University of Nebraska - Lincoln
DigitalCommons@University of Nebraska - Lincoln

Department of Electrical and Computer Engineering: Dissertations, Theses, and Student Research

Electrical & Computer Engineering, Department of

5-2024

VR Circuit Simulation with Advanced Visualization for Enhancing Comprehension in Electrical Engineering

Elliott Wolbach
University of Nebraska-Lincoln, ewolbach2@huskers.unl.edu

Follow this and additional works at: https://digitalcommons.unl.edu/elecengtheses

Part of the Artificial Intelligence and Robotics Commons, Computer Engineering Commons, Educational Technology Commons, Engineering Education Commons, Graphics and Human Computer Interfaces Commons, and the Other Electrical and Computer Engineering Commons

Wolbach, Elliott, "VR Circuit Simulation with Advanced Visualization for Enhancing Comprehension in Electrical Engineering" (2024). *Department of Electrical and Computer Engineering: Dissertations, Theses, and Student Research*. 151.
https://digitalcommons.unl.edu/elecengtheses/151

This Article is brought to you for free and open access by the Electrical & Computer Engineering, Department of at DigitalCommons@University of Nebraska - Lincoln. It has been accepted for inclusion in Department of Electrical and Computer Engineering: Dissertations, Theses, and Student Research by an authorized administrator of DigitalCommons@University of Nebraska - Lincoln.

VR CIRCUIT SIMULATION WITH ADVANCED VISUALIZATION FOR ENHANCING COMPREHENSION IN ELECTRICAL ENGINEERING

by

Elliott Wolbach

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Telecommunications Engineering

Under the Supervision of Professor Hamid R. Sharif-Kashani

Lincoln, Nebraska

May, 2024

VR CIRCUIT SIMULATION WITH ADVANCED VISUALIZATION FOR ENHANCING COMPREHENSION IN ELECTRICAL ENGINEERING

Elliott Wolbach, M.S.

University of Nebraska, 2024

Advisor: Hamid R. Sharif-Kashani

As technology advances, the field of electrical and computer engineering continuously demands innovative tools and methodologies to facilitate effective learning and comprehension of fundamental concepts. Through a comprehensive literature review, it was discovered that there was a gap in the current research on using VR technology to effectively visualize and comprehend non-observable electrical characteristics of electronic circuits. This thesis explores the integration of Virtual Reality (VR) technology and real-time electronic circuit simulation with enhanced visualization of non-observable concepts such as voltage distribution and current flow within these circuits. The primary objective is to develop an immersive educational platform that makes understanding these abstract concepts intuitive and engaging. This research thesis implements the design and development of a VR-based circuit simulation environment. By leveraging VR's immersive capabilities, users can physically interact with electronic components, observe the flow of electrical signals, and manipulate circuit parameters in real-time. Through this immersive experience, learners can gain a deeper understanding of different fundamental electronic principles, transcending the limitations of traditional two-dimensional diagrams and equations. Furthermore, this thesis focuses on the research and implementation of advanced and novel visualization techniques within the VR environment for non-observable electrical and electromagnetic properties to provide users with a clearer and more intuitive understanding of electrical circuit concepts. Examples include color-coded pathways for current flow and dynamic voltage gradient visualization. Additionally, real-time data representation and graphical overlays are researched and integrated to offer users insights into the dynamic behavior of circuits, allowing for better analysis and troubleshooting.

@ Copyright 2024, Elliott Wolbach

To my family

Acknowledgments

Thank you to my family and friends for their unwavering support through this journey. I would like to thank my advisor. Prof. Hamid Sharif, for all of his guidance and wisdom, which was invaluable in shaping the direction of my research. Dr. Hempel, your meticulous attention to detail and insightful feedback have significantly enriched this work. And to Dr. Ostler for your expertise and encouragement, which helped point my research in the right direction to what it is today.

Contents

List of Figures xiii

List of Acronyms viii

1 Introduction 1

1.1 Virtual Reality . . . 2

1.1.1 Headsets and Controllers . . . 2

1.1.2 Uses of Virtual Reality Technology for Training and Education . . . 2

1.2 Unity . . . 3

1.3 Circuit Analysis . . . 4

1.4 Thesis Organization . . . 5

2 Problem Statement 6

2.1 Simulation Framework . . . 6

2.2 VR Hardware in Use . . . 7

3 Literature Review 9

4 Methodology 12

4.1 Development Cycle . . . 13

4.2 Virtual Reality Device Rig . . . 15

4.3 Creation of Virtual Environment . . . 16

4.3.1 Component Models Development . . . 17

4.3.2 Breadboard Model Development 19

4.3.3 User Interfaces / Menus 19

4.4 Simulation Implementation 20

4.4.1 Main program for Simulation Control 20

4.4.2 Component Programming 22

4.4.2.1 Analog Element Script Development 22

4.4.2.2 Digital Element Script Development 26

4.5 Creation of Visualization Elements 27

4.5.1 Universal Visual Pipeline 27

4.5.2 Line Render 28

4.5.3 Color Changing 29

5 Results and Discussion **30**

5.1 Results 30

5.1.1 Visualization of Current 30

5.1.2 Visualization of Voltage 30

5.1.3 Digital Logic Implementation 31

5.1.4 User Interactions 31

5.1.5 Real-time Simulation Updates 33

5.2 Discussion 34

6 Conclusion and Future Work **36**

6.0.1 Conclusion 36

6.0.2 Future Work 37

Bibliography **38**

A Code **40**

List of Figures

1.1 Examples of KVL and KCL [3] . . . 5

2.1 HTC Vive . . . 7

2.2 Meta Quest 2 . . . 8

4.1 Development Cycle for Testing Changes off and on hardware . . . 14

4.2 Left: Controller, Right: Tracked Hand Control . . . 15

4.3 VR Rig Prefab Structure . . . 16

4.4 Example of the structure for the Resistor GameObject . . . 17

4.5 The green wire meshes indicate the positions of the colliders attached to the main body and legs . . . 18

4.6 Examples of the Resistor color bands in different configurations. . . . 18

4.7 Left: Breadboard surface model, Right: Breadboard connectors for physics-based detection . . . 19

4.8 Flowchart of "Create_net_list" script . . . 21

4.9 Shader Graph for custom dotted texture . . . 27

4.10 Example of Line Renderers showing current path, dots appear to be moving from left to right . . . 29

5.1 Simulated resistor network with current and voltage visualized . . . 32

5.2 Top down view of the two digital ICs wired to the breadboard . . . 32

5.3 Two examples of missing frames and frame errors caused by delays in Unity frame generation . . . 33

5.4 Video of Simulator Functionality (MP4 89.1 MB) 35

List of Acronyms

VR Virtual Reality

AR Augmented Realitys

HMD Head Mounted Device

UI User Interface

KVL Kirchhoff's Voltage Law

KCL Kirchhoff's Current Law

CHAPTER 1

Introduction

In electrical and computer engineering education, students can encounter difficulties comprehending abstract concepts and visualizing complex electrical circuits. Traditional teaching methods, such as using two-dimensional diagrams and equations, can lack the ability to convey the dynamic behavior of circuits or how realistic circuits would be built. Additionally, if students do not have access to hands-on laboratories, they will lack an immersive learning environment that allows them to physically interact with the electronic circuits. As a result, students may struggle to develop an understanding of fundamental principles in electrical circuits, hindering their ability to connect theoretical knowledge to practical engineering. To address these challenges, this thesis explores the use of virtual reality and advanced visualization techniques to enhance the learning experience in electrical and computer engineering education.

The main focus of this thesis is to develop a new virtual reality application that can be used for introductory electrical and computer engineering classes that visualize fundamental concepts like voltage and current. These invisible concepts can be a source of confusion for students but are integral to understanding how circuits operate. Using Unity, an application will be designed that will allow students to design circuits and simulate different conditions with the ability to visualize the invisible aspects of the circuits.

1.1 Virtual Reality

Virtual Reality (VR) allows users to view and interact with a virtual environment generated by a computer application. This virtual environment allows users to be immersed in places and activities and experience interactions firsthand.

1.1.1 Headsets and Controllers

Multiple platforms allow users to experience virtual reality both in how to view virtual spaces and interact with them. Using a head-mounted device (HMD), users look at two screens, one per eye, which allows for a depth of field to be rendered. Different HMDs have different aspects that users should be aware of. Some HMDs are stand-alone devices, which means that all of the hardware and software to render and run applications are included in the headset. Apposed to this are tethered HMDs that need to be connected to an external computer in order to work. In both cases, HMDs will have different screen resolutions and refresh rates that affect how well users are immersed in the virtual content. For interacting with objects and menus in VR, there are two main options, the first is using a tracked controller pad. These controller pads have joysticks and buttons that allows the user to select objects and navigate menus within the system. Either using tracking sensors on the headset itself or external tracking sensors, the position of the controller in real space is mapped to its position in virtual space. An alternative method of interacting involves tracking the hands of the user in real time and allows for a virtual replica of their hands to be mapped into virtual space. This allows users to have more dexterity when interacting with objects, pinching and pointing their fingers.

1.1.2 Uses of Virtual Reality Technology for Training and Education

Many studies have looked into the effects that virtual reality has on education and training. In [1], the authors reviewed how VR has been used for training in different domains. They

gave examples from training for first responders, medical, military, and transportation fields. VR training in all of these fields provides several benefits from traditional training. Firstly, training may not be possible because of risks associated with the training objective. For medical training, for high-risk operations, VR enables surgeons and other medical staff to gain experience without any risk to a patient. Other research will be discussed in Chapter 3 with the works more specifically related to this thesis.

1.2 Unity

Unity [2] is a development platform utilized to develop interactive 2D, 3D, VR, and AR applications. With a robust set of features, Unity gives developers various tools to be used via scripting in C and also from editor menus. Central to Unity's functionality are its physics engine, lighting/rendering system, and integration with multiple platforms.

Unity incorporates a sophisticated physics engine that enables developers to simulate realistic physical interactions within virtual environments. Leveraging rigid body dynamics, collision detection, and other user-defined constraints, this allows developers to create dynamic and interactive spaces where objects can behave according to real-world principles. Developers can define how each object in the scene is affected for example, choosing to make one object completely weightless while another can be affected by gravity.

Integral to the aesthetic appeal and realism of digital environments is Unity's lighting effects. Developers can utilize dynamic lighting effects, global illumination, and real-time shadows to enhance the visual fidelity of scenes. Additionally, Unity offers a range of post-processing effects, particle systems and shaders to further augment the visual quality of rendered objects and scenes.

Unity supports multiple programming languages, but C is the primary language used to code scripts that control how interactions take place, logic, and other behaviors. As previously mentioned, Unity also provides developers with an intuitive scripting interface

and extensive documentation. It allows those with limited coding experience to code the API through a user interface instead of writing the code themselves. This also allows for more experienced developers more freedom of how to create interactions and behavior. Unity supports multiple platforms such as Windows Desktop, Mac OS Desktop, Android, and iOS among others. Unity has robust support for VR development through the support for Android platforms. Developers can integrate VR hardware and peripherals with Unity projects through dedicated APIs and frameworks. This facilitates the creation of immersive VR applications ranging from games to simulations to training modules, as mentioned in the section above.

Finally, Unity offers extensive API support, providing developers with access to a vast array of libraries, frameworks, and plugins to extend the engine's capabilities. These can be third-party assets, platform-specific features, and custom development tools, Unity accommodates a wide range of development needs.

1.3 Circuit Analysis

A large concept in this thesis is circuit analysis. For students learning about electrical and computer engineering, they must start out with the basics. Circuit analysis, on a basic level, is the ability to discern unknown voltages and currents acting on a circuit. Two main concepts intrinsic to circuit analysis are Kirchhoff's Voltage Law (KVL) and Kirchhoff's Current Law (KCL). KVL proves that in a circuit, all of the voltage drops from components like resistors, and voltage increases from voltage supplies will be equal to 0. On the other hand, KCL focuses on current, stating that all currents entering a node must equal the total current exiting that same node. Examples of these concepts can be seen in Figure 1.1.

In my experience as a TA for an introductory circuits analysis class, students sometimes have difficulty visualizing these voltage drop and current paths when looking at real-life circuits. This is a barrier to the student's progress in understanding these fundamental

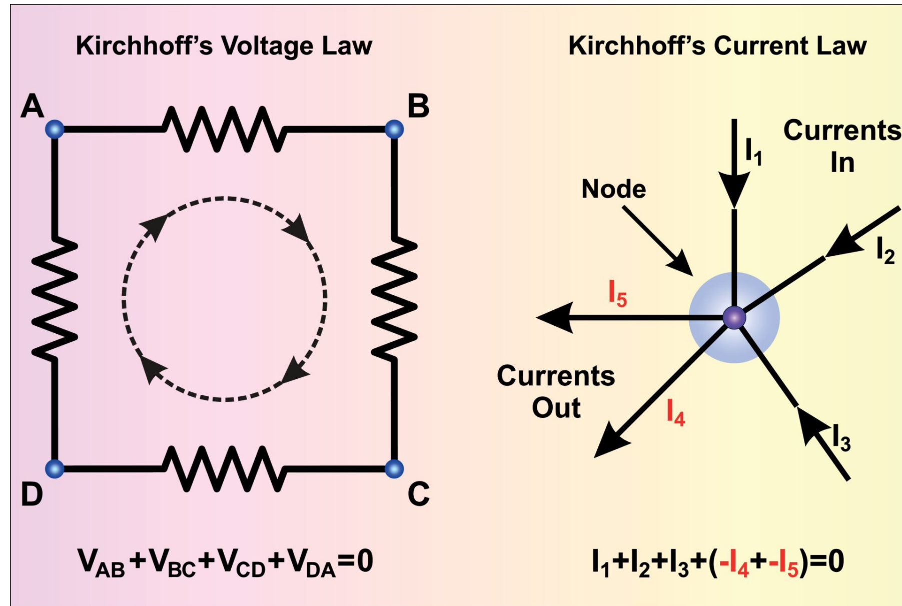

Figure 1.1: *Examples of KVL and KCL [3]*

concepts.

1.4 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 presents the problem statement of this thesis, focusing on implementing a VR circuit simulator with advanced visualization. Chapter 3 will cover related works in the domain of virtual reality and electrical engineering education. Chapter 4 presents the proposed application and new visualization methods explored. Chapter 5 will present the finalized application and discuss different elements. Finally, Chapter 6 concludes this thesis.

CHAPTER 2

Problem Statement

In this thesis, we propose a practical VR Circuit simulator that can be used to simulate different circuits and visualize electrical aspects. The main focus is developing a visual system that is able to integrate with an immersive virtual environment with interactive electrical circuit components. The scope of simulation is limited to DC components like resistors, switches, and LEDs, which have the additional capability of modeling capacitors and a few digital integrated circuits. In order to achieve an application that can be used for education, the simulator must be able to perform circuit simulation with a low level of latency so as not to disrupt the VR user's experience. High levels of latency can lead to screen tearing. To achieve these goals, a virtual environment will have to be designed from the ground up. Using an existing simulation framework to perform the majority of the circuit analysis calculations and integrating that with virtual circuit components and UI designed in Unity.

2.1 Simulation Framework

An important aspect of this thesis is the simulation of circuits; however, the creation of a new simulation framework is out of scope. The main features needed from a simulator are a light footprint and flexible functionality, which allows it to be run natively on VR hardware.

Also, this simulator must be able to simulate multi-component circuit voltage points in a time frame that does not disrupt the VR experience. The simulator that was determined to fit all of the above criteria was Spice# [4]. This is a C# based circuit simulator that was designed after the original Spice program created at the University of California, Berkeley [5]. Written in C#, it is compatible with Unity out of the box. Spice# is also very lightweight, with the source files only being 2.61MB. All of Spice#'s source code is open source along with a well-documented API that allows for easy programming for our application.

2.2 VR Hardware in Use

There are many different VR devices on the current market, each with pros and cons. The HTC Vive was originally used for early development. It is a tethered HMD with a tracked controller via separate wall-mounted tracking modules.

Figure 2.1: *HTC Vive*

While this platform worked well for initial designs, the tethered cable limited mobility, and if the tracking modules lost sight of the controllers, the user would lose track of the controllers. With these shortcomings, we moved development to the Meta Quest 2 headset

[6]. This HMD is a fully stand-alone platform that can run applications independently with additional features like hand tracking and vision pass-through. Development did not change after the initial setup of a Meta Developer account and downloading the Meta Quest Developer Hub application, allowing for the headset to be connected directly to the computer for streamed development through Unity's editor hub or for built packages to be uploaded to the Quest 2 to be run independently.

Figure 2.2: *Meta Quest 2*

CHAPTER 3

Literature Review

Virtual Reality has been a point of interest to be used as a tool to help promote learning, and COVID-19 has played a prominent role in advancing the interest in remote labs or home learning for electrical labs in recent years. Various methods have been researched and attempted, from fully immersed VR with head-mounted devices to augmented reality applications and web-based interfaces.

The authors of [7] created a realistic lab space using 3ds Max and Unity that simulated integrated circuit manufacturing and testing. Using a web-based interface, students were able to interact with the VR environment, carry out experiments, and were able to get feedback on their performance. Even though not as immersive as an HMD interface, students' grades showed improvement after completing the virtual experiments. Some feedback shows that some students experience issues with network connectivity and long loading times.

Cao et al. [8] introduced a guided virtual reality education simulation that allowed primary school students to learn about basic analog circuit concepts. Basic components of switches and light bulbs were displayed with 3d models, and a simplified visualization of current was shown through an overlay above the wires. Forgoing realistic circuits for simple ones allows the younger students to grasp the different topics, with 90% of students showing improvement from a pre-experiment quiz. Jiang et al. [9] developed a similar application with the ability to allow multiple users to experiment on the same circuit. Like in [8], the

circuitry was simplified blocks to allow for easier configuration of simple analog circuits.

Similar approaches were taken by Khairudin et al. [10] with an interactive virtual laboratory that focused on visualizing and simulating logic circuits in real-time. Taking a simplified approach circuits were shown as 2D images, the circuits inputs were controlled by switches that could be toggle allowing for students to try different combinations.

In the article [11], the authors describe their approach to implementing an application that can read a resistor network topology and generate a netlist. Then, using LTspice, the properties of the individual components are outputted to the terminal of Unity.

Authors in [12] used augmented reality (AR) to create an immersive simulator. AR allows users to see the surrounding space and overlays virtual models and UI to create a mixed-reality experience. Using tokens to identify different circuit components and a simulator back end, users could change circuit parameters and see real-time changes in graphs depicting different output voltages and power. This approach focused on being a low-cost solution. The platform was an Android mobile device or table, which allowed for easy-to-use functionality.

In a different approach, Zamojshi et al. [13] created a virtual reality "escape room," allowing students to learn through a series of different puzzles that were related to electronics and telecommunications. All of the interactive modules were on the walls of the virtual room, allowing users to input answers on keypads. They provided that users' scores on a quiz of questions related to the topics improved after using the application and remained improved 2 weeks after when the survey was given again. This proves that VR education applications can be used to help students learn STEM-related concepts.

In conclusion, while various VR and AR applications have been developed to facilitate the learning of electrical concepts, there remains a gap in providing a fully immersive, hands-on circuit simulation and visualizing basic concepts like voltage and current. These approaches ranging from web-based interfaces to guided VR simulations have shown promise in improving students' understanding and performance. However, issues such as

simplified circuit models and limited immersion persist across these studies. This thesis aims to address these limitations and incorporate advanced visualization and immersive models and controls to simulate circuits.

CHAPTER 4

Methodology

This thesis aims to create an immersive VR application that visualizes different electrical properties when simulating circuits. As mentioned in the literature review, there have been projects using VR to create education tools for electrical and computer engineering, without fully utilizing VR's capabilities. Before the visualization elements could be developed, a base platform needed to be created in Unity. The Unity engine has different builds that are not all compatible with each other, the build that will be used in this thesis is 2021.3.9f1. Before starting with this thesis's methodology, a few concepts need to be defined.

- GameObjects: In Unity, any object that is in a scene is referred to as a GameObject. GameObjects can have the structure of a parent GameObject with children and grandchildren GameObjects. Generally, these are the fundamental entities.

- Prefab: Prefabs are templates of GameObjects that can be used multiple times within scenes. They can be reconfigured with models, components, and scripts. When brought into a scene, all instances of a prefab are initialized to a default state. What is utilized in the scenes is the ability to change a specific prefab object at runtime without affecting all of the same prefab instances.

- Components: Components are elements that make up GameObjects. These classes

or scripts define specific behavior, functionality, or properties of GameObjects. An example of a component is Transform, where the GameObject is located in the scene given in xyz coordinate and rotation fields.

4.1 Development Cycle

To have an efficient development cycle, a general procedure was created that can be seen as a flow chart in Figure 4.1. The main development occurred in Unity's Editor and VScode for writing scripts in C#. For small changes, an HMD simulator was used that allowed for the inputs and movement of a VR system to be mimicked by using a keyboard and mouse within Unity's Editor. Testing changes like lighting or other graphical/model changes were beneficial to ensure the settings were correct. To test directly on the device, the Quest 2 was connected to the computer, and the "Quest link" was started that allowed for programs to access the HMD as a display device. Then, when starting the application from the Unity Editor, it will display the camera view to the headset and track the inputs from the controllers. This allowed for the testing of inputs, UI, and other GameObjects to be manipulated and tested for different behaviors. With the application running on the computer, Unity's developer console was able to be used to debug scripts and check different states of the GameObjects. Finally, for a full system test, the relevant scenes were selected and built with specific settings for XR HMDs, and an Android Package, APK, was then uploaded to the Quest 2 either directly from Unity or uploading the .apk through the Meta Quest Developer Hub. With this, the application could be run directly on the Quest 2's hardware, and all functionality could be tested. To version control this thesis, TEL's Gitlab was utilized, as well as Unity's own cloud storage, Gitlab, for major changes, while Unity's system was used for day-to-day changes. This allowed for multiple computers to be used for development.

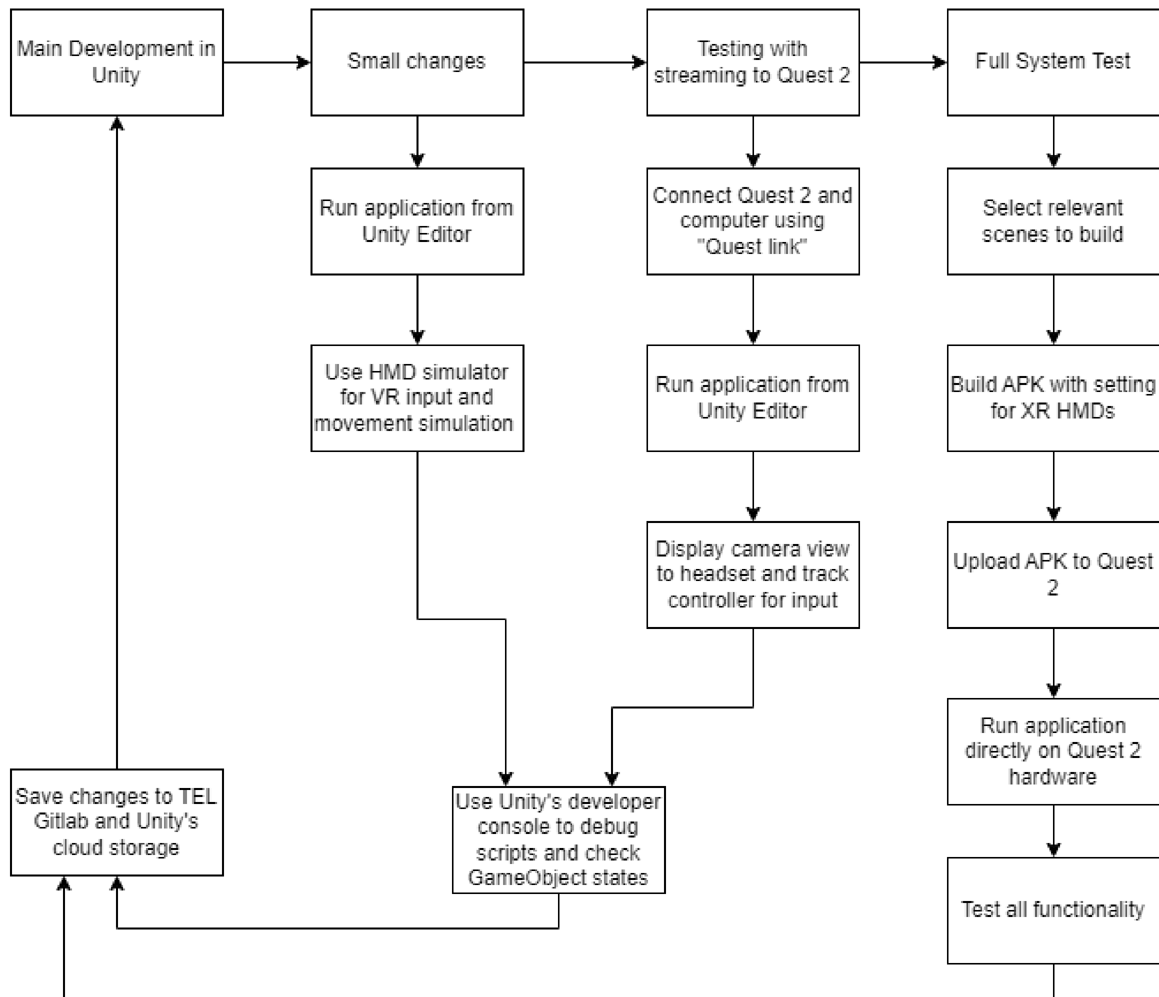

Figure 4.1: *Development Cycle for Testing Changes off and on hardware*

4.2 Virtual Reality Device Rig

In order for the user to be placed in a 3d virtual space, the program needs to communicate with the HMD sensors. This is achieved through a virtual reality device rig in Unity. Seen below in Figure 4.3 is the Prefab structure of the VR Rig, this includes all of the the manager that are responsible for handling user inputs, virtual interactions with objects, and other events. The substructure of "XR Origin" contains the main camera and controller/hand components. These components allow users to see and interact with the virtual space in various ways, poking, grabbing, and pointing rays to interact at a distance. This prefab allows for the use of both controller and hand-free interactions. As seen in Figure 4.2, the system will automatically switch between controllers and hands-free control, rendering the applicable model for each mode.

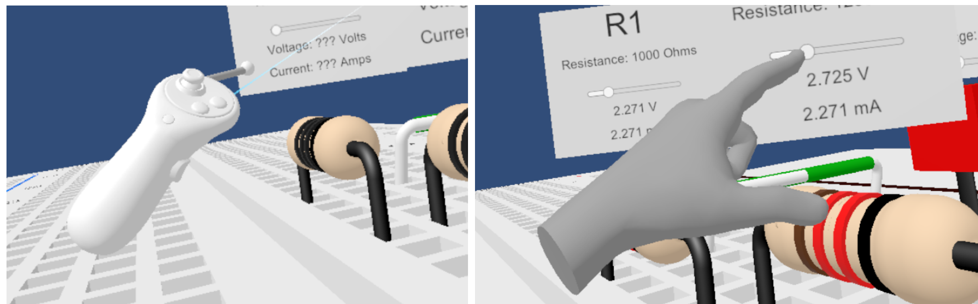

Figure 4.2: *Left: Controller, Right: Tracked Hand Control*

This prefab is present in all scenes as this represents the user in the virtual space, tracking head movement and interactions. In the Component Programming section, we will discuss in more depth how each GameObject is prepared to be interactive.

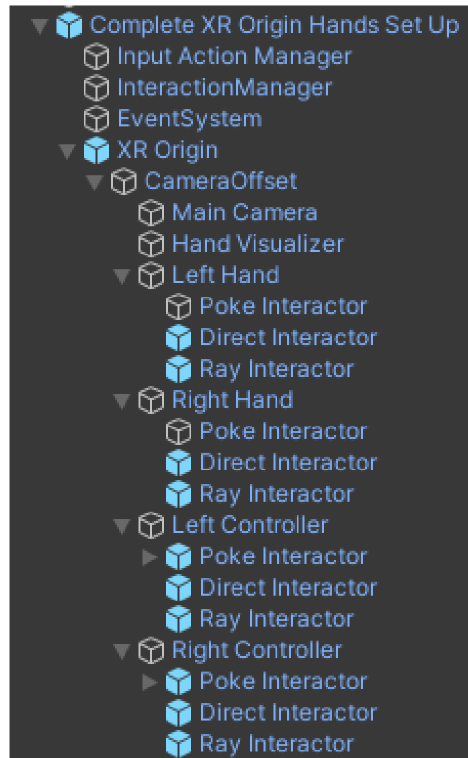

Figure 4.3: *VR Rig Prefab Structure*

4.3 Creation of Virtual Environment

When a Unity project is created, there are no GameObjects in the scene. Any models need to be imported and added to the scene as GameObjects. In this case, the main objects in our environment are electrical components like resistors and wires, as well as the breadboard that will be used to create the functional circuits. To save time, models of these objects were found on online repositories [14],[15],[16],[17] and modified to the correct size scales using Fusion360 CAD.

4.3.1 Component Models Development

All of the electrical elements' GameObjects share a similar structure, with a parent GameObject representing the main body of the electrical part and children GameObjects representing the legs.

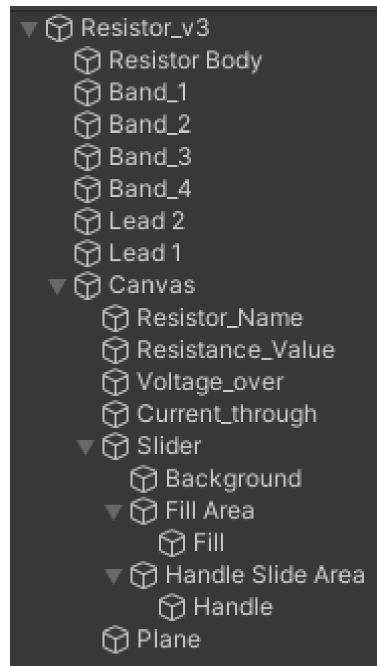

Figure 4.4: *Example of the structure for the Resistor GameObject*

The parent GameObject contains multiple components that control the behavior and interaction elements. The main script containing all of the electrical properties used by the simulator is attached to the parent GameObject. This script will be explained in more depth in the Component programming section. Other notable scripts are "Transparent_view" and "Color_change_on_voltage," which allow for the visualization of voltage and current elements. In addition, the parent GameObject has Rigidbody and collider components. These components work with the interaction manager to allow for the GameObjects to be able to be picked up by the user's controllers or tracked hands. For the child GameObjects that represent the leads of the electrical components, each has a Rigidbody and collider component and a script that manages the contact points of the legs and the breadboard.

The Resistor prefab has additional children components that represent the color bands

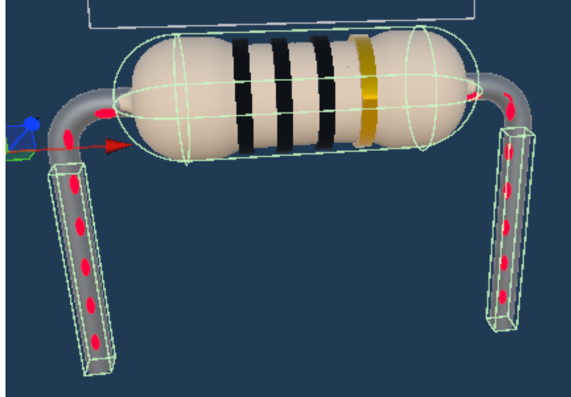

Figure 4.5: *The green wire meshes indicate the positions of the colliders attached to the main body and legs*

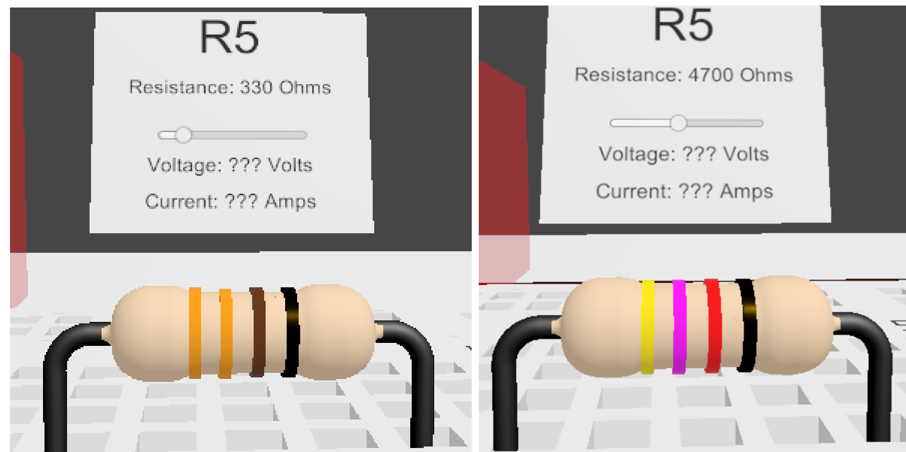

Figure 4.6: *Examples of the Resistor color bands in different configurations.*

of a resistor. A script connected to all bands controls each band's color depending on the parent resistor's resistance value.

The models of the wires, capacitors, switches, and LEDs are created in a similar fashion, each element is assigned Rigidbody and collider components. With these components, the physics engine can simulate life-like interactions between elements and the breadboard. This means that if a resistor lead is not seated correctly, not making full contact with the breadboard, the simulator will simulate this open circuit condition.

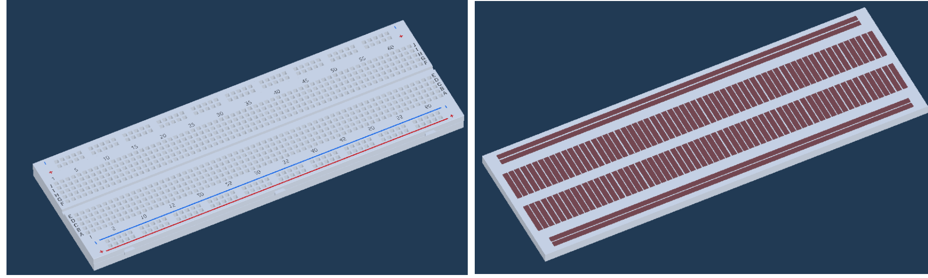

Figure 4.7: *Left: Breadboard surface model, Right: Breadboard connectors for physics-based detection*

4.3.2 Breadboard Model Development

The breadboard model uses a top plate with all of the holes for the electrical component legs and interconnectors that mirror a breadboard's connection points in real life. This allows the simulator to detect what components are connected together when the simulator is creating a netlist. The top plate has a Mesh Collider, which allows the model to have collision points that conform to the mesh of the object rather than a simple box or cylinder. This allows the components' legs to slot into the breadboard's holes like in real life. The inner connectors act as wires that connect the electrical components, with each column representing a node. A script tracks the node voltage and number of components in contact with that connector.

4.3.3 User Interfaces / Menus

Also, with the GameObjects that represent electrical components, different menus were implemented to allow users to control various aspects of the simulator. Each of the electrical components except wires has a UI that represents the voltage over and current through that specific component. Also, this UI has a control slider for its electrical characteristics. For example, the resistor's UI has a slider that allows the resistance value of that resistor to be changed in the simulation. Other menus allow users to spawn in additional instances of the electrical components and navigate to different scenes that demonstrate various capabilities of the application.

4.4 Simulation Implementation

Using Spice# as the circuit simulator back end, each electrical component must be added to the simulation netlist to create the circuit topology. This process is controlled by the application's main script, which is responsible for parsing all active electrical components for values that the simulator needs and updating relevant fields after the simulation solves the circuit's nodal voltage.

4.4.1 Main program for Simulation Control

Whenever an electrical component's value is updated, the "Create_net_list.cs" script is invoked, rerunning the simulation. The simulator clears the old circuit and creates two arrays of all active components and breadboard wires that have a connection. The array of electrical components is parsed, and each object is assigned a name with an incrementing number tag. The relevant values are collected from the objects to create a netlist element with the Datatype corresponding to the electrical part. For example, a temporary variable is created with the Resistor class when adding a resistor to the simulation. The name of the resistor, connection points of the left and right legs, and resistance value are passed into the constructor. This variable is then added to the simulator's netlist. After all electrical components are added to the simulator, the output streams of the simulation are assigned to the active node wires. When the simulator calculates the node voltages, it will update the voltage_node field for each of the node wires. After the simulation is run, each of the circuit components is then updated. The analog components' update functions are called to use the node voltages to calculate the current through each, respectively. For digital logic, additional steps are taken to create voltage sources that act as the drivers for the IC's output pins. These new voltage sources are added to the circuit's netlist, and the simulation is rerun to update any of the components that are connected to the output pins. A flowchart of this method can be seen in Figure 4.8.

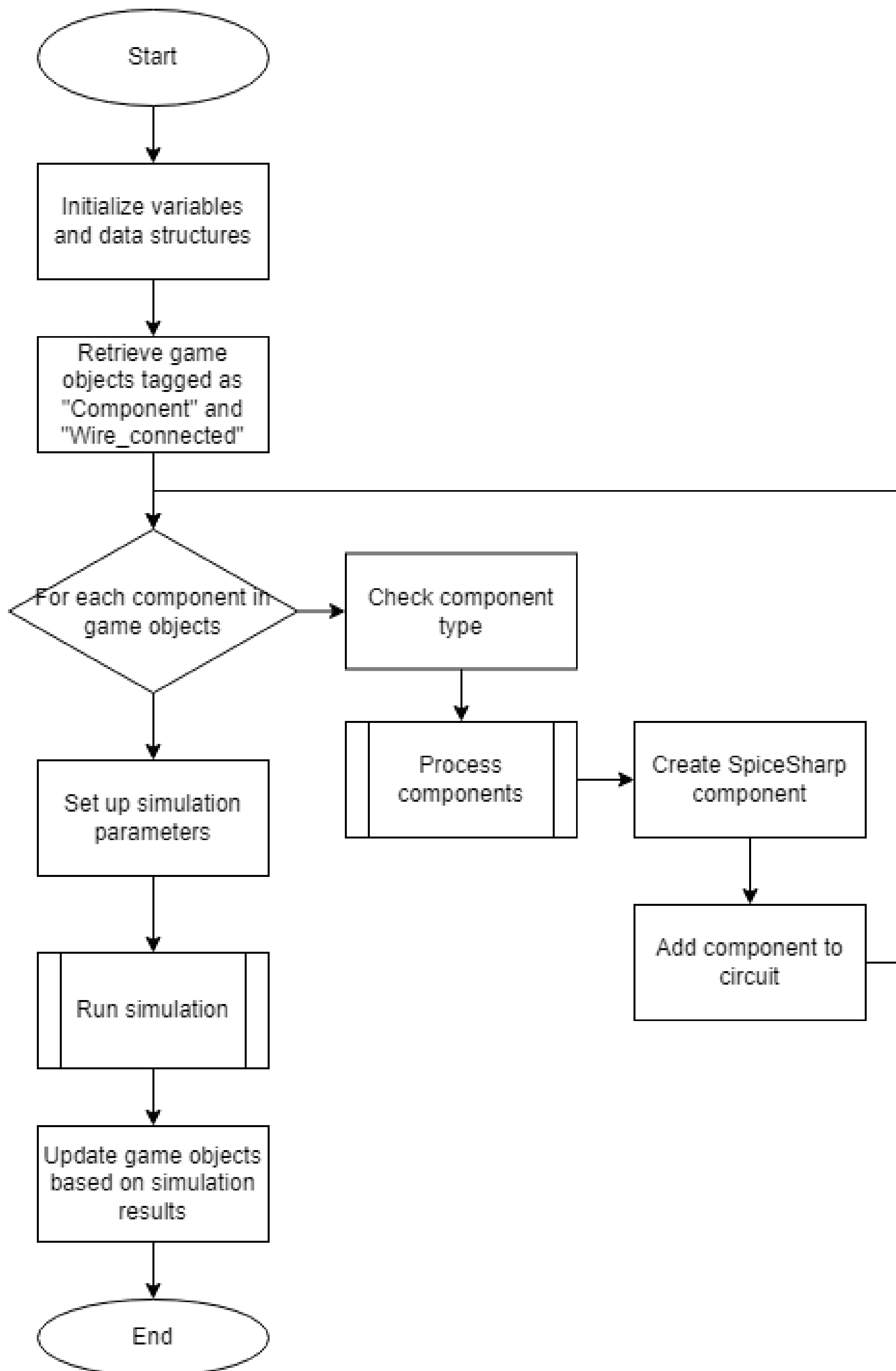

Figure 4.8: *Flowchart of "Create_net_list" script*

4.4.2 Component Programming

Similarly to the structure of the 3D models of the electrical components, each part has one or more scripts attached to it controlling various aspects of the electrical component. These scripts use both the Unity scripting API and the editor in order to interact with the GameObjects. Using the editor, entire GameObjects could be imported and edited within the code using the scripting API. An example of this is importing a model into a script and then using the Unity scripting API to change the texture color of the model.

4.4.2.1 Analog Element Script Development

All analog element programming consisted of scripts controlling the electrical characteristics, scripts interfacing with the physics engine for the legs, and scripts controlling the element's visualization aspects.

The electrical element of the resistor main script, "Resistor_info.cs", can be found in Appendix A, there are multiple fields: resistance, voltage_over, current_through, connection_A, and connection_B are public fields that represent the resistors resistance, voltage, and current measured across/through, and the physical connections of the legs of the resistor to the wire elements in the breadboard. For the element's UI, the fields txt, voltage_txt, and Current_txt are public fields of type "Text" from UnityEngine.UI namespace. This allows for the script to access those text fields to update them with the simulation values. The final two fields are 'material' and 'lineRenderer' of types "Material" and "LineRenderer," respectively, from the UnityEngine namespace. These fields are used to update the visualization components of the resistor. The scripts that handle these components will be discussed in the section 4.5.

Methods:

Start(): This method is built into the Unity scripting namespace. It is run only once

when the script is first activated. In this case, the resistor is initialized, and this function gets the 'LineRenderer' component attached to the main GameObject. The public variable material is assigned to the 'LineRenderer's material so that another function can access this parameter to change the visualization of the current.

resistor_info_print(): This is a debugging method that prints the resistance, voltage drop, current through, and connection information to the console of the Unity Editor. It was used to make sure that resistors were properly connected in the circuit and that the proper voltage and current values were being assigned.

resistor_update(): This method is responsible for calculating the voltage over and current through the resistor. It first looks to see if both legs are connected to the breadboard; otherwise, it will handle the NullRefernceExeption that is raised in the event that one or both of the legs are not connected. If both legs are connected, the script gets the voltage_node parameter from the connected breadboard wire. Using both of these node voltages, voltage_over and current_through are calculated. In order to determine the direction of current flow, a simple if statement checks to see if one voltage node is larger than the other and assigns the current flowing a negative sign to show that the current is flowing backward. Finally, this script uses the calculated current to create a Vector4 variable that is used to change the speed of the animation for current flow by changing the vector parameter of the 'material' variable. This is further discussed in section 4.5.2.

Another script that controls the behavior of the resistor GameObject is "Resistor_slider.cs". This script handles events generated when the slider bar on the resistor UI is changed. When

the slide bar is changed, this updates the resistance assigned to that specific resistor, changes the color bands and UI text accordingly, and reruns the simulation to update the voltage and current values to reflect the new resistance. This script has serialized fields to allow it to be configured in the Unity Editor. '_resistance_value' of type "Slider" allows for all aspects of the slider bar to be accessed. The field 'resistance' points to the UI Text element while the three GameObject fields connect to the 3 color bands on the resistor. Finally, there are two arrays, 'resistor_sizes' contains 10 standard resistor values, and ' band_colors' contains Color definitions for black, brown, red, orange, yellow, green, blue, magenta, grey, and white. Brown and orange needed to be defined separately, and Unity did not have those colors predefined.

Methods:

OnValueChanged(): This method is called whenever the slider bar is updated. With the slider bar having 10 different values, using the currently selected value, the resistance parameter in the 'Resistor_info' script is updated, passing the corresponding value from the 'resistor_sizes' array. Then, the 'Update_color_bands' method is called, passing the resistance value so the color band can be updated to match the new resistance value. Finally, the text of the Resistor's UI is updated to match the new resistance.

Update_color_bands(): Using the inputted resistance value, this script loops, dividing the input by 10 and incrementing a count to keep track. This count is used to determine the multiplier band color. Once the value is down to less than 99, converting this to a String and subtracting 48 (ASCII for 0) from the [0] index position in this string will create the integer number of the 'tens' place of this number. Similarly, the 'ones' place is calculated the same

way but with the [1] index character in the String. With these numbers, the corresponding color is selected from the 'band_colors' array, and the corresponding band's color is updated. These bands will update in real-time when the know of the slider is changed.

OnPointerUp(): This is a method defined by Unity that is triggered when the event system detects that the slider knob was selected and released. When the knob is released, the simulation is run to update the circuit's values with the new resistance value.

Finally, there is a script that interfaces with the physics engine to track connection points to the breadboard. This script is fairly straightforward and uses two methods: 'OnCollisonEnter' and 'OnCollisionExit.' 'OnCollisonEnter' is called when there is a collision between a GameObject with a Ridgebody component and another GameObject with a Collider component and visa versa. The 'collision' parameter contains the information about the collision; using this, the event is filtered to only look at collisions with GameObjects with the tag 'Wire' or 'Wire_connected.' These tags are assigned to the connection points within the breadboard GameObject. If the collision was with one of these tagged wires, the script will update the tag of the GameObject to 'Wire_connected' and access the 'contact' parameter within the 'Wire_info,' and the increment is counted. Then, depending on if the script is connected to "Lead 1" or "Lead 2," it will update the connection parameters 'connection_A' and 'connection_B' within the parent 'Resistor_info' script with the name of the collider with which it collided.

Various other electrical elements were implemented, such as LEDs, voltage sources, switches, wires, and capacitors. Each of these elements was constructed similarly to the resistor prefab, each with a main body component and children GameObjects representing the legs of the element. Contact scripts allowed for the elements to interact with the

breadboard. "Info" scripts allowed for the unique characteristics of the electrical elements to be stored and modified by the user using the UI component with a slider. One note: Spice# does not have a native model for a simple switch. In order to simulate a switch, the simulation model of a resistor was used; when the switch was open, it was assigned a negligible amount of resistance; while closed, the switch was assigned 1e10 ohms of resistance, ensuring all current was stopped.

4.4.2.2 Digital Element Script Development

Two different digital logic integrated circuit packages were created, one based on the 74ls08, a quad AND gate package, and the 74ls283, a 4-bit full adder with a fast carry package. These two ICs were selected to model a basic and complex IC to prove that the application can simulate digital logic. Initially, attempts were made to simulate the ICs at the transistor level with the Spice# simulator framework. While the transistor layout could be modeled and simulated with Spice#, the simulation calculations were not fast enough, taking 200ms, which caused disruptions in Unity's frame generation for the headset. The last image frame was generated would stay static, if the user turned their head, no new frames were generated, there would be black space in the unfilled regions. This was disorienting and would break the immersive experience of the Simulator. Deeming that this low-level simulation was not feasible for a VR application, an alternative solution was found. The solution was to abstract the circuitry of the ICs and interface with the rest of the virtual circuit with the input and output pins. The contact scripts are similar to the analog elements, except handling more connection points specifically 14 and 16 pins. The script for the 74ls08 has two input fields, 'connections' and 'logic_levels', which represent the physical leg connections and their respective logic levels. For the Update method, a dictionary 'outputs' is initialized to contain the connection points and logic levels. Two other variables, 'high' and 'low,' are initialized from the voltage levels are the VCC and GND pins. Then the script loops through all of the connections to update their logic values.

Then, through a series of logical operations, ANDing the gate inputs and assigning them to the output pin of that AND gate. Then, all of the voltage node levels are updated with the logic levels. These outputs are updated in the 'output' dictionary and returned to the method call. The implementation of the 74ls283 is similar, except the logical operation is a bit-wise addition with carry bits implementing the 4-bit adder.

4.5 Creation of Visualization Elements

4.5.1 Universal Visual Pipeline

Unity's Universal Render Pipeline (URP) is a lightweight rendering pipeline designed to optimize real-time rendering performance. [2] URP is optimized for performance across a wide range of hardware and suitable for projects targeting mobile platforms such as VR applications. With different configuration options, it aims to provide a balance between performance and visual quality. The two features of the URP that are mainly used are lightweight materials, shaders, and shader graphs. The lightweight material and shaders enable the Quest 2 to display the virtual environment and circuitry components with different and changing colors while not affecting performance. The shader graph allows for the creation of custom shaders through a visual interface. This was used to develop the custom

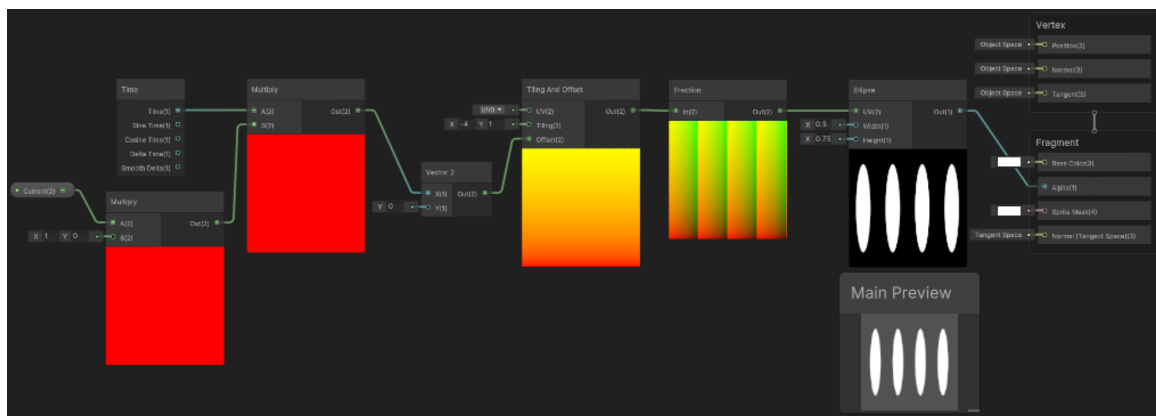

Figure 4.9: *Shader Graph for custom dotted texture*

shader for the animated line that depicted the current through electrical elements seen in Figure 4.9. When creating a shader, variables are allowed so outside scripts can affect the shader. In the figure above, multiple nodes were created and linked to create a moving dotted line. The first input is ’Current,’ which will be the current passing through the parent electrical element. This value is then multiplied by system time to create a vector that will increase based on the current. The use of the ’Tiling And Offset’ and ’Fraction’ nodes creates a scrolling texture that is driven by the time and current-based vector. This scrolling texture is then piped through an ’Ellipse’ node to create a dotted masked texture. Finally, the ’Vertex’ and ’Fragment’ nodes allow for the texture to be rendered by the pipeline when assigned as a material component. The finished shader can be seen in Figure 4.10 as the yellow dotted lines.

4.5.2 Line Render

Line Renders are a special component in Unity that allows for an input of set points representing points in 3d space and creates a straight line connecting said points. This component has parameters that can be adjusted, such as the width of the drawn line, light generation, and texture modes, among others. The custom texture was applied to the ’Material’ field, and a yellow color was selected to simulate the current. This made the line drawn look like a series of dots moving along the path of the line, giving the illusion of motion. When these lines renders are visible, this is referred to as the "Current View Mode." In this mode, the GameObjects needed to be sightly translucent, this visual change is handled by the ’Transparent_view" script. This script is applied to the affected parent GameObject; children GameObjects that are to be translucent are assigned to the renderers list, while GameObjects to be made invisible are assigned to the hidden_renderers list. If enabled, both lists will be parsed, respectively, made either translucent by changing the alpha or disabling the renderer, making that specific GameObject invisible. If not enabled, the inverse will happen, and all of the GameObjects in the lists will be reset to their normal

visual states.

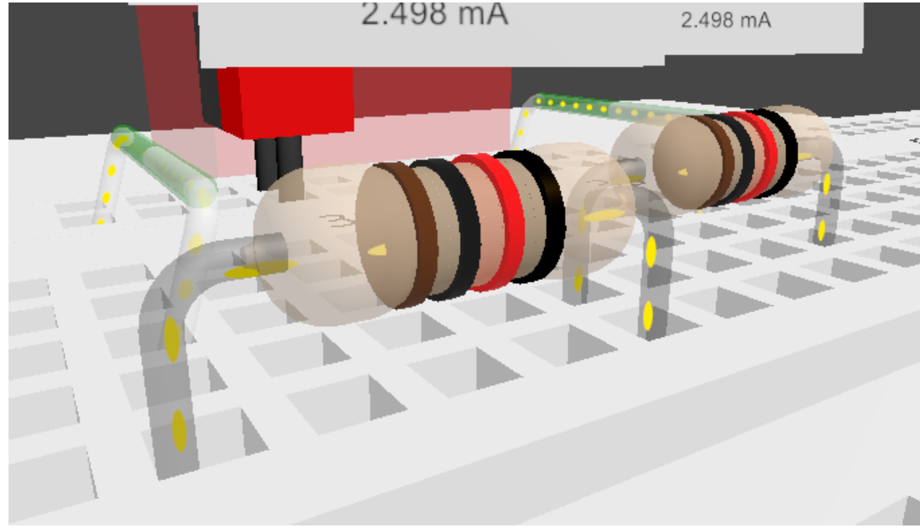

Figure 4.10: *Example of Line Renderers showing current path, dots appear to be moving from left to right*

4.5.3 Color Changing

As mentioned in the sections above, GameObjects with a mesh model have a component responsible for that mesh's color. Using Unity's API, this component can be modified by scripts, in this case the "Color_change_on_voltage". This script changes the color of the GameObject it is attached to based on that GameObject's voltage. When this script is initialized, it stores the GameObject's 'Renderer'; this is to revert the color of the GameObject if the script is disabled. If this script is enabled, the current voltage of the component is assigned to 'voltageOver'. Depending on the value of 'voltageOver', the script interpolates between the below threshold color and the middle color or between the middle color and the above threshold color. This interpolated color is then applied to the GameObject's 'Renderer,' updating the color for the next frame.

CHAPTER 5

Results and Discussion

5.1 Results

The result of this thesis is a functional virtual reality circuit simulator with visualization of electrical characteristics of voltage and current. Users can navigate a VR environment using the Quest 2 headset and interact with the electrical elements and menus using either the controllers or their free hands. With the different circuit elements modeled, users can create resistive circuits and RC circuits and use switches and LEDs to control inputs and outputs. Two modeled digital ICs allow for experimentation with digital logic.

5.1.1 Visualization of Current

As discussed in the previous chapter, visualization of the current was achieved by using Unity's Line Render with a custom texture. This gives the illusion of current flow, as shown in the figure below. With this visualization, the direction and magnitude of current in a circuit are no longer hidden from students as an invisible aspect of electrical circuits.

5.1.2 Visualization of Voltage

Similarly to current, voltage in terms of the voltage over circuit elements and how that relates to KVL allows users to see the cascade of voltage over a resistive network. Paired

with the voltage displayed on each element's UI, it would allow for circuit analysis to be performed. Students could double-check their homework by setting up a replica of the circuit they are working on in the simulator and adjusting the parameters to find the exact outputs for voltages and currents.

In Figure 5.1, both branch currents and element voltages are displayed. All of the resistors were configured to 1k Ohm. The "red" resistor has a 5-volt drop over it, making it have the largest voltage, followed by the "orange" resistor with a 3-volt drop. The middle "yellow" resistor has a voltage drop of 2 volts, and the remaining 2 resistors are green because they both have a voltage drop of 1 volt each. With this example, students can learn about concepts of what happens to the voltage and current of resistors in series-parallel networks.

5.1.3 Digital Logic Implementation

Two ICs implemented to prove digital logic simulation are the 74LS08 and the 74LS283. The 74LS08 is a quad 2-input AND gate package, and the 74LS283 is a 4-bit full adder with fast carry. Users can manipulate the inputs of these ICs by connecting wires to VCC or GND. As seen in Figure 5.2, the 74ls08 has two of the AND gates wired up with LEDs connected to the outputs. The output from pin 6 is also connected to an input of the second adder from the 74ls283. The 74ls283 is wired with inputs being connected to +5V or ground.

5.1.4 User Interactions

Multiple UI elements were implemented, allowing users to spawn additional electrical elements to add to their circuits. The electrical components have their own menus that provide simulation results and control over the different values, such as voltage, resistance, and capacitance. These can interact with both controllers or tracked hand control, giving the user the choice of how they want to use the application. Additionally, all of the electrical

Figure 5.1: *Simulated resistor network with current and voltage visualized*

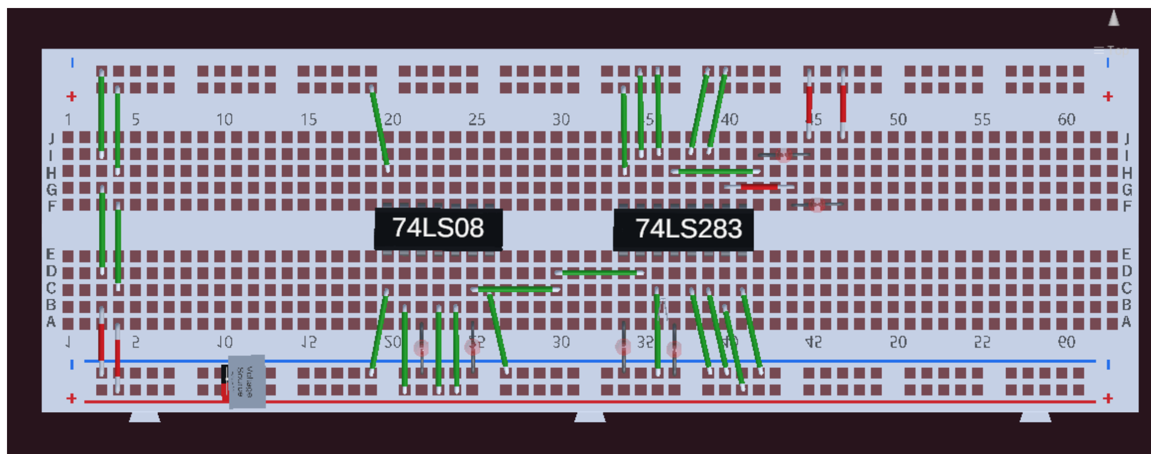

Figure 5.2: *Top down view of the two digital ICs wired to the breadboard*

elements are also physically simulated. This means that all of the

5.1.5 Real-time Simulation Updates

All of the visualization elements of this simulator are updated in real-time as soon as a user updates a parameter. This means that all of the voltage and currents are up-to-date. Additionally, focus was directed at ensuring that the simulation did not bottleneck the application. Unity has a linear pipeline for operations, first completing all of the Physics calculations, then moving to Game Logic. After all of the scripts complete execution in the Game Logic stage, Unity moves to the Rendering stage to generate a new frame. With the first state of simulating digital logic, there is enough delay with the simulation to desynchronize and cause the Quest 2 not to have enough frames to give an immersive view. Two examples of this behavior can be seen in Figure 5.3. This can be extremely disorienting; implementing the digital translation layer allowed for the simulation of the digital logic to be accelerated and fixed this issue.

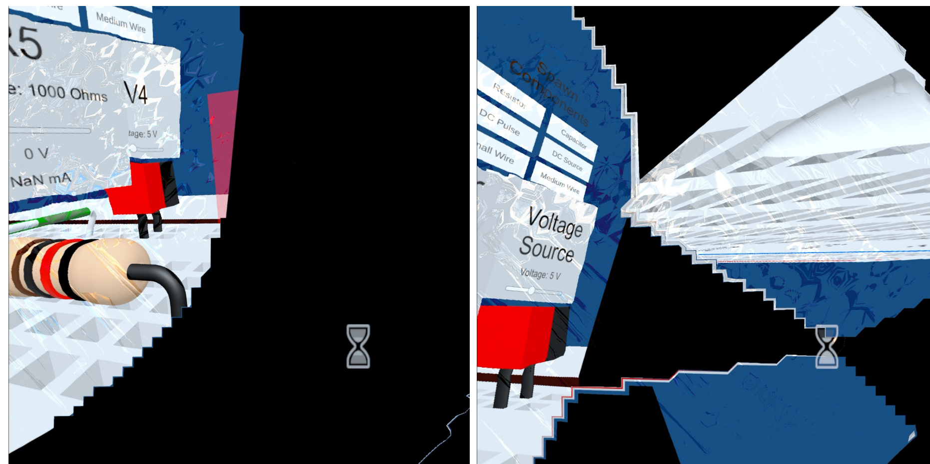

Figure 5.3: *Two examples of missing frames and frame errors caused by delays in Unity frame generation*

5.2 Discussion

This application allows users to have an immersive experience building and testing circuits. While not a direct replacement for hands-on learning, this can act to supplement students' education and further their understanding of these basic concepts. Currently, simple analog circuits and two selected digital IC packages can be built.

Using the simulator is very user-friendly. A user will start the application from the Quest 2 menu and will be loaded into a default scene with a voltage divider on the breadboard. This allows new users to get used to the controls and see the simulator in action. They can adjust the resistance levels using the controller or their hands and observe the changes in current and voltage. Enabling 'Current Mode" or "Voltage Mode" will allow the user to see the advanced visuals of current and voltage. From this initial scene, users can select different premade scenes, like a switch and LED or the premade digital logic circuit. Selecting "new scene" will load an empty breadboard that allows users to select and place any of the available electrical elements in any configuration to build out their own circuits.

Some limitations were experienced in the development of this thesis. As discussed above in 4 , digital logic can not be simulated at the hardware level. The translation layer works to mesh the simulation with the boolean logic needed to simulate these ICs, but new implementation may be needed for more complex ICs like microcontrollers. Another limitation is that this application was only tested on the Meta Quest 2 headset with and without controllers. There are many other VR platforms out on the market, but with this in mind, the VR Circuit Simulator uses Unity's Extended Reality Software Development Kit (XR SDK). This supports any device with an OpenXR runtime, including Meta, Vive, and Microsoft HMDs. So, this application could theoretically be used on devices other than the Quest 2; however, these platforms are out of the scope of this thesis. The source code, along with all media files, are available upon request.

This thesis has produced an application, a video snippet captured from Quest 2 is provided with the link below to a small example of the VR Circuit Simulator's capabilities.

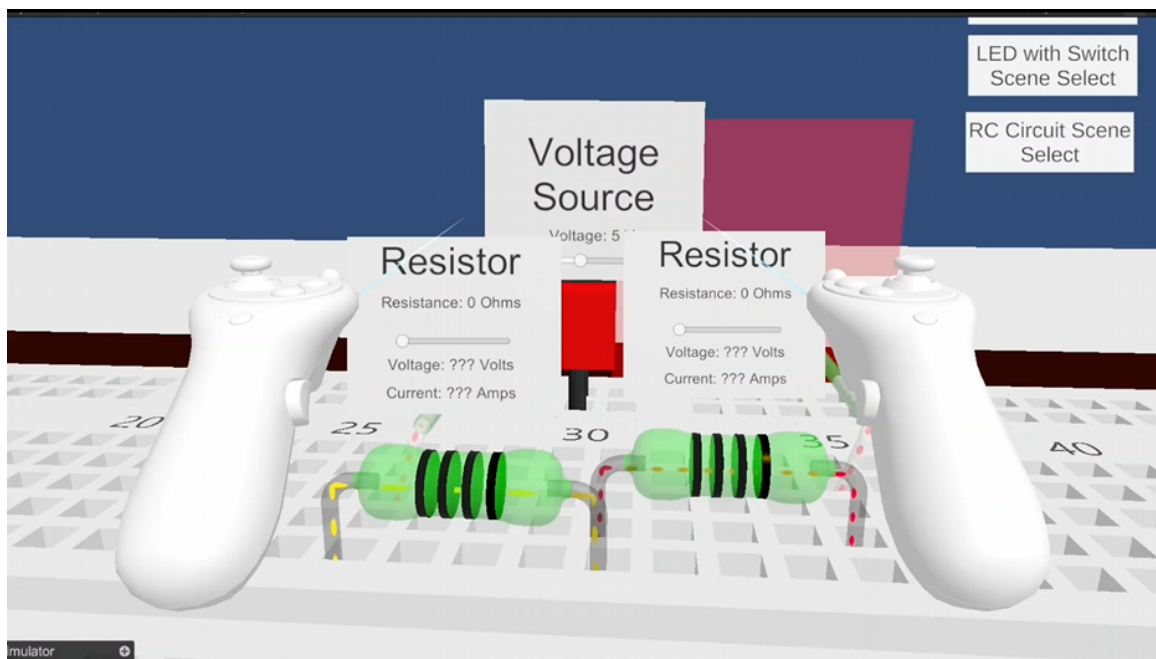

Figure 5.4: *Video of Simulator Functionality (MP4 89.1 MB)*

CHAPTER 6

Conclusion and Future Work

6.0.1 Conclusion

In this thesis, we created a new VR circuit simulator with advanced visualization for fundamental electrical concepts. Developed using the Unity engine and implemented for the Quest 2 headset, the simulator allows users to interact with circuit elements and run various simulations. Different viewing modes were implemented to allow users to see current flow direction, magnitude, and voltage drops displayed as a gradient of the entire circuit. Using Quest 2, users can interact with the simulator with the paired controllers or through tracked hand motions.

Through the process of completing this thesis, many parts needed to fit together. All of the scenes and GameObjects needed to be created in Unity; some of the mesh renders were modified from publicly available 3d models. Prefabs were created from these GameObjects so that multiple instances could be used to create different circuit topologies. All of the prefabs included custom scripts that controlled different aspects, such as managing contact points between the prefabs and the breadboard, storing and updating the prefab's internal variables like voltage over, current through, etc. All of the electrical elements and UIs were made to be able to be interfaced by controllers and hands-free control schemes. Multiple example scenes were created to showcase the capabilities of the application; these include a

basic voltage divider circuit, a resistor ladder circuit, a resistor-capacitor circuit, and a digital logic IC circuit. Custom shaders and scripts were created to visualize current and voltage. A notable challenge was interfacing the electrical elements with the Unity physics engine and using the connection points to create netlists that the Spice# simulator could use to simulate the nodal voltages of the circuits. For example, if a resistor's legs had colliders that were not correctly sized when placing the resistor into the breadboard, the physic engine would send said resistor flying. All collider and Ridgebody components were tuned to minimize these physics engine glitches. When building out simulation functionality, the impacts of the simulation and frame generation were investigated, and optimized scripts were created to enable the simulation of digital logic without impacting the user's experience. To the best of our knowledge, there is no similar system available to our students.

6.0.2 Future Work

Future work on this research would expand the visualization to digital logic, allowing students to "see the gates inside the integrated circuits package." This would help students connect the symbolic representation of the digital gate to the physical packages that we use in real life. Also, additional electronic parts can be added to the simulator to create a larger, encompassing library that would allow users to simulate and explore more circuits.

Bibliography

[1] B. Xie, H. Liu, R. Alghofaili, *et al.*, "A review on virtual reality skill training applications," *Frontiers in Virtual Reality*, vol. 2, 2021, ISSN: 2673-4192. DOI: 10.3389/frvir.2021.645153. [Online]. Available: https://www.frontiersin.org/articles/10.3389/frvir.2021.645153.

[2] Unity Technologies, *Unity*, https://unity.com/, 2005.

[3] E. Robledo, *KVL and KCL*. Sep. 2017. [Online]. Available: https://www.autodesk.com/products/fusion-360/blog/kirchhoffs-law-for-complex-circuits/.

[4] S. Boulanger, M. Golebiowski, katzb123, and O. Sodalom, *Spicesharp*, version 3.1.5. [Online]. Available: https://github.com/SpiceSharp/SpiceSharp.

[5] [Online]. Available: https://ptolemy.berkeley.edu/projects/embedded/pubs/downloads/spice/.

[6] [Online]. Available: https://www.meta.com/quest/products/quest-2.

[7] Z. Lai, Y. Cui, T. Zhao, and Q. Wu, "Design of three-dimensional virtual simulation experiment platform for integrated circuit course," *Electronics*, vol. 11, no. 9, 2022, ISSN: 2079-9292. DOI: 10.3390/electronics11091437. [Online]. Available: https://www.mdpi.com/2079-9292/11/9/1437.

[8] Q. Cao, B. T. Png, Y. Cai, Y. Cen, and D. Xu, "Interactive virtual reality game for online learning of science subject in primary schools," in *2021 IEEE International Conference on Engineering, Technology Education (TALE)*, 2021, pp. 383–389. DOI: 10.1109/TALE52509.2021.9678916.

[9] T. Jiang and Z. Zhuang, "A multi-person collaborative simulation system for circuit experiment system base on virtual reality," in *2021 International Conference on Intelligent Transportation, Big Data Smart City (ICITBS)*, Mar. 2021, pp. 253–259. DOI: 10.1109/ICITBS53129.2021.00071.

[10] M. Khairudin, A. Triatmaja, W. Istanto, and M. Azman, "Mobile virtual reality to develop a virtual laboratorium for the subject of digital engineering," 2019.

[11] Y. Li, Y. Shen, C. Sukenik, B. Sanders, P. Delacruz, and J. Mason, "Work-in-progress: Rapid development of advanced virtual labs for in-person and online education," *2022 ASEE Annual Conference amp; Exposition Proceedings*, 2022. DOI: 10.18260/1-2--40667.

[12] P. Lucas, D. Vaca, F. Domínguez, and X. Ochoa, "Virtual circuits: An augmented reality circuit simulator for engineering students," in *2018 IEEE 18th International Conference on Advanced Learning Technologies (ICALT)*, 2018, pp. 380–384. DOI: 10.1109/ICALT.2018.00097.

[13] P. Zamojski, N. Barczyk, M. Frankowski, *et al.*, "Ohm vr: Solving electronics escape room challenges on the roadmap towards gamified steam education," in *2023 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, 2023, pp. 532–535. DOI: 10.1109/VRW58643.2023.00117.

[14] M. Bradley, *Red led 5mm*, Aug. 2012. [Online]. Available: https://grabcad.com/library/red-led-5mm.

[15] M. .-. WORKSHOP, *Breadboard*, Oct. 2020. [Online]. Available: https://grabcad.com/library/breadboard-15.

[16] Z. Salamacha, *Cd4021b-mil - cmos 8-stage static shift register*. [Online]. Available: https://www.3dcontentcentral.com/download-model.aspx?catalogid=171&id=685535.

[17] P. Šafář, *Tht resistors 1/4 to 5 w (vertical)*, Jun. 2021. [Online]. Available: https://grabcad.com/library/tht-resistors-1-4-to-5-w-vertical-1/details?folder_id=10517080.

APPENDIX A

Code

```
using System;
using System.Collections.Generic;
using Unity.VisualScripting;
using UnityEngine;
using UnityEngine.UI;


public class Resistor_info : MonoBehaviour
{

    [SerializeField]
        public double resistance = 0;
        public double voltage_over = 0;
        public double current_through = 0;
        public string connection_A = "none";
        public string connection_B = "none";
        public Text txt;
        public Text Voltage_txt;
        public Text Current_txt;
        public Material material;
        public LineRenderer lineRenderer;

    void Start()
    {
        // Get the LineRenderer component attached to this
    GameObject
        lineRenderer = GetComponent<LineRenderer>();

        // Get the current material of the LineRenderer
        material = lineRenderer.material;
    }

   public void resistor_info_print()
    {
        Debug.Log("Resistance: " + resistance);
        Debug.Log("Voltage Drop: " + voltage_over);
        Debug.Log("Current Through: " + current_through);
        Debug.Log("Connection A attached to " + connection_A);
```

```
38          Debug.Log("Connection B attached to " + connection_B);
39      }
40
41      public List<string> resistor_info()
42      {
43          var resistor_info_list = new List<string>
44          {
45              resistance.ToString("0.00e0"),
46              voltage_over.ToString(),
47              current_through.ToString(),
48              connection_A,
49              connection_B
50          };
51
52          return resistor_info_list;
53      }
54
55      public void resistor_update()
56      {
57          try
58          {
59              double a = GameObject.Find(connection_A).GetComponent<
    Wire_info>().voltage_node;
60              double b = GameObject.Find(connection_B).GetComponent<
    Wire_info>().voltage_node;
61
62              voltage_over = Math.Abs(b - a);
63              current_through = Math.Round(voltage_over / resistance *
     1000,3);
64
65              if(a < b)
66              {
67                  current_through = -current_through;
68              }
69              if (gameObject.name.Contains("R"))
70              {
71                  Voltage_txt.text = voltage_over + " V";
72                  Current_txt.text = current_through + " mA";
73              }
74
75              var current = new Vector4((float)current_through,0.0f
    ,0.0f,0.0f);
76
77              material.SetVector("_Current",current);
78
79          }
80          catch(NullReferenceException)
81          {
82              Debug.Log("Circuit not complete, Try Again");
83          }
84      }
85
```

```
86 }
```

Listing A.1: *Resistor_info*

```
using System.Collections.Generic;
using UnityEngine;
using System;
using SpiceSharp;
using SpiceSharp.Components;
using SpiceSharp.Simulations;
using SpiceSharp.Simulations.IntegrationMethods;


public class Create_net_list : MonoBehaviour
{
    Resistor_info resistor;
    Voltage_info voltage;
    Capacitor_info capacitor;
    LED_info LED;
    Switch_info sw;
    IC_74ls08 quad_and_gate;

    IC_74ls283 four_bit_adder;
    GameObject g;
    private Circuit OR;
    private SubcircuitDefinition OR_GATE;
    private Circuit AND;
    private SubcircuitDefinition AND_GATE;
    private SubcircuitDefinition NAND_gate;
    private SubcircuitDefinition XOR_GATE;
    private SubcircuitDefinition Full_Adder;
    private SubcircuitDefinition ic_74ls283;
    private SubcircuitDefinition ic_74ls08;
    private SubcircuitDefinition ic_74ls00;
    public string NetList;

    private string GND;

    private Circuit ckt = new Circuit();

    public void Creating_net_list()
    {
        // Stopwatch stwh = new Stopwatch();

        // stwh.Start();




        ckt.Clear();
        NetList = "Circuit Simulation\n";
        GameObject[] all_components = GameObject.
    FindGameObjectsWithTag("Component");
        GameObject[] all_nodes = GameObject.FindGameObjectsWithTag("
    Wire_connected");
        List<string> info;
        int index = 0;
        int sim_mode = 0;
```

```
53          var dModel = new DiodeModel("LED")
54          .SetParameter("is", 93.2e-12)
55          .SetParameter("rs", .042)
56          .SetParameter("n", 3.73)
57          .SetParameter("bv", 0.0);
58          var dModel_flag = false;
59          Simulation sim = null;
60          index = 1;
61          Dictionary<string,RealPropertyExport> ExportCurrents = new
    Dictionary<string,RealPropertyExport>();
62          Dictionary<string,double> Currents = new Dictionary<string,
    double>();
63
64          foreach (GameObject n in all_nodes)
65          {
66              n.GetComponent<Wire_info>().clear_time_voltage();
67          }
68
69          foreach (GameObject go in all_components)
70          {
71              if (go.name.Contains("74ls283"))
72              {
73                  go.name = "74ls283_" + index.ToString();
74                  index = index + 1;
75              }
76              else if (go.name.Contains("74ls08"))
77              {
78                  go.name = "74ls08_" + index.ToString();
79                  index = index + 1;
80              }
81              else if (go.name.Contains("SW"))
82              {
83                  go.name = "SW" + index.ToString();
84                  index = index + 1;
85                  sw = go.GetComponent<Switch_info>();
86                  info = sw.switch_info();
87
88                  if (info[1] != "None" && info[2] != "None")
89                  {
90                      var sw1 = new Resistor(go.name, info[1], info
    [2], 1e-10 + (Convert.ToDouble(info[0]) * 1e10));
91                      ckt.Add(sw1);
92                  }
93              }
94              else if (go.name.Contains("LED"))
95              {
96                  go.name = "LED" + index.ToString();
97                  index = index + 1;
98                  LED = go.GetComponent<LED_info>();
99                  info = LED.led_info();
100
101                 if (info[1] != "None" && info[2] != "None")
102                 {
103                     var led = new Diode(go.name, info[1], info[2], "
```

```
    LED");
104                     ckt.Add(led);
105                     if (dModel_flag == false)
106                     {
107                         ckt.Add(dModel);
108                         dModel_flag = true;
109                     }
110                 }
111             }
112             else if (go.name.Contains("R"))
113             {
114                 go.name = "R" + index.ToString();
115                 index = index + 1;
116                 go.GetComponent<Resistor_info>().txt.text = go.name;
117                 resistor = go.GetComponent<Resistor_info>();
118                 info = resistor.resistor_info();
119 
120                 if (info[3] != "None" && info[4] != "None")
121                 {
122                     var res = new Resistor(go.name, info[3], info
    [4], Convert.ToDouble(info[0]));
123                     ckt.Add(res);
124                     var currentExport = new RealPropertyExport(sim,
    res.Name,"i");
125                     ExportCurrents[res.Name] = currentExport;
126                     NetList += go.name + " " + info[3] + " " + info
    [4] + " " + info[0] + "\n";
127                 }
128             }
129             else if (go.name.Contains("W"))
130             {
131                 go.name = "W" + index.ToString();
132                 index = index + 1;
133                 resistor = go.GetComponent<Resistor_info>();
134                 info = resistor.resistor_info();
135 
136                 if (info[3] != "None" && info[4] != "None")
137                 {
138                     var res = new Resistor(go.name, info[3], info
    [4], Convert.ToDouble(info[0]));
139                     ckt.Add(res);
140                     NetList += go.name + " " + info[3] + " " + info
    [4] + " " + info[0] + "\n";
141                 }
142             }
143             else if (go.name.Contains("pulse"))
144             {
145                 go.name = "pulse" + index.ToString();
146                 index = index + 1;
147                 go.GetComponent<Voltage_info>().txt.text = go.name;
148                 voltage = go.GetComponent<Voltage_info>();
149                 info = voltage.voltage_info();
150 
151                 if (info[1] != "None" && info[2] != "None")
```

```
152                 {
153                     var voltage = new VoltageSource(go.name, info
    [1], info[2], new Pulse(0.0, Convert.ToDouble(info[0]), 0.01, 1e
    -3, 1e-3, 0.02, 0.04));
154                     sim = new Transient("Tran 1", new FixedEuler {
    Step = 1e-3, StopTime = 1 });
155                     sim_mode = 2;
156                     ckt.Add(voltage);
157                     NetList += go.name + " " + info[1] + " " + info
    [2] + " " + info[0] + "\n";
158                     print("Transient mode active");
159                 }
160             }
161             else if (go.name.Contains("V"))
162             {
163                 go.name = "V" + index.ToString();
164                 index = index + 1;
165                 go.GetComponent<Voltage_info>().txt.text = go.name;
166                 voltage = go.GetComponent<Voltage_info>();
167                 info = voltage.voltage_info();
168                 GND = info[2];
169                 if (info[1] != "None" && info[2] != "None")
170                 {
171                     var voltage = new VoltageSource(go.name, info
    [1], info[2], Convert.ToDouble(info[0]));
172                     sim = new DC("dc", go.name, Convert.ToDouble(
    info[0]), Convert.ToDouble(info[0]), 0.001);
173                     sim_mode = 1;
174                     ckt.Add(voltage);
175                     NetList += go.name + " " + info[1] + " " + info
    [2] + " " + info[0] + "\n";
176                 }
177             }
178             else if (go.name.Contains("Cap"))
179             {
180                 go.name = "Cap" + index.ToString();
181                 index = index + 1;
182                 go.GetComponent<Capacitor_info>().txt.text = go.name
    ;
183                 capacitor = go.GetComponent<Capacitor_info>();
184                 info = capacitor.capacitor_info();
185 
186                 if (info[2] != "None" && info[3] != "None")
187                 {
188                     var res = new Capacitor(go.name, info[2], info
    [3], Convert.ToDouble(info[0]));
189                     ckt.Add(res);
190                     NetList += go.name + " " + info[2] + " " + info
    [3] + " " + info[0] + "\n";
191                 }
192             }
193         }
194         //print(NetList);
195 
```

```
196         sim.ExportSimulationData += (sender, exportDataEventArgs) =>
197         {
198             foreach (GameObject go in all_nodes)
199             {
200                 //print("Node " + go.name + ": " +
    exportDataEventArgs.GetVoltage(go.name) + "V");
201 
202                 try
203                 {
204                     GameObject.Find(go.name).GetComponent<Wire_info
    >().voltage_node = Math.Round(exportDataEventArgs.GetVoltage(go.
    name), 3);
205                 }
206                 catch (System.Exception)
207                 {
208 
209                 }
210 
211                 if (sim_mode == 2)
212                 {
213 
214                     GameObject.Find(go.name).GetComponent<Wire_info
    >().add_time_voltage(exportDataEventArgs.GetVoltage(go.name));
215                 }
216 
217             }
218 
219             foreach( var current in ExportCurrents)
220             {
221                 Currents[current.Key] = current.Value.Value;
222                 //UnityEngine.Debug.Log(current.Key + " " + current.
    Value.Value);
223             }
224         };
225 
226         // Run the simulation
227         try
228         {
229             //Stopwatch stwh = new Stopwatch();
230 
231             //stwh.Start();
232 
233             sim.Run(ckt);
234 
235             //stwh.Stop();
236            // double ts = stwh.Elapsed.TotalMilliseconds;
237             //UnityEngine.Debug.Log(ts);
238 
239             Digital_IC_Update(all_components,sim);
240 
241             foreach (GameObject go in all_components)
242             {
243                 if (go.name.Contains("R"))
244                 {
```

```
245                     // if(Currents.TryGetValue(go.name,out double
    current))
246                     // {
247                     //     GameObject.Find(go.name).GetComponent<
    Resistor_info>().current_through = current;
248                     //     UnityEngine.Debug.Log(current);
249                     // }
250
251                     GameObject.Find(go.name).GetComponent<
    Resistor_info>().resistor_update();
252                 }
253                 else if(go.name.Contains("W"))
254                 {
255                     GameObject.Find(go.name).GetComponent<
    Resistor_info>().resistor_update();
256                 }
257                 else if (go.name.Contains("Cap"))
258                 {
259                     GameObject.Find(go.name).GetComponent<
    Capacitor_info>().capacitor_update();
260                 }
261                 else if (go.name.Contains("LED"))
262                 {
263                     GameObject.Find(go.name).GetComponent<LED_info
    >().led_update();
264                 }
265             }
266         }
267         catch (Exception e)
268         {
269             UnityEngine.Debug.Log(e);
270         }
271
272
273         if (sim_mode == 2)
274         {
275             var oscope = GameObject.Find("Oscope").GetComponent<
    Osilliscope_screen>();
276             oscope.flag = false;
277
278             var probe = GameObject.Find("Probe 1").transform.
    GetChild(0).gameObject;
279             probe.GetComponent<Probe_voltages>().set_voltages();
280
281             var probe2 = GameObject.Find("Probe 2").transform.
    GetChild(0).gameObject;
282             probe2.GetComponent<Probe_voltages>().set_voltages();
283
284             oscope.flag = true;
285             oscope.pos = 0;
286
287         }
288
289         // stwh.Stop();
```

```
290         //      double ts = stwh.Elapsed.TotalMilliseconds;
291         //      UnityEngine.Debug.Log(ts);
292     }
293 
294     private void Digital_IC_Update(GameObject[] all_components,
    Simulation sim)
295     {
296         //Dictionary<string,double> outputs = new Dictionary<string,
     double>();
297         Circuit temp = new Circuit(ckt.Clone());
298 
299         foreach (GameObject go in all_components)
300         {
301             if (go.name.Contains("74ls08"))
302             {
303                Dictionary<string,double> new_output = GameObject.
    Find(go.name).GetComponent<IC_74ls08>().ic_74ls08_update();
304                 //     foreach(var new_values in new_output)
305                 //     {
306                 //      outputs.Add(new_values.Key,new_values.Value);
307                 //     }
308 
309 
310                 foreach (KeyValuePair<string, double> output in
    new_output)
311                 {
312                     //print("VS"+output.Key + " " + output.Value);
313                     var source = new VoltageSource("VS" + output.Key
    , output.Key, GND, output.Value);
314                     temp.Add(source);
315                 }
316                 //print("sim inside ic update");
317                 sim.Run(temp);
318             }
319             else if (go.name.Contains("74ls283"))
320             {
321                 Dictionary<string, double> new_output2 = GameObject.
    Find(go.name).GetComponent<IC_74ls283>().ic_74ls283_update();
322                 //    foreach(var new_values2 in new_output2)
323                 //     {
324                 //      outputs.Add(new_values2.Key,new_values2.Value
    );
325                 //     }
326 
327 
328                 foreach (KeyValuePair<string, double> output in
    new_output2)
329                 {
330                     //print("VS"+output.Key + " " + output.Value);
331                     var source = new VoltageSource("VS" + output.Key
    , output.Key, GND, output.Value);
332                     temp.Add(source);
333                 }
334                 //print("sim inside ic update");
```

```
335                 sim.Run(temp);
336             }
337         }
338         // Circuit temp = new Circuit(ckt.Clone());
339 
340         // foreach(KeyValuePair<string,double> output in outputs)
341         // {
342         //     //print("VS"+output.Key + " " + output.Value);
343         //     var source = new VoltageSource("VS"+output.Key,output
    .Key,GND,output.Value);
344         //     temp.Add(source);
345         // }
346         // print("sim inside ic update");
347         // sim.Run(temp);
348 
349     }
350 }
```

Listing A.2: *Create Net List*