them but have renderers and materials assigned so that they are visible to the user. The collider houses the functionality for the base button and is invisible but triggers a collision event when pressed by the user. Every button in the project must do the following:

- Detect whether the player pressed the button
- Toggle between pressed and unpressed when the player interacts with it
- Create an event that other scripts can start execution from
- Move the Press game-object to visually indicate that the button is pressed
- Audibly indicate the button has been pressed

The following code shows two functions that trigger when collisions occur. The first is when another game-object intersects with the collider and the second is when another game-object leaves the collider having intersected with it. Both functions check whether the other game-object is the player and then invokes an event. In this script, there is the option to cause events to start execution when the button is pressed or when the button is unpressed.



```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (buttonEnabled)
    {
        if (!isPressed & other.gameObject.tag == "Player")
        {
            button.transform.localPosition = new Vector3(0, 0.004f, 0);
            onPress.Invoke();
            soundEffect.Play();
            isPressed = true;
        }
    }
}

Unity Message | 0 references
private void OnTriggerExit(Collider other)
{
    if (other.gameObject.tag == "Player")
    {
        button.transform.localPosition = new Vector3(0, 0.013f, 0);
        onRelease.Invoke();
        isPressed = false;
    }
}
```

Figure 18 –Button controller script: **Event invoking**

Inside both functions, the “button” also changes its local position which gives the visual effect that the button top has been pressed into the button base. OnTriggerEnter also plays the sound effect for the button when pressed.

Component power controllers:

All gates and wires required a power controller to store whether they were carrying signal. Every power controller had to communicate with a set of inputs and a singular output. Gates map all inputs to one output and wires map input to output. From this, the power controller was responsible for evaluating the inputs and determining the output in gates and was responsible for carrying signal in wires.

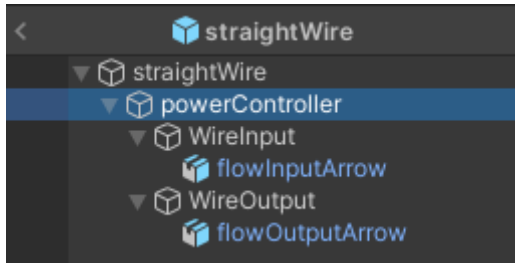


Figure 19 –Straight Wire Prefab: **Child game-object power components**

The power controller game-object had both input and outputs as child game-objects so that the parts could send messages upwards [19] or broadcast downwards [20] to allow parent-child objects to communicate.

Note: this is considered bad practice because the components have no guarantee of parent or child objects containing the required function specified. A much better implementation would be to assign references to the scripts manually and use the references from within the prefab. This was one of the first sections of the project to be developed and was created this way due to the desire to separate components and use parent-child relationships for game objects within the hierarchy. Whilst the parent-child relationship is extremely useful for functions such as group translation/rotation, the use of it in scripts for calling custom functions should be limited.

To allow wires and gates to understand when they are connected, two colliders were created for inputs and outputs. The script within the input has two public functions that receive and withdraw power. When a connection is made from an output to an input, the output references and calls the receive or withdraw power functions that tell the components power controller whether it has signal. When an input box is giving power to a circuit, its connection is considered an output as it is outputting signal into a wire. Therefore, the start of any signal starts with the input box which is toggled on by using its assigned button.

Note: not all code for inputs, outputs, wire power controllers, and gate power controllers will be shown due to the extensive length of code.

```
Unity Message | 0 references
private void OnTriggerEnter(Collider other)
{
    if (other.gameObject.tag == "Input")
    {
        connection = other.gameObject;
        UpdatePowerOutput();
    }
}
```

Figure 20 – Output script: **Refencing connection based on collision detection**

```

3 references
private void UpdatePowerOutput()
{
    if (connection != null)
    {
        if (powered)
        {
            if (connection.name == "WireInput")
            {
                connection.GetComponent<WireInput>().ReceivePower();
            }
            else if (connection.name == "GateInput A" | connection.name == "GateInput B")
            {
                connection.GetComponent<GateInput>().ReceivePower();
            }
        }
    }
}
else
{
    if (connection.name == "WireInput")
    {
        connection.GetComponent<WireInput>().PowerWithdrawn();
    }
    else if (connection.name == "GateInput A" | connection.name == "GateInput B")
    {
        connection.GetComponent<GateInput>().PowerWithdrawn();
    }
}
}

```

Figure 21 – Output script: **Updating power output to connected inputs**

Note: the connection names were checked when it would have been better to tag check instead. The distinction between GateInput and WireInput was due to the power controllers having to differentiate between inputs given. A mistake from the power controller design that moved forward into this code, had the gate power controllers taken reference the distinction would not have been needed because each input would map to its own reference.

The gate power controllers used short comparisons based upon their respective gate type to determine what output the two inputs provided. Whenever an input changes for any of the power controllers, it updates the signal state for the component and updates the output. This leads to a chain effect similar to how signal would be passed along a set of connections in the real world.

```

3 references
private void UpdateOutputBroadcast()
{
    if (active == true)
    {
        if (inputA & inputB)
        {
            //Debug.Log("Power controller gave power to output");
            BroadcastMessage("PowerOutput");
        }
        else
        {
            BroadcastMessage("StopPowerOutput");
        }
    }
}

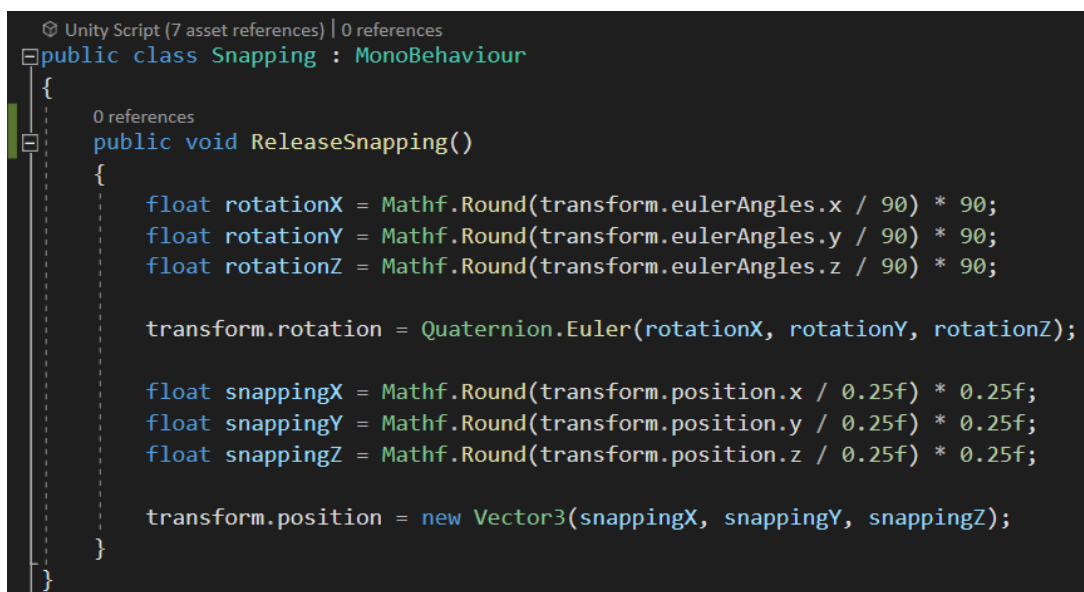
```

Figure 22 – AND gate power controller script: **Evaluating signal output based on inputs**

Component orientation:

The polygon meshes for gates and wires were designed to fit together so that when aligned the connections were flush. In order for them to connect together seamlessly within the project, they had to be oriented and translated when let go by the user to account for the minor inconsistency and inaccuracy in user input.

The snapping script takes the rotational x, y, and z value of the game-object it is attached to and rounds it to the closest 90 degrees. It then uses these rounded values and rotates the game-object with these values so that the game-object snaps to a perfect 90-degree average every time the function is called. The same process is done with the position of the game-object. Each boundary cube takes up 0.25m² of virtual environment space and the snapping script transforms the position so that it locks in positions with a factor of 0.25. This creates an invisible grid of snapping points spaced 0.25m apart from each other in the x, y, and z directions.



```
Unity Script (7 asset references) | 0 references
public class Snapping : MonoBehaviour
{
    0 references
    public void ReleaseSnapping()
    {
        float rotationX = Mathf.Round(transform.eulerAngles.x / 90) * 90;
        float rotationY = Mathf.Round(transform.eulerAngles.y / 90) * 90;
        float rotationZ = Mathf.Round(transform.eulerAngles.z / 90) * 90;

        transform.rotation = Quaternion.Euler(rotationX, rotationY, rotationZ);

        float snappingX = Mathf.Round(transform.position.x / 0.25f) * 0.25f;
        float snappingY = Mathf.Round(transform.position.y / 0.25f) * 0.25f;
        float snappingZ = Mathf.Round(transform.position.z / 0.25f) * 0.25f;

        transform.position = new Vector3(snappingX, snappingY, snappingZ);
    }
}
```

Figure 23 – Snapping script: **Rotation and position rounding**

All wires are scaled to fit perfectly within these 0.25m² boxes that the boundaries take up. Gates take up four boxes of space in a 2 x 2 configuration due to requiring two inputs for them to generate an output.

The snapping script is called when the component is let go by the user. XR Grab Interactable [12] can invoke Unity events when a game-object is interacted with. This can be grabbing, letting go, activating or hovering etc. “Select” is the keyword Unity uses to differentiate between picking up and activating a game-object with XR Grab Interactable. Activation is treated as a separate event and a common use for it is in the implementation of projectile spawning in games with weapons. Selecting the weapon allows the user to grab the weapon however activating the weapon fires the projectiles. In this case, select exited is the event which the user lets go of an object and this is when the rounding of rotation and position is needed.

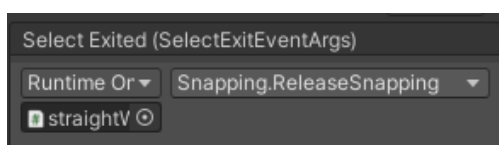


Figure 24 – Snapping script: **Executing on select exited for XR Grab Interactable**

Component visual changes:

Components that carry signal were required to provide visual feedback in the form of glowing from non-functional simulation requirement 3. To achieve the glowing effect, an emissive modifier [21] to the material was added that could be turned on and off using an activating script. The active emission script uses the keyword “_EMISSION” to activate the glowing affect when called by the power controller script of the game-object component.

```
0 references
public void ActiveGlowOn()
{
    emissiveMaterial.EnableKeyword("_EMISSION");
}

0 references
public void ActiveGlowOff()
{
    emissiveMaterial.DisableKeyword("_EMISSION");
}
```

Figure 25 – Active emission script: **Enabling keyword _EMISSION**

```
3 references
public void ReceivePower()
{
    powered = true;
    //Debug.Log("Input collider set power");
    BroadcastMessage("ActiveGlowOn");
    SendMessageUpwards("SetPower");
}

6 references
public void PowerWithdrawn()
{
    powered = false;
    //Debug.Log("Input collider stopped power");
    BroadcastMessage("ActiveGlowOff");
    SendMessageUpwards("StopPower");
}
```

Figure 26 – Wire input script: **Broadcasting glow activation function to child game-objects**

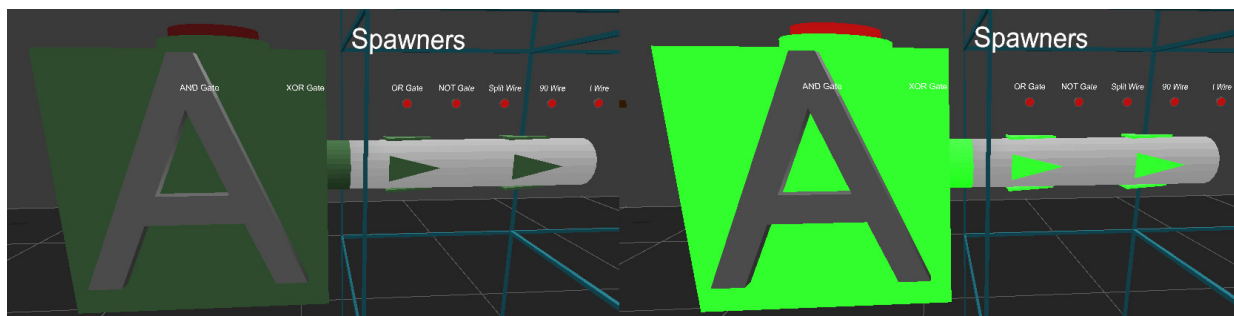


Figure 27 – Straight wire: **Glow effect comparison**

Note: the active emission script was reused multiple times for the inputs, outputs, gates, and wires with their respective emissive materials. The effect was also used for the buttons and truth table cells so that they would light up.

Puzzle evaluation:

With wires, gates, inputs, and outputs able to carry signal with visual feedback, puzzle evaluation was the last step in creating learning tasks for users. The puzzle evaluation scripts are all dependant on the puzzle that is being shown. Each puzzle has to:

- Recreate every state of the inputs from the provided truth table
- Check the output values in the simulation
- Check that the values match with expected values from the truth table

Puzzles included references to each input and output box because they were the only values that needed to be read from and modified. When going through each state in the truth table, the referenced input boxes would be powered on or off and then the output would be checked after a buffer. This buffer time was important because of the time delay as each component updated its signal carrying status one by one. Whilst this was very quick, the tiny delay in script execution would take the value of output without this chain completing without the buffer time. This buffer time was achieved using the invoke function [22] which takes two arguments: the function name and the time until invoking. This extract shows switching on input boxes and buffering before checking the output value and return a pass or fail state for state 1. (Both inputs are set to on, therefore the checkbool value should return true and allow the room controller to be notified that the state has been passed)

```
0 references
private void StateSet1()
{
    inputBoxA.GetComponent<InputPower>().PowerOn();
    inputBoxB.GetComponent<InputPower>().PowerOn();
    Invoke("CheckSet1", stateChangeTime);
}

0 references
private void CheckSet1()
{
    CheckOutputPower();
    if (checkBool)
    {
        stateCheck += 1;
        roomController.GetComponent<TableController>().PassState(1);
    }
    else
    {
        roomController.GetComponent<TableController>().FailState(1);
    }
    StateSet2();
}
```

Figure 28 – AND gate puzzle evaluation: **Checking state passing**

Note: invoking to cause a buffer is bad practice. Using a coroutine and yielding execution is a much more effective way of calling functions with delay. It ensures that the coroutine called exists and has correct parameters for execution whereas invoking assumes the function will be available to execute.

Truth table writing:

In the sandbox room, the truth table appends to itself based upon the outputs that have been provided within the sandbox space. The sandbox truth table consists of three inputs and three outputs with a total of eight combinations of inputs and outputs. The script used to append to the table follows the same logic as the puzzle evaluation script but takes the signal value of the output boxes and writes to the text labels in the table.

```
0 references
private void StateSet1()
{
    inputAPower.PowerOn();
    inputBPower.PowerOn();
    inputCPower.PowerOn();
    Invoke("StateSet2", stateChangeTime);
}

0 references
private void StateSet2()
{
    UpdateRow(1);
    inputAPower.PowerOn();
    inputBPower.PowerOn();
    inputCPower.PowerOff();
    Invoke("StateSet3", stateChangeTime);
}
```

Figure 29 – Truth table updater script: *Setting states then invoking for buffer*

To make this script more concise than the puzzle evaluation script, appending to the table happened in the invoked function before the next set of inputs is changed. This ensured that there was buffer time whilst cutting down on the amount of invoke calls in the script.

```
8 references
private void UpdateRow(int rowNumber)
{
    OutputCheck();
    if (rowNumber == 1)
    {
        if (tempA)
        {
            row1A.GetComponent<SandboxCellChange>().PositiveCell();
        }
        else
        {
            row1A.GetComponent<SandboxCellChange>().NegativeCell();
        }

        if (tempB)
        {
            row1B.GetComponent<SandboxCellChange>().PositiveCell();
        }
        else
        {
            row1B.GetComponent<SandboxCellChange>().NegativeCell();
        }

        if (tempC)
        {
            row1C.GetComponent<SandboxCellChange>().PositiveCell();
        }
        else
        {
            row1C.GetComponent<SandboxCellChange>().NegativeCell();
        }
    }
}
```

Note: a more efficient way of checking all the states and writing to the table would have been to construct each row of the table as a class. Then the whole table could have been represented as a list of row objects that could be iterated through with their input values read and set. This would then be appended to and would be a possible way for generating tables of scalable size depending on the constraints of the class.

The following extract was one a long logic sequence that manually assigned the references cells their signal value. Due to the emissive labels in the truth table being referenced directly, each comparison had to be made and then the corresponding cell updated.

Figure 30 – Truth table updater script: *Checking output states and changing cell values*

Marker and eraser:

The marker and eraser game-objects also had the XR Grab Interactable [12] component attached to them which allowed users to hold the marker similar to how they would write in real life. The marker and eraser functioned identically by using a raycast to detect a wall and then change the texture of the wall. To achieve this the following steps are performed:

1. Creating a coordinate position of the marker based on the x and y values of the texture
2. Changing pixels on the texture based on the coordinates of the last position of the marker and the new position and the marker size
3. Set the current position as the last position of the frame
4. Repeat until the pen no longer touches the wall

```
1 reference
private void Draw()
{
    if (Physics.Raycast(tipTransform.position, transform.forward, out touchedObject, tipHeight))
    {
        if (touchedObject.transform.CompareTag("Board"))
        {
            if (boardObject == null)
            {
                boardObject = touchedObject.transform.GetComponent<Board>();
            }

            touchPosition = new Vector2(touchedObject.textureCoord.x, touchedObject.textureCoord.y);

            var xPosition = (int)(touchPosition.x * boardObject.textureSize.x - (tipSize / 2));
            var yPosition = (int)(touchPosition.y * boardObject.textureSize.y - (tipSize / 2));

            if (yPosition < 0 || yPosition > boardObject.textureSize.y || xPosition < 0 || xPosition > boardObject.textureSize.x)
            {
                return;
            }

            if (touchedLastFrame)
            {
                boardObject.texture.SetPixels(xPosition, yPosition, tipSize, tipSize, colourArray);

                for (float percent = 0.01f; percent < 1.00f; percent += 0.02f)
                {
                    var lerpX = (int)Mathf.Lerp(lastTouchPosition.x, xPosition, percent);
                    var lerpY = (int)Mathf.Lerp(lastTouchPosition.y, yPosition, percent);

                    boardObject.texture.SetPixels(lerpX, lerpY, tipSize, tipSize, colourArray);
                }

                transform.rotation = lastTouchRotation;

                boardObject.texture.Apply();
            }

            lastTouchPosition = new Vector2(xPosition, yPosition);
            lastTouchRotation = transform.rotation;
            touchedLastFrame = true;
            return;
        }
    }

    boardObject = null;
    touchedLastFrame = false;
}
```

Figure 31 – Marker script: **Drawing to a wall with tag "Board"**

The eraser achieves the erasing effect by changing the affected pixels to the default colour of the wall boards. From the perspective of the user, the marker is erasing the drawn lines so it saved implementing a new script.

Note: a lot of scripts and details had to be skipped over due to the sheer number of repeating elements and small scripts for joining sections together. For example, the gate scripts follow the same format but because they explicitly provide different outputs and are attached to different components, they are required to be full scripts to allow the game-objects to work.

Note: these scripts presented are the most important and explain how the simulation derives signals and carries them between components. The full scripts can be found within the included in supporting files under “scripts” folder for extra inspection.

Video demonstration

Before the full evaluation it would be highly recommended to watch the following video demonstration of the project. Due to a big part of the simulation being able to update in real-time, the visual demonstration will aid in viewing how the project functions as a whole and what the project looks like. The main aim of this project was to be a visualisation simulation and screenshots really do not capture the essence of the project. The video demonstration also shows a that a lot of initial requirements were passed during development.

The video for the demonstration can be found in the supporting files.

Results and Evaluation

This section explores the following:

- General commentary of project outcome
- Description of test procedures
- Testing against criteria set out by requirements

Project outcome:

The project reached a satisfactory level in development. The core functions of the simulation were implemented, and the components were able to carry signal and update in real-time. Only three puzzle rooms were constructed and there were no challenge rooms constructed. Whilst it would have been ideal to have more, the base foundations for constructing challenges had been created. With the boundaries and evaluation code, it would be fairly simple to implement more puzzles.

The project also had very good visual and audible feedback for users as seen within the video demonstration. Each section that used distinct colours was easily distinguishable from within the headset and did not cause any eye strain.

The control scheme and grab mechanics integrated seamlessly into the project. Unity’s pre-built functionality in the XR Interaction Toolkit made the base VR implementation very easy to accomplish and allowed development of the simulation to be the main focus.

Overall, all parts of the project function exactly as required and provide a smooth and functioning experience.

Description of testing procedures:

In order to test the simulation, the project was built using Unity's application builder and then tested on a PC tethered Valve Index and an Oculus Quest standalone headset. Each requirement was tested on both headsets before evaluating as passed.

To test the functionality of components, the sandbox area was used to house components for testing. The components required for testing were placed into the boundary areas and the output was noted and compared to the expected outcome for the components. With truth table testing and puzzle evaluation testing, multiple sets of configurations were tested to check that the table updated correctly, and the evaluation checked the outputs correctly.

Testing of the user interactions was based mainly on the control scheme and expected outcomes of user input. Unity provided functionality such as the movement by joystick was tested as well despite it being unnecessary to ensure that the project functioned as expected.

All tests were run as individually as possible however certain aspects such as the moving of objects would be required for configurations of components to be altered. The outcome of the tests were highly positive.

Testing against requirements:

This section is split into the two sections that were highlighted earlier in the requirements section of the report. The results of the project are as follows:

Simulation:

Functional Requirements:

1. *Four basic gates (AND, OR, XOR, NOT) must be implemented and provide the correct output for their assigned type given a set of inputs*
PASS – All gates provided expected outputs
2. *Three wires (straight, 90-degree, splitter) must be implemented and carry signal to its connected components when given signal*
PASS – All wires carried signal to connected components
3. *All gates and wires must have 3D meshes representative of their 2D counterparts*
PASS – All gates and wires had suitable 3D mesh models
4. *All gates and wires must have the same centre of origin and have the same diameter connections when lined up*
PASS – All gates and wires lined up and connected seamlessly
5. *All gates and wires must only be active when inside a boundary cube*
PASS – All gates and wires were only active inside the boundaries
6. *All gates and wires must snap to the closest 90-degree rotation and align to the Unity world space grid when placed*
PASS – All gates and wires snapped to 90-degree rotation and to the closest 0.25m
7. *All gates and wires must misalign themselves when a user attempts to make them occupy a space that is already taken to avoid overlapping components*
PASS – All gates and wires pushed each other out of alignment when attempting to occupy the same space
8. *Input signal boxes must be able to be toggled ON and OFF*
PASS – Input signal box prefab had a button that toggled their signal

9. *Input signal boxes must carry signal to its connected components when turned ON*
PASS – *Input signal box prefab carried signal to connected components*
10. *Signal travelling through components must update every time there is a change in connection or change in signal*
PASS – *All components that carry signal updated signal status every time there was a change in connection or signal*
11. *The sandbox section must include a truth table that appends output values to itself based upon the circuit constructed in the sandbox section*
PASS – *The sandbox truth table appended the output values to the results when the “write table” button was pressed*
12. *All puzzle evaluations must test a provided circuit using all state configurations and verify based on predicted output values*
PASS – *All created puzzles evaluated the circuit by testing all input combinations and verifying the output values*

Non-functional Requirements:

1. *All gates and wires must be visually distinct when not inside a boundary cube*
PASS – *All gates and wires have a distinct glow when being placed in the boundary cube*
2. *All gates, wires, inputs, and outputs must visually show the direction they carry signal*
PASS – *All gates and wires have green arrows visually indicating the direction they carry signal*
3. *All gates, wires, inputs, and outputs must visually show when they are carrying signal*
PASS – *All green arrows that indicate direction of signal glow when signal is being carried*
4. *All puzzle evaluations must show each individual state that is being tested in the truth table*
PASS – *All input boxes in a puzzle room glow to indicate the states that the puzzle evaluation is setting them to*
5. *All puzzle rooms must visually indicate whether an evaluation has passed or failed*
PASS – *The room walls show green or red depending on passing or failing the puzzle evaluation*
6. *All boundary cubes must be visually distinct from the gates, wires, inputs, and outputs*
PASS – *Boundary cubes are blue and have thin models which are visually distinct from the components inside them*

User Interaction:

Functional Requirements:

1. *The user must be able to move within the 3D environment using joystick input*
PASS – *Joystick input on both controllers moved the XR rig in the environment*
2. *The user must be able to control the position of virtual hands using controllers*
PASS – *Movement of both headset controllers moved the pink sphere hand substitutes in the environment*
3. *The user must be able to grab and pick up gate and wire components using grip input*
PASS – *Grip input on both controllers grabbed all gates and wire prefabs in the environment*
4. *The user must be able to press buttons to interact with the environment*
PASS – *All buttons that were enabled functioned when pressed by the pink sphere hand substitutes*

5. *The user must be able to spawn and destroy components*
PASS – Pressing the spawn buttons spawned the assigned component and placing components in the bin deleted them from the environment
6. *The user must be able to create notes within the environment*
PASS – The marker and eraser created marks on the walls which can be used to create notes

Non-functional Requirements:

1. *All buttons must provide audio feedback for when they are pressed*
PASS – The button prefab provided audio sound effect when activated
2. *All buttons must provide visual feedback for when they are pressed*
PASS – The button prefab moved its visual button top to simulate pressing of the button
3. *All labels for buttons must be distinct from the background and simulation*
PASS – All labels were in white to be visually distinct from the walls and components
4. *All environments must be mute and simple colours and distinct from the simulation*
PASS – All walls were black or dark grey
5. *All text information must be large and in a clear font*
PASS – All text was larger than components and in Unity's default font for clarity

Future work

The project managed to complete all goals set out by the requirements but also has a lot of room for potential growth. Scope of the project was taken into consideration when planning the project and as such, the completion of the project was not a surprise. However, the scope was limited due to the time and complexity considerations and with more time, the project could have had more functionality. The areas of possible future development can be split into three categories: user centred, content centred, and simulation centred.

The following are the categories of possible future development and their justifications:

User centred:

- Included the ability to change the dominant hand
During development it was decided to split control of the hands so that one would control movement and the other would control components. For users who prefer a different dominant hand for moving, this expansion would lead to easier control.
- Include the ability to spawn components using the controllers
With spawning tied directly to the buttons, sometimes users must move away from the construction area to grab more components. Whilst it simplified the control scheme at the time, it meant that large configurations required multiple trips back and forth between the puzzle areas and the spawners. By allowing the user to spawn a component using the controller, it would reduce the interruption to thinking flow.
- Include the ability to carry a note tablet
Writing and noting on the walls was a good but suffered the same flaws as spawning. By constantly turning around to look at notes or having to move to edit them, there was a lot of time wasted through moving. Carrying a note tablet removes this movement however it would be difficult to get clear and accurate writing because the user has nothing to push against.

Content centred:

- Adding more puzzles to the project
The number of puzzles and lack of challenge puzzles hinders the projects' ability to be a learning tool at the moment. It serves as a visualisation tool but with more rooms could be a good learning reinforcement puzzle game.
- Include ability to save and load configurations
Saving and loading configurations of components can allow users to recreate and move components between rooms. This could be incredibly helpful if becoming stuck in a puzzle and finding a solution in the sandbox room when experimenting. It also allows duplication of configurations which cuts down on construction time for larger circuits. This could be included alongside challenges of recreating 8-bit adders or similar challenges.
- Adding different types of puzzles to the project
Currently the puzzles in the project focus around reverse engineering a logic configuration however, this could be expanded. An example of this could be to find an alternate solution to an existing circuit by removing components or distilling the circuit into its logic statement.

Simulation centred:

- Adding more components
The project chose to omit gates such as NAND due to their ability to be reconstructed using an AND and NOT gate together. Despite this, it would allow for more complex puzzles and more compact boundaries.
- Altering features to include electrical engineering applications
Allowing signals to decay and also limiting signal can lead to applications for electrical engineering to be represented within the simulation. It could also be modified to represent electrical circuitry for physics-based learning material.

With all these considerations, it is easy to see how the project could be expanded to become a versatile and informative learning tool. It could also be adapted to focus on puzzles and shift into a puzzle style game that can be played for fun. The project is still open ended and is a solid foundation for any of the explorations listed above.

Conclusions:

The main aim of the project was to provide a visualisation of logic gates within 3D space and allow users to configure circuits and analyse them in respects to spatial requirements. With the project passing all core requirements set out during planning, it could be said that the project was a success.

Unity's engine and provided components gave the project a strong virtual environment and the functionality required for user interaction. The simplification of aspects in the project such as the control scheme allowed for easy use as well as streamlined the development process.

All components within the simulation including the gates, wires, inputs, outputs, buttons, and spawners function exactly as planned and provide an accurate experience when constructing logic configurations. The simulations of circuits and subsequent evaluations of puzzles was everything the project specification set out to achieve.

The scope of the project was manageable and considered what could be achieved in the time frame given. Its simple aesthetic and designed look reflect how the focus was centred on the simulated

components rather than the spectacle of Virtual Reality. Given more time, the project scope would have included features mentioned in the Future Work section.

Overall, the project was a good success and able to satisfy all the requirements set out with all bugs ironed out during the development process.

Reflection on learning

Despite the strong outcomes and passing of all requirements, the project was extremely difficult to develop. The Unity engine was a game engine I had limited experience in the past with and I had never encountered the VR packages that I needed to complete it. Unity and game engine programming was not taught in the Computer Science degree program and as such, all the knowledge required was learnt during the development process.

It was due to these factors that I limited the project scope to a small set of requirements from the idea. It was a combination of streamlining and simplifying my work and overall let me take incremental steps at a time to achieve the goals set out.

The following is a list of the skills I picked up throughout the project:

- Using the XR rig in Unity
- Creating polygon meshes in Blender [23] by extruding shapes
- Referencing scripts within Unity
- Emissive materials in Unity
- XR rig input
- XR Interaction Toolkit features such as XR Grab Interactable and XR Ray Interactor
- Invoking and coroutines in Unity
- Collisions and event triggers in Unity
- Using C# to script and interact with the Unity API

At the start of development, I was constantly researching ways of implementing features into the project. The first week consisted of watching tutorials on Blender and Unity VR so that I could set up a basic XR rig with control scheme and set up a basic environment.

Every step in development involved learning something new about the Unity engine and I quickly learnt that the engine provided a lot more than I originally thought. There are whole sections of Unity including particle effects and static objects that I did not get to explore but were shown briefly in a lot of tutorials I watched.

The main lesson I learnt from this project was that there is always more to learn. Every aspect of a project can be improved upon and I found myself coming across better solutions and tools for problems I had already solved in inefficient ways. This is what led to the notes within the implementation section. Whilst there are quite a few areas in which I could have improved, I realised that time constraints wouldn't have let me fix them and continue development on time. Managing and planning ahead played a key role in ensuring that the project was making good progress.

Had I had more time I would have chosen to fix the implementations I already had before expanding the feature set. Whilst the simulation functions as expected, the expandability of certain aspects of code in the power controllers and evaluations was limited and I would have prioritised to fix them first.

If I had to create the project again from scratch, I would choose to approach the problem in the same order but apply the new skills I learnt to improve upon it. I would build the environment and XR capabilities first and then focus on the modular components and then finish with the evaluations, puzzle and truth table appending. I would however attempt to consolidate multiple scripts together such as the power controllers and inputs/outputs. The unnecessary bloat that some of the code may inhibit the ability to develop further.

Overall, I learnt an entirely new skillset of working with VR in the Unity engine. The project highlighted how much I still have to learn in writing code and project architecture but served as a good learning step for future VR projects. I plan to carry forward this knowledge and use Unity in my professional career and am grateful that I was given the chance to experiment and learn Virtual Reality development as part of the course.

References

1. *Logic Gate Simulator*. Academo 09/05/2022]; Available from: <https://academo.org/demos/logic-gate-simulator/>
2. *Locig.ly*. Bowler Hat LCC 09/05/2022]; Available from: <https://logic.ly/demo/>
3. Labarete, R-A. Ortiz, J M C. Villaverde, J F. – *VR in Logic Circuits and Switching Theory Laboratory 1 using Support Vector Machine* 10/05/2022]; Available from: <https://dl.acm.org/doi/pdf/10.1145/3470716.3470720>
4. Zen Marmot Digital. *Logic Level Test* 10/05/2022]; Available from: https://www.youtube.com/watch?v=Mwwy_58NKXE&ab_channel=ZenMarmotDigital
5. Zen Marmot Digital. *Logic Gates VR playlist* 10/05/2022]; Available from: https://www.youtube.com/watch?v=pgyIpuUIKU&list=PLA9W3lkRtAfOe6XoSYNxxkwVTujofrBoW&index=1&ab_channel=ZenMarmotDigital
6. *Unity Game Engine* 10/05/2022]; Available from: <https://unity.com/>
7. Unity Technologies *Unity Wins Best Engine at Develop Awards, Recognized by AlwaysOn Global 250 and OnMobile Top 100* 10/05/2022]; Available from: <https://www.globenewswire.com/fr/news-release/2012/07/12/1252010/0/en/Unity-Wins-Best-Engine-at-Develop-Awards-Recognized-by-AlwaysOn-Global-250-and-OnMobile-Top-100.html>
8. *Unity Platform* 10/05/2022]; Available from: <https://unity.com/products/unity-platform>
9. Unity Documentation *Unity Scripting Reference* 10/05/2022]; Available from: <https://docs.unity3d.com/2020.3/Documentation/ScriptReference/index.html>
10. Unity Manual *Getting started with VR development in Unity* 10/05/2022]; Available from: <https://docs.unity3d.com/2020.3/Documentation/Manual/VROverview.html>
11. Unity Manual *XR Interaction Toolkit* 11/05/2022]; Available from: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/index.html>
12. Unity Manual *XR Grab Interactable* 11/05/2022]; Available from: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/xr-grab-interactable.html>
13. MetaQuest *Explore Oculus Quest* 10/05/2022]; Available from: <https://www.oculus.com/quest/features/>
14. Valve *Valve Index* 10/05/2022]; Available from: <https://www.valvesoftware.com/en/index>
15. Unity Documentation *MonoBehaviour Update* 11/05/2022]; Available from: <https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html>
16. International Data Corporation *AR/VR Headset Shipments Grew Dramatically in 2021, Thanks Largely to Meta's Strong Quest 2 Volumes, with Growth Forecast to Continue, According to IDC* 11/05/2022]; Available from: <https://www.idc.com/getdoc.jsp?containerId=prUS48969722>
17. Unity Manual *Locomotion XR Interaction Toolkit* 12/05/2022]; Available from: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/manual/locomotion.html>
18. Unity Manual *XRRayInteractor XR Interaction Toolkit* 12/05/2022]; Available from: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.0/api/UnityEngine.XR.Interaction.Toolkit.XRRayInteractor.html>

19. Unity Documentation *Send Message Upwards* 12/05/2022]; Available from: <https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Component.SendMessageUpwards.html>
20. Unity Documentation *Broadcast Message* 12/05/2022]; Available from: <https://docs.unity3d.com/2020.3/Documentation/ScriptReference/Component.BroadcastMessage.html>
21. Unity Documentation *Emissive Materials* 12/05/2022]; Available from: <https://docs.unity3d.com/2020.3/Documentation/Manual/lighting-emissive-materials.html>
22. Unity Documentation *Invoke* 12/05/2022]; Available from: <https://docs.unity3d.com/2020.3/Documentation/ScriptReference/MonoBehaviour.Invoke.html>
23. Blender 13/05/2022]; Available from: <https://www.blender.org/>