



AUBURN UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

SENIOR DESIGN OPERATION READINESS REPORT

TEAM 3: 3D MODELING OF DETACHED METAL WHISKERS

By

MANAGER: KURT KNUDSEN

SCRIBE: SHAWN EOM

CARSON REAMS

SPENCER HURST

NATHAN NICHOLS

SPONSOR: DONNA HAVRISIK, ERROL REID

TECHNICAL ADVISOR: DR. GEORGE FLOWERS

TABLE OF CONTENTS

LIST OF TABLES	5
LIST OF FIGURES	6
LIST OF ABBREVIATIONS	9
INTRODUCTION	10
DESIGN SPECIFICATIONS	12
Project Constraints.....	12
Objective.....	14
Mission Environment.....	14
BACKGROUND INFORMATION	15
Whisker Formation	17
Failure Modes	17
Simulation and Software.....	18
Unity Game Engine	18
Monte Carlo Simulation	19
Functional Decomposition.....	19
DESIGN CONCEPTS	21
Morphological Matrix.....	22
DESIGN DESCRIPTION	25
Simulation Units	25
Boundary Box / Container.....	26
Whisker ID.....	26
Pad ID	27
Storable Text Files.....	27
Graphical Interface Camera.....	27
Graphical Interface Indicator	28
CCA Interpretation	28
File Processing.....	28
Material Processing	29
Random Whisker Dimensional Distribution.....	29
CHANGES TO 2D SIMULATION.....	31
Change From X-Y Planar View to an X-Z Planar View	31
Simulation Units	33
Boundary Box/Container.....	33

Whisker ID.....	34
Pad ID	36
Storable Text Files.....	38
Output Window.....	40
Graphical Interface Camera.....	41
OPERATION READINESS SECTION	44
Testing	44
Online Beta	45
UI Improvements	46
Better Visual Analysis	49
Simulation Manipulation	52
Critical Pairs/Nodes	56
Output File/Screenshot	57
Excel File for Data Analysis.....	60
Further Development.....	68
CONCLUSION	71
APPENDIX A	73
Programmatic Questions.....	73
APPENDIX B	75
Gantt Chart.....	75
APPENDIX C	76
BETA Website.....	76
APPENDIX D	77
UIScript.cs	77
CallForces.cs.....	101
CameraControl.cs	102
CameraOrbit.cs	104
DataManager.cs	110
DataSaveManager.cs.....	111
DistSelect.cs.....	111
HeatMap.cs	112
MatSelect.cs.....	114
ResetBoard.cs	115
ShockManager.cs.....	115
TriggerControl.cs.....	120
TriggerTracker.cs.....	123
VibrationManager.cs.....	124

WhiskerControl.cs	129
WhiskerData.cs	138
CircuitBoardFinder.cs.....	139
csvReader.cs	142
DropDownHandler.cs	144
InputFieldtoFloat.cs	145
PositionUpdate.cs	146
TabControllerFront.cs.....	146
TabControllerStay.cs	147
WhiskerCubeScaler.cs.....	148
WhiskerSpawnCubeVisibility.cs.....	148
BoardDetector.cs.....	149
WallCreator.cs	151
SaveManager.cs	155
SimulationController.cs.....	156
SimulationData.cs.....	159
RandomFromDistribution.cs.....	159
LIST OF REFERENCES	169

LIST OF TABLES

Table

Table 1: Morphological Matrix.....	23
Table 2: .csv File Names and Their Functions.....	40
Table 3: User Interface Scripts and Their Functions	49

LIST OF FIGURES

Figure

Figure 1: Overall Simulation with all User Inputs.....	16
Figure 2: Functional Decomposition.....	20
Figure 3: Raspberry Pi PCB.....	29
Figure 4: Team 1 2D Simulation with x-y camera axis.....	31
Figure 5: Team 2 2D Simulation with x-z camera axis	32
Figure 6: Metal Whisker Team 1's Simulation Units.....	33
Figure 7: Boundary Wall Additions.....	34
Figure 8: Whisker Hover and Bridge ID Feature.....	35
Figure 9: Whisker ID in Unity Hierarchy	36
Figure 10: Conductor ID in Simulation Hierarchy	37
Figure 11: Conductor ID and Randomization.....	38
Figure 12: Whisker Dimension .csv File	39
Figure 13: Complete Whisker Data .csv File.....	39
Figure 14: CSV and Screenshot Window	41
Figure 15: Initial view of Simulation.....	42
Figure 16: Zoomed in Simulation.....	43
Figure 17: Zoomed In with Board Shifted in Simulation	43
Figure 18: Online Beta.....	46
Figure 19: Team One's Original User Interface	47
Figure 20: Team Two's Revised User Interface	47
Figure 21: Team Two's Revised User Interface with a Tab Open	48
Figure 22: Bridged Whisker Zoomed In (left) and Zoomed Out with Rotation (right).....	50

Figure 23: Close-Up of Momentarily Bridged Whisker	51
Figure 24: Whisker Number Indicator when in Close-Up View of Bridged Whisker(s)	51
Figure 25: Camera Controls and Drop Down to Target Whisker to View	51
Figure 26: Whisker Properties Tab.....	52
Figure 27: Board Control and MCS Tab.....	53
Figure 28: Default Spawn Area (left) and Custom Spawn Area (right)	54
Figure 29: Default Spawn Location (left) and Customer Spawn Location (right)	54
Figure 30: Whiskers in Earth Gravity (left) and Custom 2 Newton Gravity in the Y- Direction.....	55
Figure 31: Default Board (left) and Board with Custom Rotation (right)	55
Figure 32: Simulation with Walls Added	55
Figure 33: Critical Pairs Tab.....	56
Figure 34: Directory and File Name Input Field.....	57
Figure 35: Excel File Name Based on Input Field.....	57
Figure 36: Excel File of Output Data from Simulation	58
Figure 37: In-Simulation Bridged Whisker Output Tab	59
Figure 38: Screenshot Button.....	59
Figure 39: Screenshot File	60
Figure 40: List of All Whiskers in Excel.....	61
Figure 41: List of Bridged Whisker in Excel.....	61
Figure 42: List of Critical Bridged Whiskers in Excel	62
Figure 43: List of Simulation Inputs	62
Figure 44: List of Bridge Probability and Critical Bridge Probability for Each Iteration	64
Figure 45: List of Summary Statistics.....	64
Figure 46: Bridged Pair Analysis and Filters.....	65
Figure 47: Bridged Pair Plot	66

Figure 48: Bridge Count Distribution.....	67
Figure 49: Bridge Count Distribution Plot.....	67

LIST OF ABBREVIATIONS

1D	One-Dimensional
2D	Two-Dimensional
3D	Three-Dimensional
BoM	Bill of Materials
CAD	Computer Aided Design
CCA	Circuit Card Assembly
.csv	Comma Separated Values (file type)
.fbx	Filmbox (file type)
ID	Identification
MDA	Missile Defense Agency
.mtl	Material (file type)
NASA	National Aeronautics and Space Administration
NSIN	National Security Innovation Network
.obj	Object (file type)
PCB	Printed Circuit Board
SI	International System of Units
.step	Standard for Exchange of Product Data (file type)
STIG	Security Technical Implementation Guide
.txt	Text File Document (file type)
UI	User Interface
US	United States
WP	Working Principle

CHAPTER 1 INTRODUCTION

In many areas of today's life machines and robots are filled with all sorts of components and wiring. All these systems require some sort of circuit board to operate and achieve their daily functions. Unfortunately, this has led to them being vulnerable to the phenomenon known as metal whiskers. These microscopic whiskers have plagued many machines and have led to thousands of dollars of damage including almost singlehandedly shutting down NASA's space shuttle programs in the early 2000s. These whiskers are known to be very hard to prevent and almost impossible to predict when and where they might occur. This means that all systems that require some level of assurance whether they oversee human lives or military applications must undergo thorough testing. These tests are to see where these whiskers might prove a problem. This would be imperative due to how a simulation of this magnitude would be beneficial to a failure analyst when a machine breaks because of metal whiskers or help prevent a short caused by metal whiskers all together. This is what this metal whisker project is attempting to accomplish.

The project's purpose is a revision of the previous project on metal whiskers done by Metal Whiskers Team 1 (referred to as Team 1) who worked on this simulation in the previous semester by the current team (Team 2). The previous team had completed both a rudimentary 2D program and started a complex 3D program that models a basic shape representing metal whiskers falling on a PCB. A revision of the code will be done to improve performance of the simulation within the unity program and improve the UI so that the simulation will be more user friendly and accurate. This project aims to identify key components such as units, whiskers, pads, bridges, node pairs, etc. that will be implemented for ease of tracking and readily available information. This project can be applied to many fields or industries as metal whiskers can

appear in almost any system with solder or electrical circuit board components. This project is done in conjunction with members associated with the US Missile Defense Agency and NASA, who take special interest in failure modes of circuitry as it is critical to mission success.

CHAPTER 2 DESIGN SPECIFICATIONS

Starting the design process requires an in-depth understanding of the project's current constraints, an understanding of how to link them to the project's objective, and a comprehensive understanding of the mission environment. This project is a continuation of the work done by Team 1, therefore many of the project constraints will remain the same. Building on the progress the previous team has made; this section outlines the essential factors that must be assessed to deliver a practical and functional solution for simulating metal whiskers under various conditions.

Project Constraints

The constraints for this project are focused on the improvement of the simulation Team 1 has created. These constraints exist to ensure that the simulation produces useful results and is as user-friendly as possible. The constraints were given by the project sponsors Donna Havrisik (MDA) and Errol Reid (NSIN).

- The simulation must be created using unity.
 - The Unity Game Engine must be used for this simulation.
- The program must recognize units used in .step files.
 - When importing a .step file, the program must be able to recognize the units used to ensure that dimensions are correctly transferred. The metal whiskers must be accurately scaled according to the uploaded .step file.
- The program must identify and label node pairs.

- The program must accurately identify and label pairs of exposed surface areas that act as a conductor within the circuit. Identifies node pairs that are at risk of being bridged by a metal whisker.
- The program must create an ID number for each metal whisker generated.
 - An identification number will be created for each metal whisker. Each metal whisker's length, diameter, and resistance must be linked to their associated ID number.
- The simulation must have a graphical interface.
 - The user must have a way to easily identify and locate any bridged nodes. The simulation must provide a visual indication of bridged nodes.
- The program must output desired information.
 - The program must be able to generate a file containing information gathered from the simulation. This includes details such as which whisker caused a bridge, and which nodes were affected.
- The design must consist of one code and one script language.
 - Users will need to be able to refine the code should they decide to make changes.
- The metal whiskers are detached and airborne.
 - Whiskers are assumed to not be attached to or grow from any components. They will be dropped from a location that requires them to be airborne.
- Users must be able to import created or accessed CCA models.
 - The program must allow for the importation of user created CCA models.

Objective

The main objective of this project is to refine and improve upon the metal whisker simulation created by Team 1. This simulation runs a detached whisker simulation using a user-imported CCA CAD model. The simulation accepts specific user inputs such as whisker characteristics, dropping location, boundary conditions, and various environmental conditions. The simulation outputs relevant information to the user, including whether a bridge was created, and evaluates the risk probability based on the user's inputs. The simulation will be capable of importing .step files and identifying important information such as the units and materials used.

Mission Environment

This project's mission environment is flexible and completely dependent on the user. The simulation will be able to adapt to many different environments based on the inputs the user gives. The mission environments were given by the project sponsors. A still and stable environment such as computers, cameras, televisions, and more can be affected metal whisker causing bridges. Environments with high vibrations and impacts such as missiles, rockets, rough terrain vehicles, and G-Forces are environments in which metal whiskers can move and cause issues. Low orbit satellites that experience lower gravity or radiation may have use for the 3D simulation. Climates with fluctuating temperatures can cause varied whisker growth whether that's due to the sun, space, hot machinery, or more. Lastly, rovers or other objects on different planets with different gravity or floating through space with little to no gravity may have use for the 3D simulation. All of which are environments in which the 3D simulation will be effective and helpful for predicting and recording the potential for metal whiskers to be an issue. The simulation will account for many of these factors to help better understand and protect objects affected by metal whiskers in a wide variety of conditions and environments.

CHAPTER 3

BACKGROUND INFORMATION

The lognormal and normal distribution is used to design and model the metal whiskers. The simulation needs a method to emulate environments that are encountered in the real-world and therefore a distribution method for the simulation like lognormal is required. The Mu (μ) and Sigma (σ) user inputs are used for both the length and diameter of the whisker cylinders. However, due to the distribution type selected, lognormal distributions require a location (μ) and scale (σ) parameter to model whiskers. The whisker dimension input goes hand in hand with the normal distribution because the number of whiskers directly correlates with the distribution of whiskers to get the desired amount. The gravity user input allows for the user to pick the desired force of gravity from different planets gravities, this influences how the whiskers fall in the simulation. The whisker material user input gives the user the selection between tin, zinc, or cadmium whiskers. Material input will allow for the user to adjust whisker mechanics in the simulation due to the change of the materials coefficient of friction, density, and resistivity. The coordinate drop user input allows for the user to determine where they would like the whiskers to drop from in the X, Y, and Z coordinate system in millimeters (mm). The external forces user input allows for the user to add certain external forces for vibration and shock to the whisker environment as needed. Figure 1 shows the UI and how the user might change the actual user inputs for every simulation.

As stated in the introduction, this project is a revision of the previous project done by Team 1. The simulation helps predict certain areas where metal whiskers would damage circuit boards by short circuiting the electronics. The simulation is created on a software called Unity Game Engine. In Unity the user can import an object (.obj) file of their CAD model into the

game window. This object (.obj) file of the PCB comes from the Altium PCB Designer and Blender. After this Unity will automatically create individual Game Objects for necessary parts of the design. Since accuracy is crucial in developing an accurate simulation, the user must also identify the volumetric constraints that bound the whiskers. Multiple user inputs have been added to make the program more applicable to different fields and areas it can be used in. The user inputs include Lognormal and Normal Distribution, Mu and Sigma for Whisker Dimensions, Number of Whiskers, Gravity, Whisker Material, Coordinate Drop Location, and External Forces.

Scale: 10 Units = 1 mm
0.01 Units = 1 micron

Length Mu [mm]	Width Mu [mm]	# of Conductors	Select Gravity
<input data-bbox="456 993 591 1014" type="text" value="Enter text..."/>	<input data-bbox="613 993 748 1014" type="text" value="Enter text..."/>	<input data-bbox="771 993 906 1014" type="text" value="Enter text..."/>	<input data-bbox="922 993 1052 1014" type="text" value="Pick One"/>
Length Sigma [mm]	Width Sigma [mm]	# of Whiskers	# of Iterations
<input data-bbox="456 1035 591 1056" type="text" value="Enter text..."/>	<input data-bbox="613 1035 748 1056" type="text" value="Enter text..."/>	<input data-bbox="771 1035 906 1056" type="text" value="Enter text..."/>	<input data-bbox="922 1035 1052 1056" type="text" value="Enter text..."/>

Total Connections:
0
Odds of Bridging:

Figure 1: Overall Simulation with all User Inputs

Another important aspect of the Unity Program are C# functions. The C# functions are written in the code that can affect or add tools to the simulation. Some of these functions include Make Conductors, Conductor Control, Distribution Select, Length Distribution Generate, Diameter Distribution Generate, Material Selection, Make Whiskers, Coordinate Location, Gravity Control, External Force Generation, Whisker Control, Dimension Storage, and Total

Restart. Within Unity the C# functions work as scripts attached to Game Objects [2] such as whiskers, conductors, and the ground. There are two main parts of each C# function: the start function and the update function. The start function runs at the start of the program, or the model being spawned, and the update function runs each frame of the simulation.

Whisker Formation

The formation of metal whiskers is still not fully understood, while there are many theories about whiskers growing as a response to a mechanism of stress relief, there are also theories about whiskers growth being attributable to recrystallization and abnormal grain growth processes which affects the grain structure which may or may not be affected by residual stress [3]. Those advocating for stress relief point towards residual stress within the tin plating, intermetallic formation, externally applied compressive stresses, bending, or stretching, scratched or nicks, and coefficient of thermal expansion mismatches. Other factors which may influence whisker growth is plating chemistry, plating process, deposit characteristics, substrate, and environment which all in general increase stress or promote diffusion within the deposit leading to greater whisker propensity [3].

Failure Modes

Metal whiskers can grow on electronics and cause connections and bridges between nodes.[4] There are 4 conditions which are possible with metal whiskers: Debris, stable short circuits in low voltage, transient short circuits, and metal vapor arc [3]. Debris are loose non-bridging whiskers that do not affect the circuit board. The stable short circuit in low voltage is when a whisker makes a bridge but does not affect either connection due to there not being enough voltage to make the whisker conductive. The transient short circuit is when the whisker

bridges and there is enough voltage to make the whisker conductive but there is not enough voltage to melt the metal whisker, creating electrical shorts. The metal vapor arc is when the whisker bridges and the voltage is high enough to melt the whisker and creates a vapor arc damaging and possibly destroying the circuit board. [3]

Simulation and Software

The simulation is run on the Unity Game Engine which uses C# scripts that are written within Visual Studios to assist the game engine in creating this simulation. After the user puts in their inputs and runs the simulation, an Excel file will be made which contains the information about the simulation within it. Then upon using the information from the Excel file you can choose from a variety of output graphs that will show you graphical results of the simulation. The addition of an output window allows the user to view the results of the simulation before having to download the csv file.

Unity Game Engine

The Unity Game Engine combines the simulation designed, the user inputs, and the C# code to create an accurate and detailed simulation. C# is the chosen coding language due to Unity supporting the language natively [4]. The simulation is pre-liminary designed around the possible inputs and outputs the user would possibly desire. The user then inputs their desired simulation, followed by the C# code and Unity Game Engine working together to produce a simulation based on design, inputs, and outputs that the user would desire from the simulation. Upon finishing the simulation from the user's inputs, Unity would output data into the Output Window the user desired and into an excel file.

Monte Carlo Simulation

Monte Carlo Simulation is used when a process involves many variables, and an outcome cannot be determined easily. Engineers often use this simulation to assess the reliability of complex systems and predict potential outcomes by running many different iterations with various inputs [6]. In this project it is used to randomly select the locations in which each metal whisker is dropped onto the CCA model. Since it is still uncertain how and why metal whiskers are formed, this simulation is crucial to accurately assess the risks metal whiskers pose to our customer's electronics.

Functional Decomposition

The functional decomposition is an analysis that breaks down an overall goal of an engineering problem into sub-functions. In doing so, engineers can break down the overall problem into smaller parts that require attention that are all pertinent to the completion of the project. The functional decomposition created for this project is shown in Figure 2.

The overarching project is based on finding whisker bridging probabilities. The team decided based on a list of objectives given by the customers that doing so would entail the actual simulation of the whiskers as well as a gathering of output data. Important characteristics of the simulation were decided to be the simulation units, the boundary box or container, and the identification of node pairs. The identification of node pairs, whisker ID, and Pad ID information was necessary to gather.

For gathering output data, the need for a storable text file, and the implementation of a graphical interface was deemed necessary. This would enable critical analysis of the simulation. To add a graphical interface, it was decided that an implementation of a camera and bridging indicator would be beneficial to be able to perform a visual analysis.

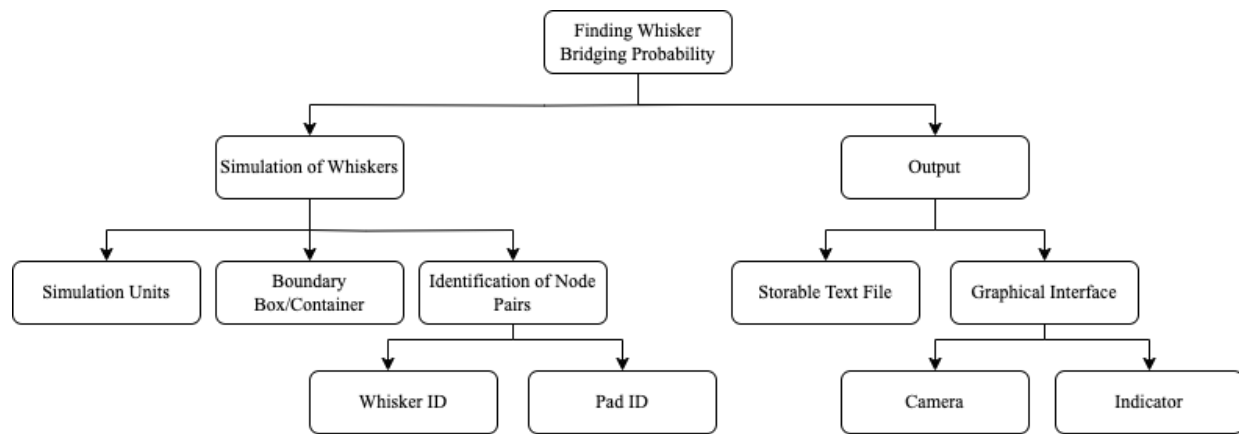


Figure 2: Functional Decomposition

CHAPTER 4 DESIGN CONCEPTS

In this section, concepts related to the functionality and visuality of the simulation will be explored. For this semester of the project, the focus was on revision of the current simulation as well as additional implementation of “ease-of-use” functions in addition to a more in-depth interface for the output of the simulation and properly recording simulation outputs.

There are three main areas for revision of the current product. The first is a revision of the system of units. In Unity, there is no standard metric of measurement, rather a measurement based solely on pixels. As a result, there will need to be a way to convert between real life units of measurement such as micrometers (μm) or millimeters (mm) to a standard for the simulation to be able to interpret.

The second area of revision is the CCA boundary box or container. In the current simulation of the 2D and the 3D simulations, there is no implementation of a physical border which the whiskers must remain throughout the simulation, as a result, a fraction of whisker/node connections may be unaccounted for. The possibilities for the boundary box include an on and off situation, invisible boundary box, or an outlined boundary box; the outlined boundary box was most realistic as boundary boxes wouldn't be coming in and out of existence within reality and an invisible boundary box doesn't also exist within reality. In implementing this feature, the simulation will be able to simulate a physical CCA more accurately within its working environment.

The third area of concern for revisions is identification of connections between nodes and whiskers. While this may not be an important factor yet, in the second module for this overall project, the MDA and NASA want to be able to identify the geometry of the whisker and which location on the CCA bridged, which will require a numerical or alphanumeric tag that will be

able to recall a specific whisker within a simulation. Spawn order was chosen as the best way to identify the whiskers in the simulation. This was identified because it is the easiest way for the unity team to implement a whisker ID and it allowed for a very easy understanding for the user in terms of why whiskers are labeled. Both sized-based and problematic whiskers are useful labeling, however, sized-based ID was a very complicated method of ID in terms of coding and the expertise that the unity team had. Also, problematic whiskers are planned to be identified by another method so there was no reason to identify the whiskers that way.

For implementation of further functions, the team decided that pursuing an exportable table of results would be beneficial for a further analysis of the results, in line with the overall scope of the simulation. Some initial concepts decided on for implementation were an option for exporting results to visually inspect the CCA and bridged nodes after the simulation.

Morphological Matrix

The morphological matrix is a method of taking subfunctions from the functional decomposition and creating working principles for each subfunction. This entails identifying the main categories of what would make each subfunction work, and options on how to accomplish each subfunction. The morphological matrix can be seen in Table 1.

Table 1: Morphological Matrix

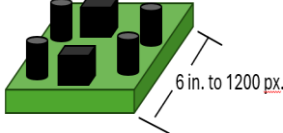
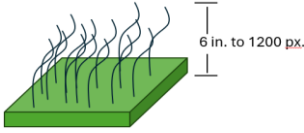
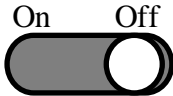
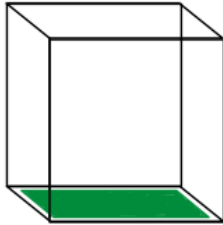
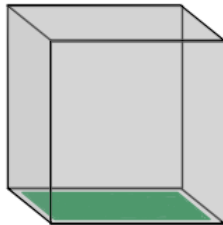


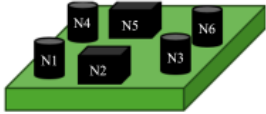

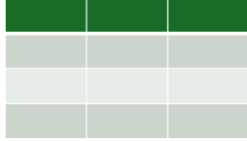
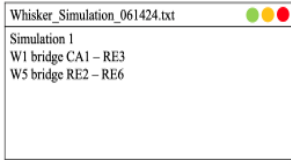
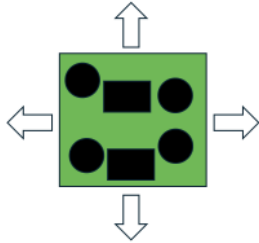
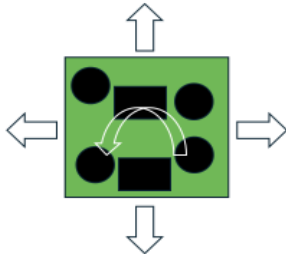
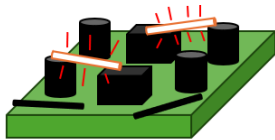
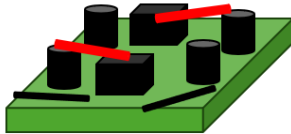
Subfunction	WP1	WP2	WP3
Simulation Units	CCA Based Units of the whisker will be based/scaled on the size of the circuit card assembly. 	Whisker Based The units of the circuit card assembly will be based on the desired whisker size. 	Simulation Standard A standard unit conversion of pixels to millimeters/micrometers will be used to spawn both whiskers and the circuit card assembly <div style="border: 1px solid black; padding: 5px; width: fit-content; margin: 10px auto;">1 cm = 100 px</div>
Boundary Box/Container	Toggle On/Off The boundary box will be able to be toggled on or off by the user for full visibility of the container or the whiskers and circuit card assembly 	Outline The boundary box will be denoted by an outline of its geometry 	Transparent The boundary box will be semi-visible for the entirety of the simulation 
Whisker ID	Size Based Whiskers will be named in order of size 	Order of Spawn-In Whiskers will be named in the order of which they were generated 	Problematic Whiskers Only ID Whiskers that are identified to induce a short.
Pad ID	Numerical The pad ID will be strictly numerical based on its locations in the circuit card assembly 	Alphanumerical The pad ID will include numerical values based on its location, as well as a few letters to denote its type 	N/A

Table 1: Morphological Matrix (continued)

Subfunction	WP1	WP2	WP3
Storable Text File	<p>.csv File The output will be able to be saved as a .csv file</p> 	<p>.txt File The output will be able to be saved as a .txt file</p> 	N/A
Graphical Interface Camera	<p>2D with 2 Axis Pan In-game camera will zoom into chosen connection from a top-down view with freedom in the x-y plane</p> 	<p>3D with 3 Axis Pan In-game camera will zoom into chosen connection from a top-down view with freedom to pan in x-y-z axes</p> 	N/A
Graphical Interface Indicator	<p>Blinking Light The connection will be indicated by a blinking light in the shape of the bridging whisker</p> 	<p>Color Change The bridged whisker will change colors to indicate a connection</p> 	N/A

CHAPTER 5

DESIGN DESCRIPTION

Implementing the right working principles from the proposed morphological matrix impacts the usefulness and outputs of the project. Bad selection in the design can lead to more issues such as wasting time and energy. The design description for the project is split into two major categories, as described in the morphological matrix. The project's two categories include the simulation of the metal whiskers, and the outputs implemented from them. The simulation category can be broken up into smaller categories such as the units, the boundary box, the pad IDs, and the whisker IDs of the simulation. The outputs are broken up into categories such as storable text files, graphical interface cameras, and graphical interface indicators. These seven subcategories describe the parameters and goals the project plans to achieve, and there are many available solutions to each as described in the morphological matrix. The best designs for this project are detailed further based on each subcategory.

Simulation Units

Units measure and describe the physical properties of objects or various items. There are two distinct unit systems: English, and SI. These various unit systems will be implemented through pixel measurements within Unity and the Altium software. Each CCA has a distinct number of pixels associated with it when implemented into Unity. These pixels can be used as a scaling measurement for English or SI units. For this project, the focus will be on the SI system, with a focus on millimeters and microns. However, English units can be added using a different pixel scaling factor and could be added later after the SI units are implemented. This way of implementing units allows for ease of use, and simplicity for how each unit system functions and

is not dependent on the simulation or spawned object size. This standardization of simulation units makes it a faster, more efficient, and consistent solution compared to the alternative ideas.

Boundary Box / Container

The boundary box or container is a physical and visual outline around the CCA to indicate the space in which metal whiskers are contained. The purpose of this box is to contain all the whiskers within the simulation so that they land on the CCA and resemble real world scenarios in which the CCA is encased in some form of housing for its protection. The chosen solution is WP 2 (outline) as seen under boundary box in Table 1. This will be a see-through visual outline of the box while still maintaining the physical functions of a normal box. This will allow the user to easily see the borders of the box while also seeing the CCA and metal whiskers and ensuring they stay contained with an unobstructed view of all simulation components.

Whisker ID

Whisker ID is important for the distinction of whiskers to gain information such as what whiskers caused bridges, shorts, or had no impact. It can be used to help distinguish information about each whisker such as length, width, etc. that will clarify why certain whiskers caused what issues. The best-proposed solution to ID the whiskers is to assign a numeric value to each whisker in the order in which they spawn. After running the simulation, if a whisker is bridging a connection, then the whisker's number will be displayed, so smaller whiskers would still be visible when making a bridge. After the simulation is finished running the user can hover their mouse over each whisker and the whisker number ID will appear, this can be used as a clarity check on if the whisker is numbered properly.

Pad ID

Pad ID is important for similar reasons to whisker ID. However, the pad ID does not need to be randomized, and it is beneficial to have a concise way of identifying the pads. WP 2 (alphanumeric), as seen in Table 1, is the selected design because it gives the best description of each pad and various component types for identification. This design version for the pad ID will also allow for any CCA to be implemented and gain pad IDs. This makes it future proof for how the code will function and work for various CCAs.

Storable Text Files

A storable text file is a file format with different visuals and outputs. For this project the two options are a text (.txt) file, which will output only text, or a .csv file, which will output information in the form of a table. The .csv file is better for displayable information. For this project it means that various information such as whisker ID, pad ID, lengths, widths, bridges, and various information will be easily displayed in a table format. This will be more visually appealing to the user, but also allow for easy organization of information to help gather data from the simulation outputs.

Graphical Interface Camera

A graphical interface camera is a camera that can move around and view the simulation. It will be added to allow for the user to easily view and find any identified parts such as whiskers and pad IDs, or various bridges that the metal whiskers form. The 3D with 3-axis pan WP will allow for a more in-depth view of the simulation and outputs by allowing the user to shift their view in 3 dimensions rather than only 2 dimensions. The graphical interface camera will allow for an improved visual analysis from more angles.

Graphical Interface Indicator

The graphical interface indicator is a way to visually represent the outcomes of the simulation. This will be in the form of what whiskers connect or bridge to what node pairs. When a whisker causes a bridge between node pairs or pad IDs, the whisker will visually change to indicate to the user that a bridge has been made. Having the whiskers change color rather than blink allows for a more noticeable change or connection. It is also easier on the eyes than a blinking indicator would be. Changing the whisker color in Unity is also easier to implement as the hex code for the whisker can be adjusted when the simulation notices any bridging.

CCA Interpretation

When using Unity, various CCAs will be implemented into the simulation by the user. As stated by Team 1, the user will be able to import their CAD model and then will need to register what parts have electrical components. This means that the CCA CAD models imported into Unity will need to be processed before translating into the simulation discussed below.

File Processing

Due to confidential information like the exact circuitry of PCBs found in satellites or missiles designed by NASA and the MDA, test PCBs are needed. This is why software like Altium is needed as referenced by Team 1. There are examples of PCBs already created online like the Raspberry Pi as seen in Figure 3, however, due to the nature of this project, more populated PCBs might be needed to accurately carry out the simulation. More populated PCBs means adding more resistors, nodes, and capacitors than normal PCBs seen in the real world. This would be able to show the worst-case scenarios that the simulation could be put in since there are more opportunities for bridging.

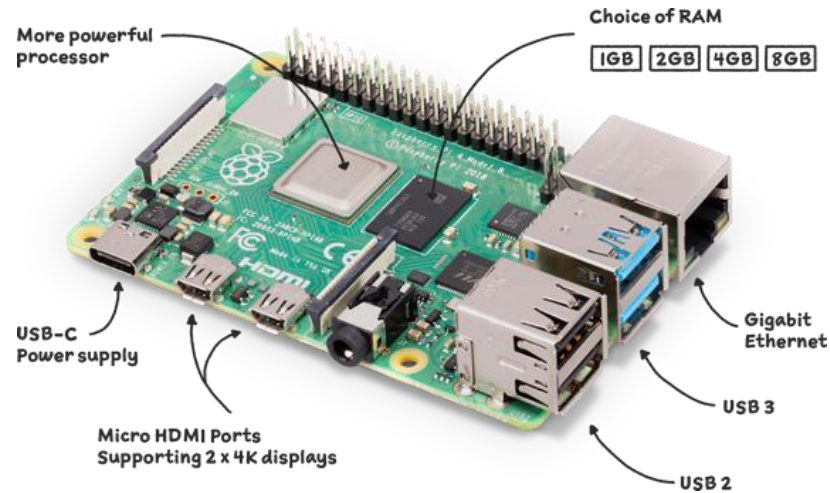


Figure 3: Raspberry Pi PCB

Material Processing

Another important task is the material processing on PCBs. This is required due to how every PCB has routes that the circuits follow that are covered by silicon, creating a barrier between them and the outside world. However, this is not the only insulated component regularly found on PCBs in today's world. There are multiple other circuitry components like capacitors that are also insulated as shown in Figure 3. This means that the software needs a method to identify certain geometry that is either insulated or not and what materials those geometries are made of. Team 1 has addressed a need for this application and believes that a method through Altium can be used to implement this in Unity. This is an area that will be investigated more with members of Team 1 and experts to see if it is truly feasible or not within our simulation.

Random Whisker Dimensional Distribution

Upon the user choosing either normal or lognormal distribution and inputting their desired length and width μ and σ , the Unity Engine will spawn the whiskers. If the user decides on a normal distribution, the μ value relates to the length and width's mean size, while

the sigma value relates to the standard deviation of the length and width from the mean size. If the user decides on a lognormal distribution the mu value relates to the location and the sigma value relates to the scale.

CHAPTER 6 CHANGES TO 2D SIMULATION

To understand the fundamentals of Unity, Team 2 was tasked with altering the 2D simulation created by Team 1. Some of the key changes that were conducted were changing the x-y planar view to an x-z planar view, nomenclature, and the output. This allowed Team 2 to create a simulation that was not only usable for the customer but also allowed Team 2 to train and gain experience with Unity. All the scripts referenced in this chapter can be found in Appendix A.

Change From X-Y Planar View to an X-Z Planar View

The first change that was implemented to the simulation was changing from an x-y axis camera which Team 1 chose for their simulation to a x-z axis camera as seen in Figure 4. This was done by implementing a planar change visually.

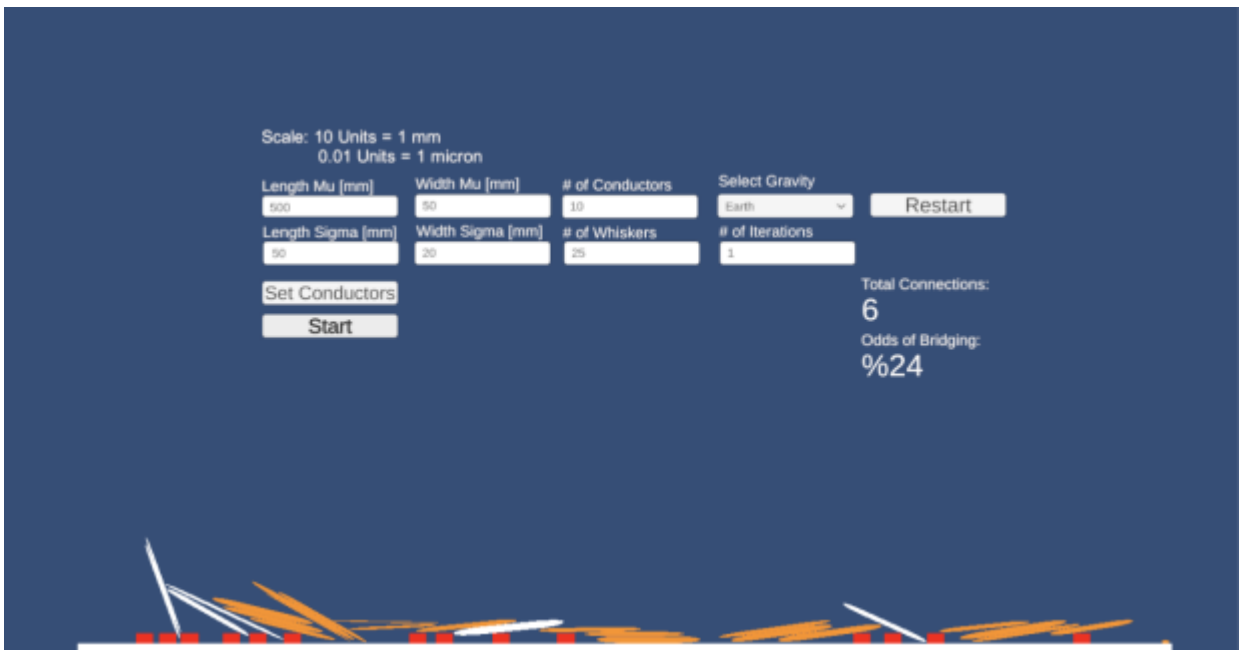


Figure 4: Team 1 2D Simulation with x-y camera axis

This change presents the user with a top-down perspective of the PCB instead of looking at a side profile, which can be seen in Figure 5 below. This means that instead of watching the whiskers sprinkle down on the PCB as shown in the previous simulation, the whiskers spawn in at random locations to simulate a completely irregular drop point and timing of whisker formation. This allows customers and other teams to read the PCB's footprint with capacitors and whiskers on it.



Figure 5: Team 2 2D Simulation with x-z camera axis

To accompany this point of view change, simulating gravity became impossible. As the base Unity program utilized a simple 2D template, simulating a “drop” of whiskers from this new viewpoint was not able to be implemented. To accommodate this change, whisker gravity needed to be turned off which was done by changing the code in “GravityControl.cs”.

Simulation Units

As indicated in Chapter 5, standardization of simulation units is critical to simulation functionality. The team had decided on a simulation standard of equating 100 pixels to 1 cm, as whisker of focus are millimeters or even microns wide.

Starting on the simulation, standard units were one of the first things the team decided to pursue. Upon initial investigation, it was discovered that Team 1 had already implemented this exact standard within their simulation which is shown in the top left corner of Figure 6.

Scale: 10 Units = 1 mm
0.01 Units = 1 micron

Length Mu [mm] <input data-bbox="219 821 435 856" type="text" value="Enter text..."/>	Width Mu [mm] <input data-bbox="467 821 683 856" type="text" value="Enter text..."/>	# of Conductors <input data-bbox="711 821 927 856" type="text" value="Enter text..."/>	Select Gravity <input data-bbox="959 821 1166 856" type="text" value="Pick One"/>	<input data-bbox="1198 821 1409 856" type="button" value="Restart"/>
Length Sigma [mm] <input data-bbox="219 898 435 934" type="text" value="Enter text..."/>	Width Sigma [mm] <input data-bbox="467 898 683 934" type="text" value="Enter text..."/>	# of Whiskers <input data-bbox="711 898 927 934" type="text" value="Enter text..."/>	# of Iterations <input data-bbox="959 898 1166 934" type="text" value="Enter text..."/>	

Total Connections:

Odds of Bridging:

Figure 6: Metal Whisker Team 1's Simulation Units

Boundary Box/Container

The design chosen for the boundary box modeling was an outline of the physical geometry. This was chosen so that visibility of the CCA was still maintained while understanding the general borders of the simulation. For the 2D simulation, the working principle shown for the boundary box, as seen in Table 1, wasn't applicable. Since the working principle was designed for a 3D simulation, a direct implementation of this design proved unfeasible. Regardless, the team carried out implementing a simplified version of this design that was fitting for a 2D rendition.

Following the changes to the CCA size, the gravity component of the spawned whiskers was disabled to simulate the top-down view. The rigid body collisions and interactions were maintained so conductors and whiskers would be simulated correctly. Following these changes, boundary walls were created within Unity's native interface to contain the conductors and whiskers. These walls were created with a toggleable visibility and a relatively thin outline to replicate the design decision, as can be seen in Figure 7. Enabling rigid body collisions for these wall components ensured conductors and whiskers would spawn and remain within this boundary.



Figure 7: Boundary Wall Additions

Whisker ID

As shown in Team 1's simulation, the whiskers that made a connection were simply indicated by a change in color from orange to white. To increase visibility of a bridged whisker that is much smaller in dimension (ie. 1 micron in width), an alternative had to be implemented. In doing so, a text box connected to the whisker was applied that would indicate its whisker ID for the user. The simulation from Team 1 did not have the whiskers numbered chronologically

from the time they were spawned, so the number ID name change was needed to complete our desired output. This was done by amending the existing “WhiskerControls.cs” code so that each clone would iterate a counter. This counter would then relay the numerical information of the whisker and attach the whisker ID as shown in Figure 8.

After implementing the number ID, a text box was added that reads the number ID of the whisker. This text box was subsequently hidden so the overlap of all the numbers wouldn’t confuse the user. After the whisker made a connection the text box was made visible so the user can see the number ID of the whisker that is making a bridge. The additional implementation of the hover function was also important as when a whisker was not making a connection you had no idea the number ID of the whisker. This made it so that whenever the user would hover their mouse over the whisker the text box which contained the number ID of the whisker would be visible, and after the mouse left the whisker the text box would then go back to hidden. This feature was added through the creation of the “WhiskerHover.cs” script.

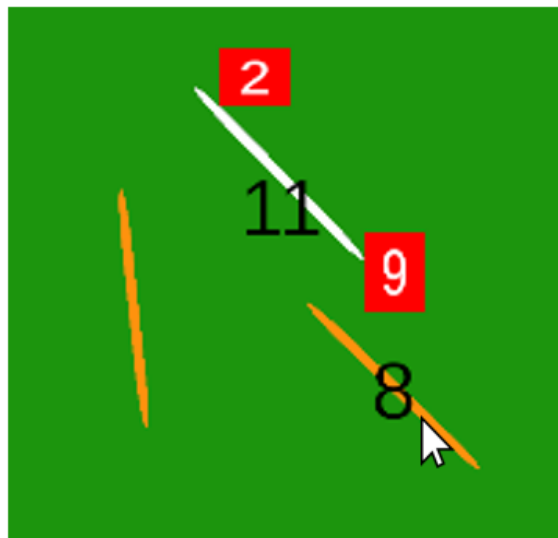


Figure 8: Whisker Hover and Bridge ID Feature

In addition to the “WhiskerHover” feature, changes were made so that whiskers were easier to identify within the simulation itself. Originally having been named “WhiskerClone”, the revision of whisker generation and naming allowed for the simulation hierarchy in Unity’s native interface to contain the whisker ID number as shown in Figure 9.

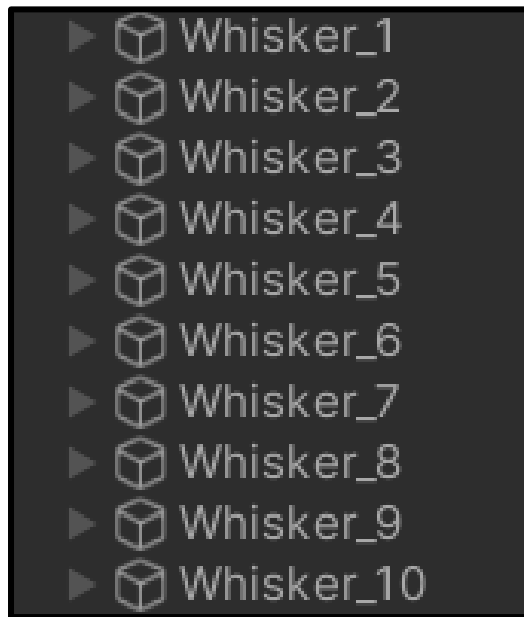


Figure 9: Whisker ID in Unity Hierarchy

Pad ID

Pad identification, similar to whisker identification, is an extremely important addition to the simulation. Within the 2D simulation, every pad acts as a conductor where if a whisker comes into contact with two of them at once, it will cause a short. In the 3D simulation that Team 1 has been working on and Team 2 will take over, pads are any conductive material that can cause a whisker to bridge. In the simulation, it is understood that each pad acts as a conductor, which is why within unity the pads are labeled as “Conductor_#”. This addition is visible in Figure 10 which shows the conductor names within Unity’s simulation hierarchy. This was done by changing the code in “ConductorControl.cs”.

To help the user identify the conductors, a visual number was added to each conductor. Furthermore, this will help the user identify the conductors that come into contact with whiskers as well as help the user correlate the conductor numbers shown in the output files to the conductors they see within the simulation. The conductor numbers shown on top of the conductors, which can be seen in Figure 11, correlate with the conductor's name within Unity. For example, Conductor_5 will have the number 5 on top of it.

To create a number over each conductor a text box was added and attached to the conductor template within Unity. Since the conductors are spawned randomly on the CCA, this feature makes it so that the numbers are always attached and show on top of each conductor. The numbers are assigned within the C# code by identifying the conductor's name within unity and matching the appropriate number to the conductor in the game view.

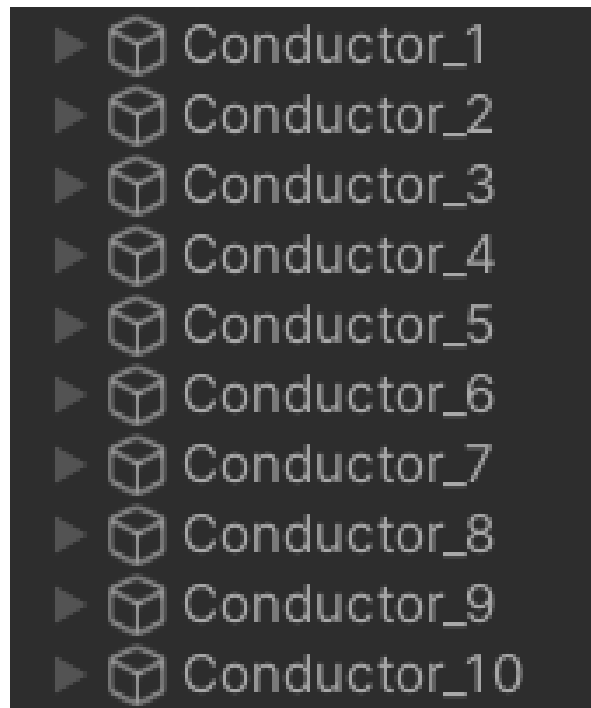


Figure 10: Conductor ID in Simulation Hierarchy

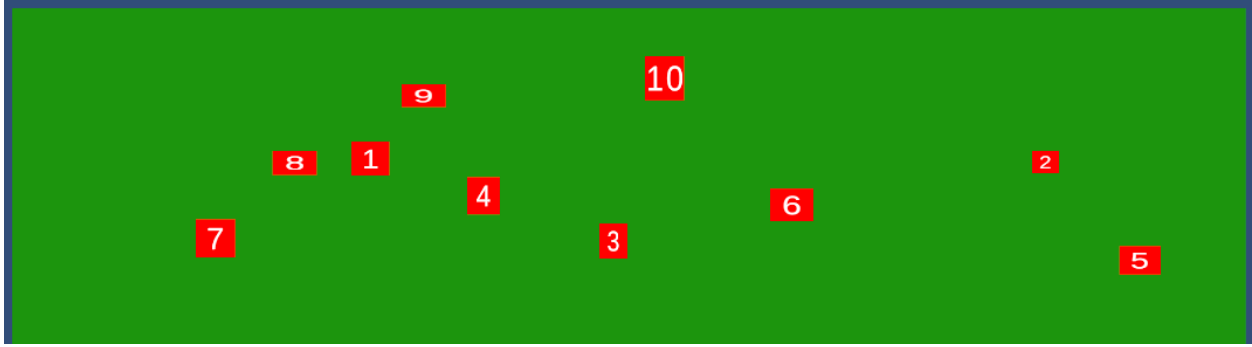


Figure 11: Conductor ID and Randomization

Storable Text Files

The importance of storable output files was made evident for immediate data analysis. For this rendition of the simulation, the following data was stored: whisker ID, whisker length, whisker width, iteration number, bridged whiskers, and bridged conductors. All this data was compiled and stored in .csv files. An example of this can be seen in Figure 12 where whisker ID, whisker length, and whisker width were compiled into a .csv file. This was done by adding code that could read and write files to the scripts that controlled the respective functions of each component. The final .csv files created for output purposes and their functions are shown in Table 2. These files were compiled through “csvReader2.cs” so that all relevant data for a given simulation can be centralized and viewed within one file. A final .csv was created called “CompletedWhiskerData.csv” which compiles all the data gathered into a single file as seen in Figure 13. For demonstrative purposes, Excel was used to show the .csv file.

	A	B	C	D	E
1	Iteration	WhiskerNumber	Length	Width	
2	1	1	596.464	35.70695	
3	1	2	636.2159	37.84564	
4	1	3	369.0472	29.48767	
5	1	4	502.6815	23.01779	
6	1	5	686.168	27.18748	
7	2	1	477.7106	46.70544	
8	2	2	498.3324	57.92635	
9	2	3	599.0598	37.74536	
10	2	4	478.4985	25.27005	
11	2	5	644.5417	41.06217	
12	3	1	615.5252	24.58461	
13	3	2	598.0656	53.49779	
14	3	3	484.424	36.19971	
15	3	4	410.6161	24.84986	
16	3	5	437.9542	36.6909	
17	4	1	387.4272	28.47632	
18	4	2	356.4211	48.94358	
19	4	3	413.6216	53.75133	
20	4	4	418.6242	32.0188	
21	4	5	514.5965	36.08313	
22	5	1	483.0211	29.33657	
23	5	2	440.4583	51.17668	
24	5	3	378.5417	51.85907	
25	5	4	589.5085	60.01059	
26	5	5	593.6145	41.73283	
WhiskerDimensions					+

Figure 12: Whisker Dimension .csv File

	A	B	C	D	E	F	G	H	I
1	Iteration	WhiskerNumber	Length	Width		Iteration	WhiskerID	Connection1	Connection2
2	1	1	596.464	35.70695		1	Whisker_5	Conductor_8	Conductor_5
3	1	2	636.2159	37.84564		1	Whisker_4	Conductor_1	Conductor_8
4	1	3	369.0472	29.48767		3	Whisker_4	Conductor_2	Conductor_17
5	1	4	502.6815	23.01779		5	Whisker_3	Conductor_9	Conductor_18
6	1	5	686.168	27.18748		6	Whisker_3	Conductor_20	Conductor_6
7	2	1	477.7106	46.70544		6	Whisker_4	Conductor_8	Conductor_13
8	2	2	498.3324	57.92635		10	Whisker_2	Conductor_11	Conductor_5
9	2	3	599.0598	37.74536		10	Whisker_4	Conductor_13	Conductor_1
10	2	4	478.4985	25.27005					
11	2	5	644.5417	41.06217					
12	3	1	615.5252	24.58461					
13	3	2	598.0656	53.49779					
14	3	3	484.424	36.19971					
15	3	4	410.6161	24.84986					
16	3	5	437.9542	36.6909					
17	4	1	387.4272	28.47632					
18	4	2	356.4211	48.94358					
19	4	3	413.6216	53.75133					
20	4	4	418.6242	32.0188					
21	4	5	514.5965	36.08313					
22	5	1	483.0211	29.33657					
23	5	2	440.4583	51.17668					
24	5	3	378.5417	51.85907					
25	5	4	589.5085	60.01059					
26	5	5	593.6145	41.73283					
CompleteWhiskerData									+

Figure 13: Complete Whisker Data .csv File

Table 2: .csv File Names and Their Functions

.csv File Name	Function
allWhiskerData.csv	Contains iteration number, whisker ID number, and respective length and width dimension
CombinedData.csv	Contains all whisker dimensions for each iteration number
CompleteWhiskerData.csv	Contains iteration number, whisker ID number, respective length and width dimensions in addition to bridging information which contains the iteration number, whisker ID, and the two connecting conductor IDs for the connection
ExportedNumbers.csv	Contains whisker length data
ExportWidths.csv	Contains whisker width data
WhiskerConnections.csv	Contains iteration number, whisker ID number, and the connecting conductors for all bridges in a run
whiskernames.csv	Contains whisker ID numbers

Output Window

In addition to having .csv files for data collection, a separate output window, shown in Figure 14, was created within Unity's native interface using the script "OutWindow.cs". This script triggered the creation of a window titled "CSV and Screenshot Window" and allowed the user to be able to type in a desired .csv file into a text box to view. The window that displays the data can then be updated by clicking the "Load CSV File" button at the top of the window. This functionality allows users to be able to examine data and carry out a preliminary analysis of the simulation straight from Unity without the need for exporting text files for every simulation carried out afterward.

This script also enabled a screenshot functionality that was coded within the same script so that users could take a photo of the game window during any game state to complement the .csv file data. This can be done by pressing the “Capture Screenshot” button at the bottom of the window, and the output file name can be edited in the text field above that button. All the .csv and screenshot file references are in the assets folder of the Unity project folder. This feature allows for reference back to certain simulation states to pair with the .csv file data that can be exported so that analysis can be carried out in the future, even after multiple simulation attempts.

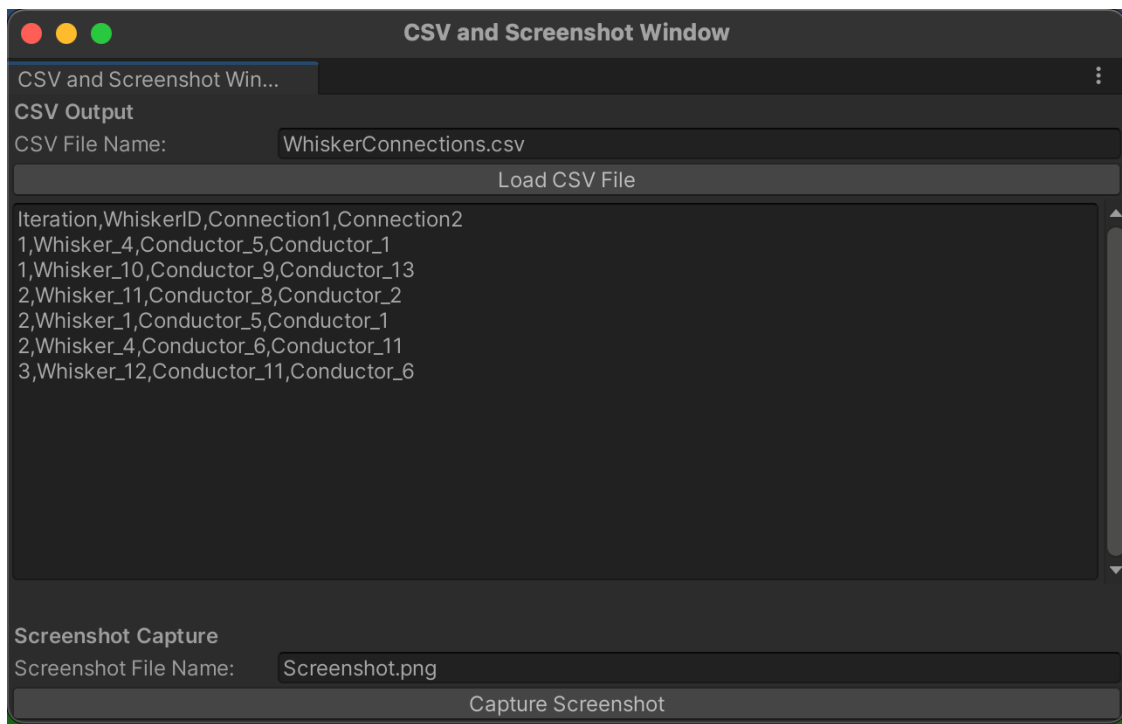


Figure 14: CSV and Screenshot Window

Graphical Interface Camera

The need for the user to see the smaller and more realistic-sized whiskers in the simulation was necessary so that when a more realistic simulation is being run an output could be reflective of a realistic simulation, the problem can be seen in Figure 15. Unity natively can

zoom in within the simulation, which allows the user to zoom into the circuit board and get a more detailed view, as shown in Figure 16. The problem with zoom is that it only zooms in on one location and cannot be zoomed in on another location. The implementation of the “CameraMove.cs” script which allows the user to move the circuit board using the arrow keys or WASD keys, was needed so the location at which the user zoomed in would be able to move based on the user shifting around the circuit board, shown in Figure 17.



Figure 15: Initial view of Simulation



Figure 16: Zoomed in Simulation



Figure 17: Zoomed In with Board Shifted in Simulation

CHAPTER 7 OPERATION READINESS SECTION

This project was a revision of Team 1's code and Unity, the testing was conducted throughout the semester. For testing the code would be altered with new changes which was then followed by immediate testing of said new changes and then the process would repeat. This means that we did not follow the typical timeline or format for most projects which would make design critiques or come up with new ideas and then at the very end combine the design changes into the final project. Therefore, the testing part of the ORR will be a list of the changes made throughout the report, which will then be followed up with discussion of the changes made by Team 2, problems faced, and changes that can be followed by future teams or organizations.

Despite the differences, Team 2 was still able to apply the knowledge gained from the 2D simulation to come up with better ways to implement the features that were experimented with previously into the 3D simulation.

Testing

There are six major changes that were made by Team 2 to Team 1's code. Team 2 improved the user interface (UI), worked on improving the visual analysis, improved simulation manipulation, worked on identifying critical pairs and node, improved the screenshots and output file, and created a detailed Excel file for data analysis. Scripts related to these are referenced in their respective section and can be found in Appendix D. Furthermore, an online beta version of this program was created and updated periodically so customers could give feedback on the simulation as the project progressed.

Taking the experiences from the 2D simulation work into consideration, methods of implementation and better coding techniques were utilized to implement features into the 3D

simulation. Keeping the 2D simulation in mind allowed the team to more easily conceptualize the features and functionality required by the program to be able to perform in the manner that the customers and team members desired.

Online Beta

As the project progressed, the customers stated a desire for the ability to beta test the simulation themselves. In doing so, they would be able to provide feedback on the usability and accuracy of the simulation. To make the simulation accessible to the users, a WebGL version of the program was built within Unity to be able to upload to GitHub. Through the GitHub repository, a web address was designated and sent out to customers to be able to access through their company devices without the need for them to download any files and pass security clearances.

This beta version, seen in Figure 18, was updated periodically when substantial features were added. When this version was updated, an email was sent out to customers with details of updated functions and a general “how to use” instruction set for each function implemented into the beta. The feedback from the beta was monumental in making changes so end-users could have an intuitive and relatively easy experience using the final program. At the time of writing, this version is still accessible through the link in Appendix C.

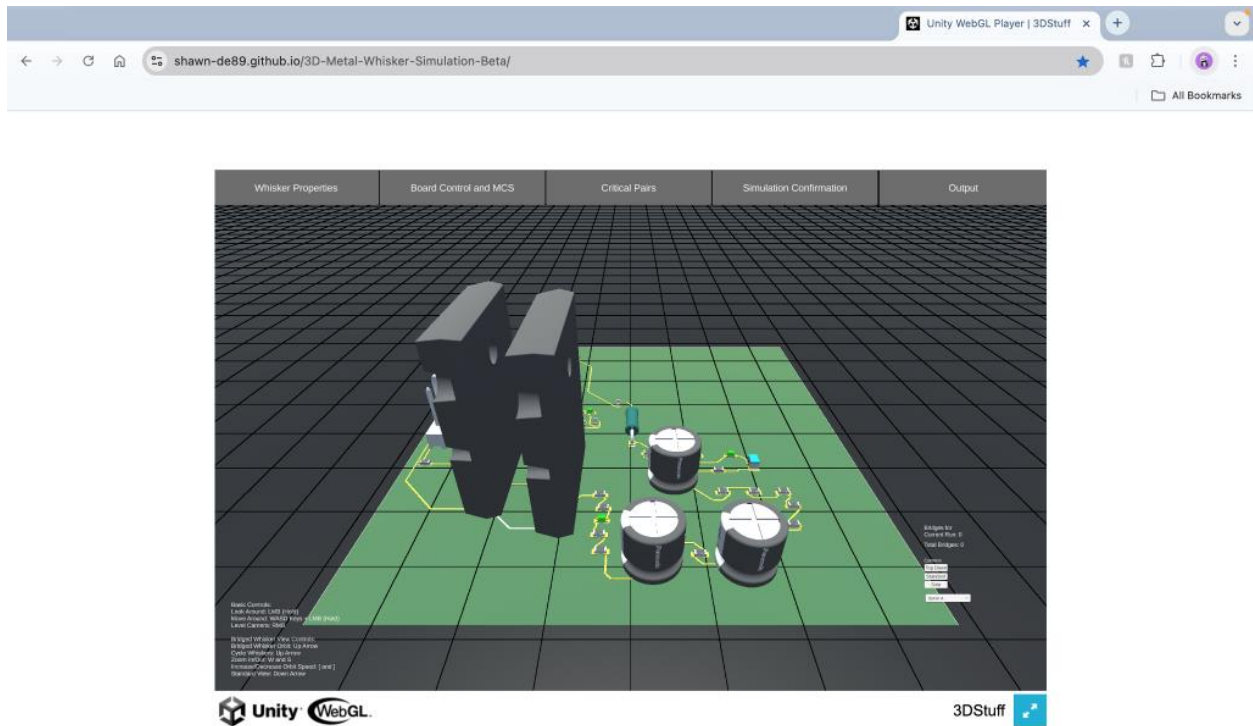


Figure 18: Online Beta

UI Improvements

An initial important improvement made was the change of the user interface. While it did not add more customization to simulation parameters, it added ease of use so the user can better find the functionality they desire while keeping the screen relatively clean. Team's 1 user interface, as seen in Figure 19, while functional was constantly on screen and made it difficult to see what was fully going on with the simulation without the user interface being in the way. As such, it was necessary to have the user interface be removeable while still being able to be accessed if needed, so a tab-based interface was decided.

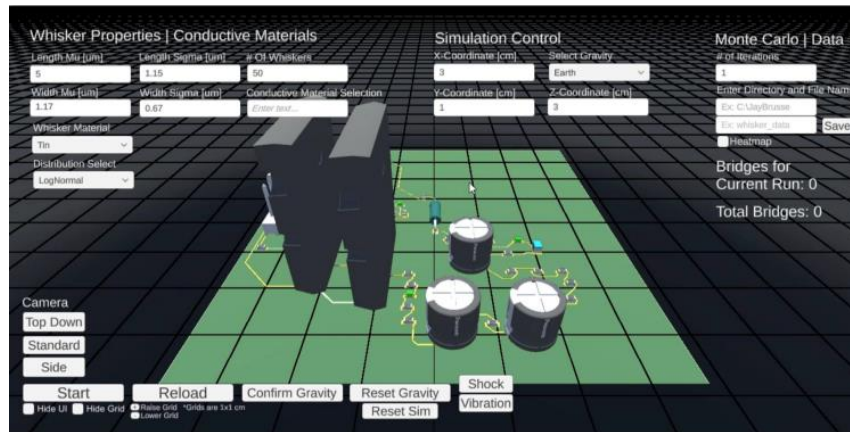


Figure 19: Team One's Original User Interface

Having tabs at the top of the screen that can be hidden was the design that was settled on and then sectioning Team 1's already existing user interface into themed tabs helped to carve out a basic idea of how the user interface would work. As more functionality was added, the user interface grew; towards the end of the project the tab's themes were finalized and all the user interface fields were organized as such in each tab as shown in Figures 20 and 21.

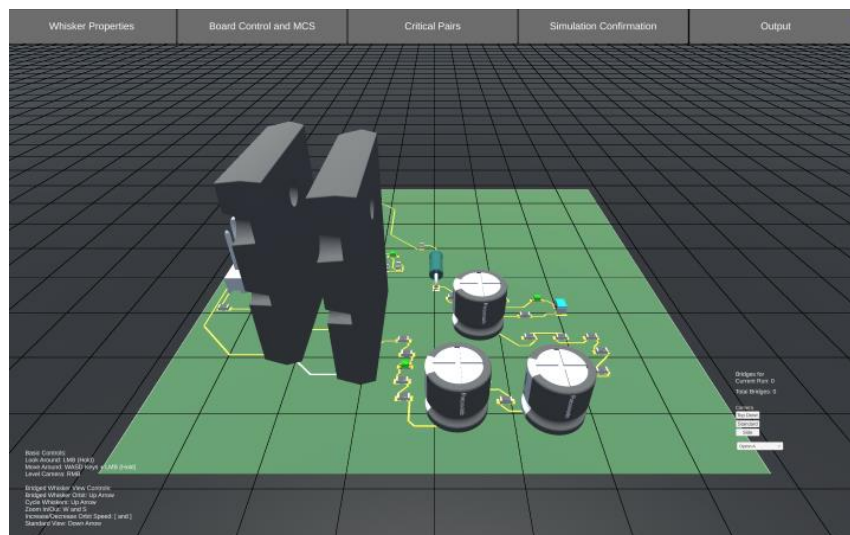


Figure 20: Team Two's Revised User Interface

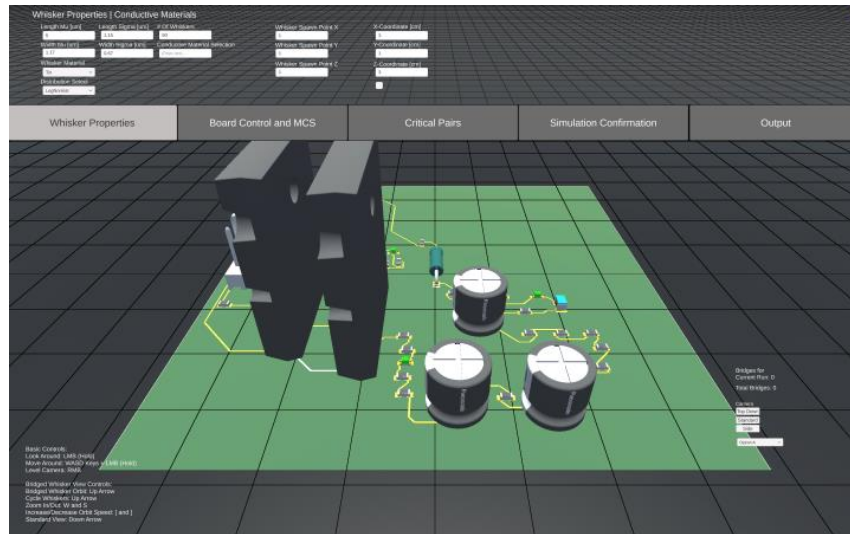


Figure 21: Team Two's Revised User Interface with a Tab Open

This user interface was continuously updated as progress was made on other features. In addition, using the online beta mentioned previously, the customers were able to get hands-on experience with the simulation and provide feedback as to what functionality should be added further, and how to better organize the user interface.

Implementing the new user interface entailed a large reorganization of the game object hierarchy within the Unity Editor and the addition of several scripts shown in the Table 3 below.

Table 3: User Interface Scripts and Their Functions

Script	Function
UIScript.cs	Controls most user interface inputs. This script was modified to better function with the new user interface
TabControllerStay.cs	Controls tab so that when a button attached to this script is pressed, the desired tab is relocated to a desired spot to “open” the tab
TabControllerFront.cs	Controls tab so that when a button attached to this script is pressed, the desired tab is shown in front of all other user interface objects.

As previously stated, this user interface change improved the ease of use for the customer and allowed for better organization of the inputs and simulation controls, which allows the user to use the simulation in the most efficient and intuitive way possible.

Better Visual Analysis

The customer stated a desire for better capabilities regarding visual analysis. This would enable users to inspect specific, critical components and obtain information about the bridging statistics of said component. In addition to monitoring critical components, this functionality would allow the user to further study whisker behaviors under specific mission environments. From a better visual analysis tool, users can gather which area of the circuit board is more likely to have more whiskers, or vice-versa, prompting potential design changes to accommodate for whisker behavior within mission environments.

Better visual analysis was implemented through “CamOrbit.cs” in conjunction with “TriggerTracker.cs” and “WhiskerData.cs” to be able to inspect whiskers. These scripts enable users to cycle through focused views of bridged whiskers using the up and down arrow keys. Additionally, users can zoom in and out through the use of the ‘W’ and ‘S’ keys, as well as control the speed at which the camera orbits the target whiskers using the ‘[’ and ‘]’ keys. This feature can be seen in action in Figure 22.

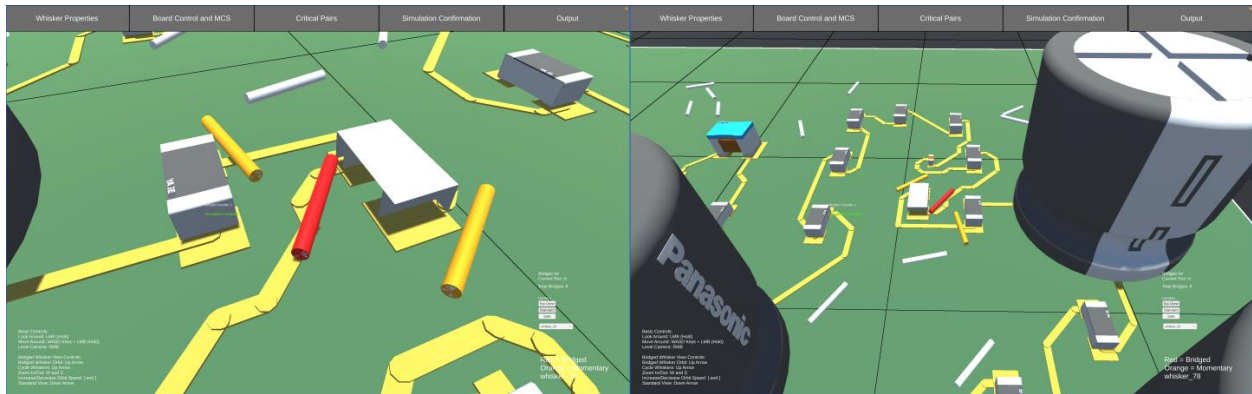


Figure 22: Bridged Whisker Zoomed In (left) and Zoomed Out with Rotation (right)

To complement this feature, and in keeping with the design decision to change whisker colors when bridged, color changes were implemented and improved from the 2D simulation so that users can distinguish which whisker is currently bridged (red) or was momentarily bridged (yellow/orange) as seen Figure 22, 23, and 24. Users also can choose a specific whisker to view, rather than cycling through them, using the drop down in the bottom right corner of the screen as shown in Figure 25. In addition, they can alter the angle at which the whiskers are viewed using the camera buttons above the dropdown, also shown in Figure 25.

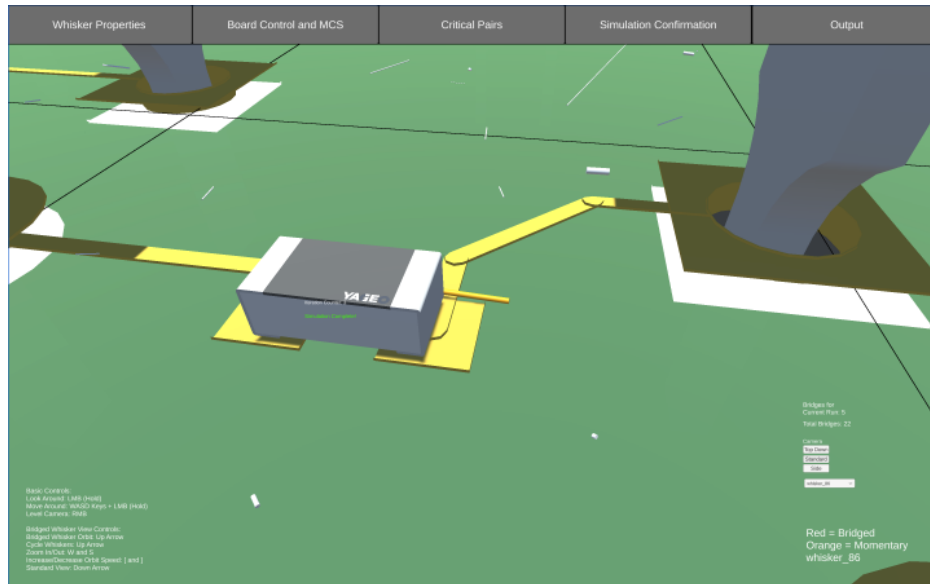


Figure 23: Close-Up of Momentarily Bridged Whisker

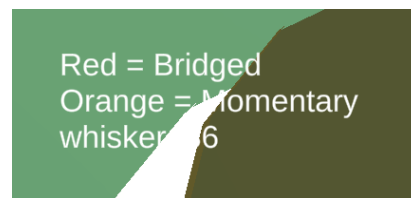


Figure 24: Whisker Number Indicator when in Close-Up View of Bridged Whisker(s)

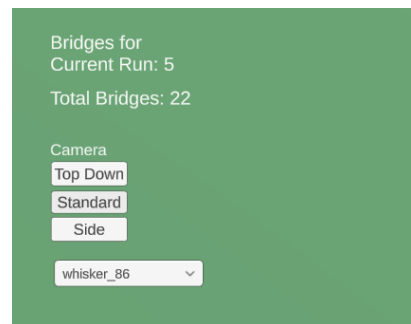


Figure 25: Camera Controls and Drop Down to Target Whisker to View

With these two features combined, the team concluded, with additional feedback from customers and sponsors, that visual analysis was improved to indicate specific bridged whiskers and was sufficient enough to conduct specific analyses related to certain pads, whiskers, and critical components.

Simulation Manipulation

The new user interface allows the user to manipulate the simulation in the way that they desire, while also designed to minimize the space the user interface takes up on screen so the user can focus and clearly see the simulation. The user interface is sectioned into five tabs, Whisker Properties, Board Control and MCS (Monte Carlo Simulation), Critical Pairs, Simulation Confirmation, and Output.

The Whisker Properties tab shown in Figure 26 gives the user the ability to control the distribution values for the whiskers (μ , σ), the material for the whiskers, distribution type (normal, lognormal), the amount of whiskers and the conductive material that if the whiskers touch would bridge; on the left side of the tab the user has the ability to change the area in which the whiskers spawn and the location of the area in which the whiskers spawn with the added ability to make the whisker spawn area visible or invisible. This toggle was coded in through “WhiskerSpawnCubeVisibility.cs” and “WhiskerCubeScaler.cs”. The first script makes it so when the toggle is checked by the user, the spawn area that is designated by the user’s inputs is made invisible or translucent. The second script scales the cube to the dimensions designated by the user through the inputs.

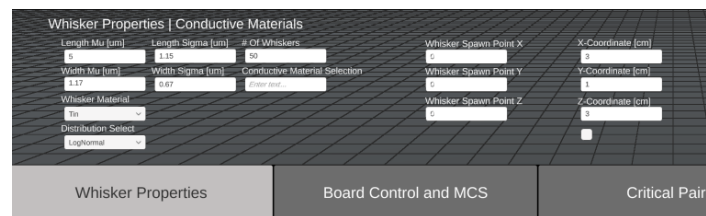


Figure 26: Whisker Properties Tab

The Board Control and MCS tab shown in Figure 27 has a variety of different functionality on the far most left is the gravity select, which has the additional ability that if the

user selects ‘Custom Gravity’ they can customize the force in the X, Y, and Z axis. Right of the gravity selection is the Monte Carlo section with allows the user to select the number of iterations, a save location for the output data, and the ability to save and load inputs to make the simulation easier to use and reuse. Left of the Monte Carlo section is the boundary section which allows the user to add a wall around the circuit board and a ceiling if the user wants. Left of the Boundary section is the rotate and spinning section which allows the user to give the circuit board an initial rotated offset and spin the board in the desired direction(s) and speed(s) to mimic more extreme conditions. On the far right of the tab is the shock and vibrate sections which allow the user to give the circuit board an initial shock movement or a continuous vibrating force to the circuit board.

Custom gravity can be selected using “DropDownHandler.cs” and “WhiskerControl.cs” within which the drop down to select the desired gravity was handled by the first script. The second script implemented the user’s gravity selection by changing the value of the gravitational effects that is put on whiskers.

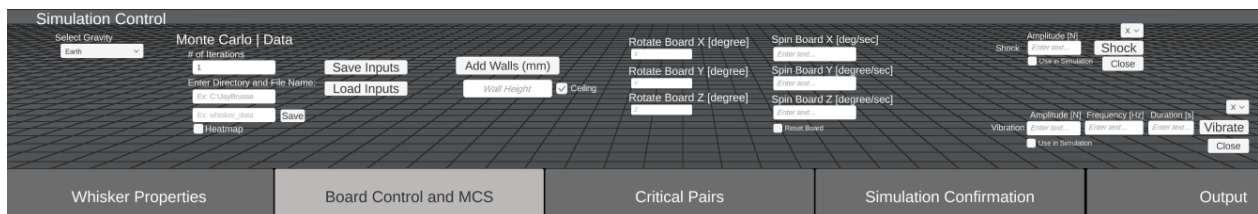


Figure 27: Board Control and MCS Tab

In Figure 28 the whisker spawn area is a blue translucent cube that is above the circuit board, on the right side of Figure 28 shows how the whisker spawn area can change size to fit the customer’s specific location. In Figure 29, the whisker spawn area can be seen again and on the right side of the figure the whisker spawn area is offset showing that the user can place the spawn area wherever they would desire.

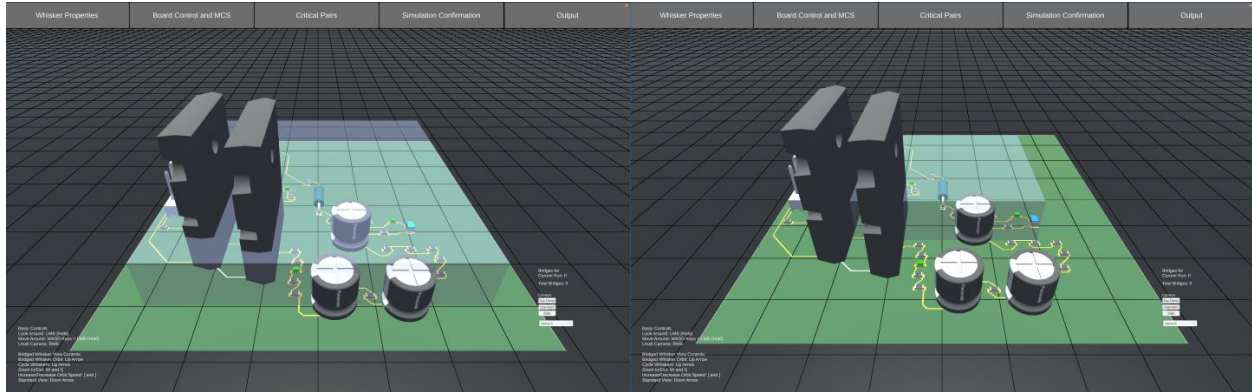


Figure 28: Default Spawn Area (left) and Custom Spawn Area (right)

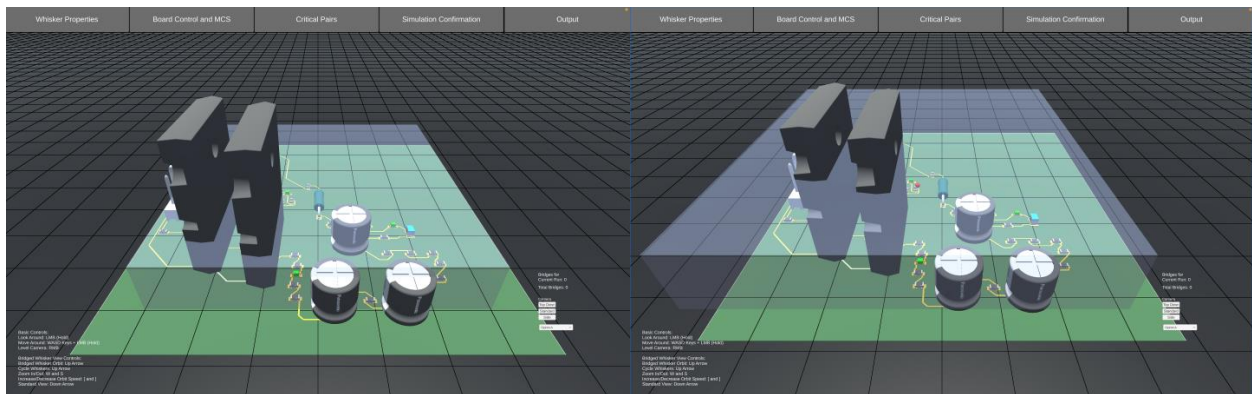


Figure 29: Default Spawn Location (left) and Customer Spawn Location (right)

In Figure 30, the effect of normal gravity on the white whiskers can be seen on the left, and on the right side of the figure the custom gravity can be seen moving the whiskers in the upward direction. In Figure 31 on the left the board is non-rotated and is parallel with the ground, while on the right the board has been given a rotate value making it askew. In Figure 32 on the left is the board normally, and on the right the white translucent borders added are the walls and ceiling at a given height. This can be useful to the user to simulate gravity when going up a hill, on other planets, or if their CCA would be upside down at any point.

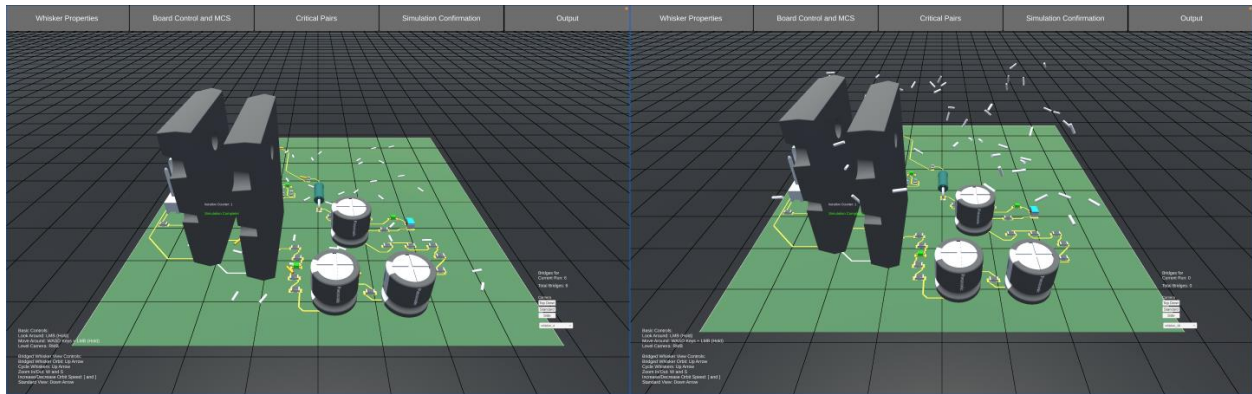


Figure 30: Whiskers in Earth Gravity (left) and Custom 2 Newton Gravity in the Y-Direction

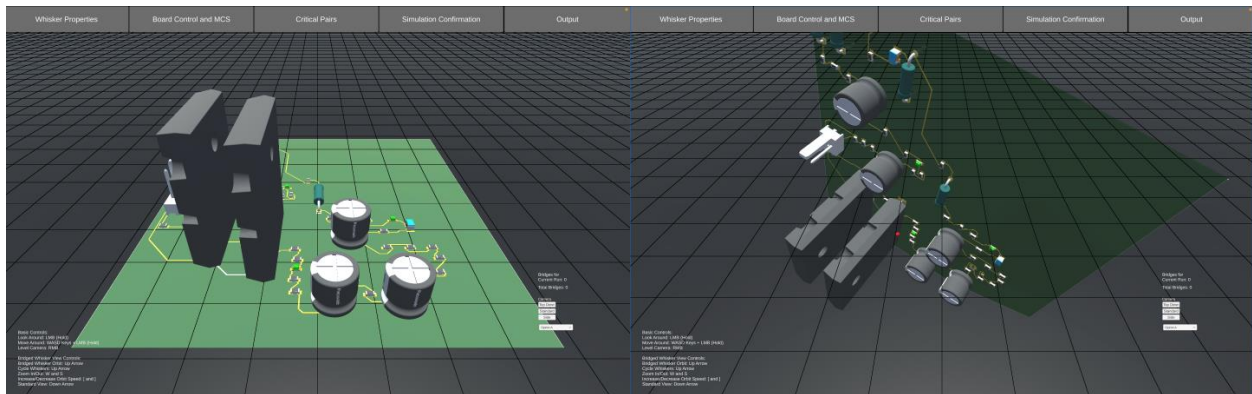


Figure 31: Default Board (left) and Board with Custom Rotation (right)

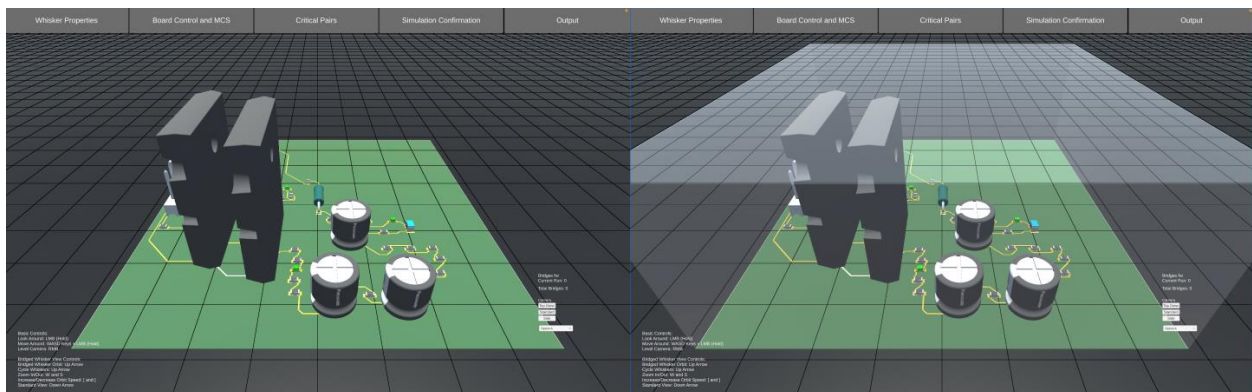


Figure 32: Simulation with Walls Added

Overall, the accuracy of whisker behavior in a mission environment was improved, and more in-depth board and simulation manipulation was achieved with the additions mentioned. These features will allow the users to manipulate the board to simulate mission environments more realistically and will provide for better design decisions based on whisker behavior within the simulation.

Critical Pairs/Nodes

To enable more in-depth analyses of bridging statistics, the ability to select “critical” pairs and nodes was desired by customers. This feature would entail selection by the users of two conductors of importance via the “Critical Pairs” tab shown in Figure 33. When those two conductors are bridged by a whisker the data related to the whisker dimensions, whisker resistance, and other relevant statistics will be outputted to a specific section in the output file for those critical pairs. Using the interface shown in said figure, users can specify any two conductors on the board as “Critical” for post-simulation statistical analyses.

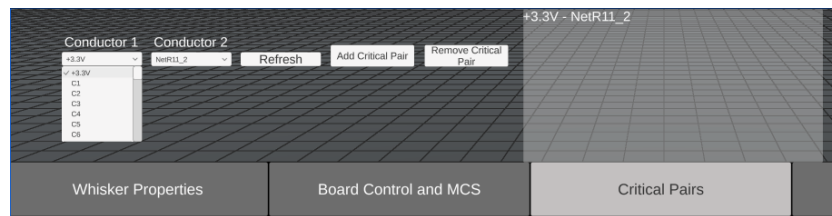


Figure 33: Critical Pairs Tab

This function can be extremely useful to the user because it allows them to prioritize specific node pairs that could have a severe negative impact if bridged. In addition, the addition of critical pair indicators allowed for the output file to be organized in a manner such that users can easily navigate the conductors of interest rather than combing through the entire dataset’s worth of information.

Output File/Screenshot

With the additional features added, numerical statistics became of further importance to be able to access and study. To that end, several features were added so that users could analyze the statistics related to their desired simulation parameters.

Team 1 had started on an output file that could be updated as more features were added. As Team 2 added the features discussed in this report, this output file was modified to reflect those additions. When the code was originally handed off, the previous team had already added a function to designate a name and directory location for this file which can be seen in Figure 34. The input field shown would then be utilized by the script “SaveManager.cs” to save a .csv file to the desired location (ie. C:/) as shown in Figure 35.

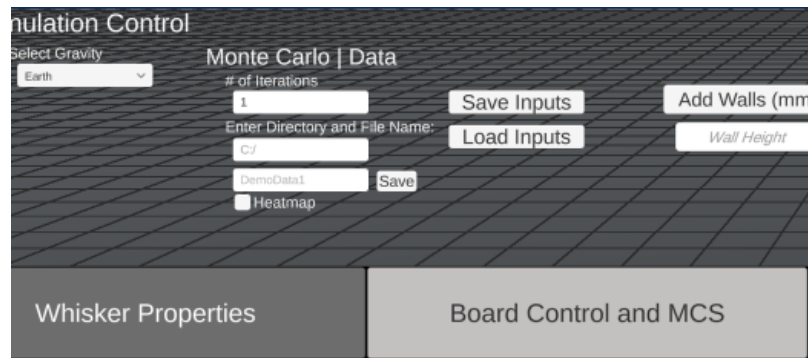


Figure 34: Directory and File Name Input Field

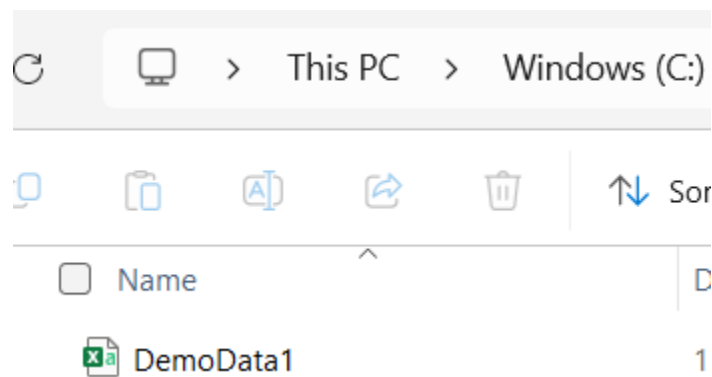


Figure 35: Excel File Name Based on Input Field

As stated previously, updates were made as new features were added so that relevant statistics could also be studied and analyzed by the end user. Added features include Critical Bridged Whiskers report in addition to Simulation Input Parameters that were specified by the user. An example of this .csv file (for this instance “DemoData1.csv”) can be seen below in Figure 36 A simplified version of this file can be accessed within the simulation as shown in Figure # in the section titled “UI Improvements”

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
1	All Whiskers						Bridged Whiskers								Critical Bridged Whiskers								Simulation Inputs	
2	Whisker #	Length (um)	Width (um)	Resistance	Iteration		Whisker #	Length (um)	Diameter (um)	Resistance	Iteration	Conductor	Conductor 2		Whisker #	Length (um)	Diameter (um)	Resistance	Iteration	Conductor	Conductor 2		Parameter Value	
3	1	706.0898	4.969244	3.968404	1		80	223.639	3.811064	2.136937		1 R29	NetL2_2										Length Mu	6
4	2	159.4535	4.346496	1.171365	1		26	168.5686	4.145041	1.361619		1 R29	NetL2_2										Length Sig	1.15
5	3	550.9734	8.86267	0.973505	1		133	778.3899	0.710864	213.7767		1 O5	NetC6_2										Width Mu	1
6	4	119.2566	16.53219	0.162112	1		28	128.8453	2.462929	2.947822		1 Q1	GND										Width Sign	0.67
7	5	1459.491	3.21008	19.65651	1		57	1171.109	6.938114	3.376379		1 Q2	NetQ2_2										# Of Whisk	200
8	6	1324.919	2.72277	24.80299	1		188	699.7596	2.861658	11.85905		1 R30	NetR29_2										Conductiv	copper
9	7	470.5246	1.605512	25.33331	1		34	2144.558	6.43328	7.191336		1 R7	NetC1_1										Whisker M	Tin
10	8	456.5552	3.867317	4.236531	1		159	2856.247	5.475912	13.21964		1 NetR11_2	R15										Distributi	LogNormal
11	9	654.5339	1.419883	45.05714	1		135	1010.912	2.680271	19.52956		1 C5	NetC5_1										Whisker St	0
12	10	243.6779	6.830623	0.724823	1		122	7290.693	1.833272	301.059		1 R19	NetD2_1										Whisker St	0
13	11	1505.699	3.402184	18.05341	1		31	3411.602	1.95098	124.3911		2 Q1	GND										Whisker St	0
14	12	180.1188	1.55321	10.36181	1		4	1170.141	1.764999	52.12977		2 C3	NetC3_2										X-Coord	3
15	13	951.4797	5.838612	3.873619	1		194	259.288	4.108179	2.132162		2 C5	NetC5_1										Y-Coord	1
16	14	929.0532	3.456557	10.7917	1		64	138.0602	3.676204	1.417771		2 NetR16_2	R17										Z-Coord	3
17	15	70.80274	5.386736	0.338638	1		163	3526.753	2.228796	98.53056		2 C2	NetC2_2										Gravity	Earth
18	16	1013.53	5.038774	5.540175	1		113	1883.842	2.6563	37.05328		3 Q1	GND										# of Iterati	15
19	17	112.4559	4.030137	0.969093	1		124	494.8606	3.873205	4.578029		3 GND	NetQ1_1										Directory F	C:\test
20	18	559.6284	5.033027	3.06604	1		77	1545.506	2.41092	36.9013		3 C3	NetC3_2										Save File h	DemoDataLc
21	19	200.4221	4.005005	1.734108	1		75	1626.337	7.338385	4.191275		3 NetR9_2	NetC2_1										Rotation-X	-90
22	20	11.71637	5.556308	0.052669	1		6	223.2751	2.050084	7.372819		3 Q1	GND										Shock Acti	N
23	21	172.8643	12.59717	0.15118	1		140	689.4528	1.637685	35.67635		3 NetR21_2	R21										Vibration f	N
24	22	757.0905	3.476016	8.696026	1		151	37.96212	3.33965	0.472373		3 L2	NetL2_1											
25	23	747.0449	2.313707	14.73757	1		82	1558.805	1.275644	137.9443		3 NetR4_2	R10											

Figure 36: Excel File of Output Data from Simulation

Despite the additions to the output file, this data was not available to users within the simulation. As such an in-simulation output tab was added as shown in Figure 37. After a simulation has been run, the output tab will look similar to how it does in the figure. The output tab is a condensed version of the output file described above and shows the length, diameter, resistance, and iteration of the whisker; the last two columns show the conductors that were bridged by the whisker.

Whisker #	Length (mm)	Diameter (mm)	Resistance (kOhm)	Thickness	Conductor 1	Conductor 2
378	148.9126	4.304008	1.115635	1	Q8	NetC8_1
37	41.90618	11.90191	0.0406039	1	R18	Net11_1
123	279.3888	3.647625	2.90382	1	Q5	Q10
222	104.632	16.30257	0.0693736	1	Q1	Q10
169	1813.943	2.996243	28.04181	1	Q1	Q10
270	55.00688	5.558792	0.287055	1	Q8	NetC9_1
258	905.7431	5.871298	3.993118	1	Q6	NetC6_1
136	877.425	6.627814	2.034144	1	R22	NetC22_2
331	64.33311	3.992773	0.5006614	1	R29	Net12_2
82	42.49613	5.905281	0.1669884	1	R38	NetC37_2
342	1132.981	6.504484	3.719411	1	R16	NetC16_2
29	182.7725	1.432111	12.52399	2	Q5	NetC5_1
318	676.824	1.853635	27.35783	2	R16	NetC6_1
122	591.7322	5.695544	2.525138	2	R35	NetC35_1
180	1796.98	2.232859	46.38216	2	R3	NetC3_2
113	249.4255	4.494284	0.902461	2	R35	NetC35_1
238	1827.578	2.823965	61.72727	2	D6	NetC6_2
229	1329.099	2.105669	41.63197	3	NetC3_1	D6
284	87.20119	1.877814	2.132819	3	R29	Net12_2
190	3178.306	2.416162	25.66831	3	Q16	NetC16_1
211	13.09969	8.925994	0.02881817	3	R29	NetC29_2
86	703.0936	5.86483	2.830866	3	R2	NetC2_1

Figure 37: In-Simulation Bridged Whisker Output Tab

Furthermore, a screenshot button was added so that users could take a picture of the simulation at any time during the simulation. This button can be seen under the Simulation Confirmation tab as seen in Figure 38. Additionally, a toggle for “Automatic Screenshots”, which can be seen in the below figure, was added so that screenshots would be taken between iterations without any user inputs. This feature automatically hides any user interface before taking the screenshot so that just the board and the whiskers are visible without any obstructions.

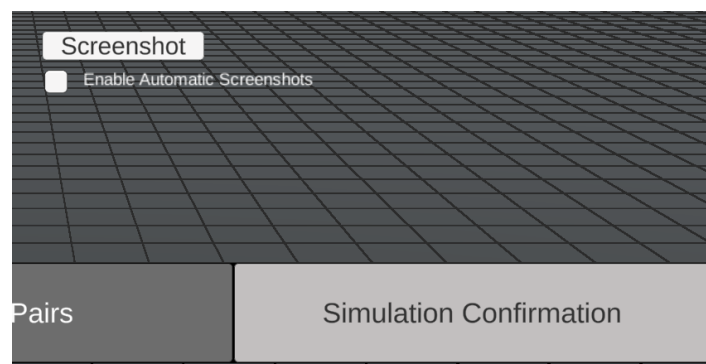


Figure 38: Screenshot Button

The screenshots are saved automatically to one’s desktop with the file name “3D Whisker Simulation - #” where the number is designated by an iterative counter within the script. The

saved file can be seen in Figure 39 below. Due to time constraints for this functionality, the ability to designate a location for the file to save at, as well as a name that the user can input was not added.

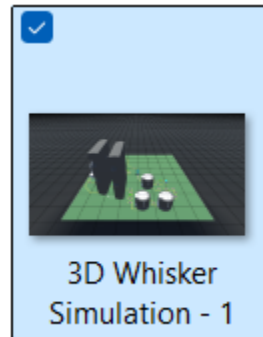


Figure 39: Screenshot File

By implementing these output file features, the ability to visually and statistically analyze the output data was improved. Using a separate Excel file, the .csv file created through the simulation could be copied in to provide charts and probabilities of the simulation. Further, the screenshot function allows for users to visualize specific aspects of the simulation to be able to compare with the statistics to better understand the physical behavior a whisker of importance.

Excel File for Data Analysis

The ability to efficiently analyze the simulation data was an important requirement for the customers. The customers requested a way to easily organize, summarize, and extract meaningful observations from the large sets of data generated from the simulation. Thus, Team 2 decided on creating an excel file to address this need by providing a structured and user-friendly format for data storage and analysis. The excel file will aid the users in performing a detailed review of whisker behaviors and bridging events that happened within the simulation.

The Excel file is separated into multiple sheets to stay organized. Each sheet has a specific purpose so the data can be managed efficiently. The first Excel sheet is where the user

pasters their data that is provided by the simulation. This includes data for All Whiskers (Figure 40), Bridged Whiskers (Figure 41), Critical Bridged Whiskers (Figure 42), and Simulation Inputs (Figure 43). Each of these sections include relevant statistics such as whisker numbers, whisker dimensions, resistance values, and iteration numbers. The Bridged Whisker and Critical Bridged Whiskers section also includes which conductors were a part of a bridge as well as the bridged pair. By placing the data into a single sheet and separating them into their own individual tables, users can easily manage, organize, and filter the outputs in a single place. This is the foundation of the Excel file and will aid in further analysis as well as making it so the user can easily add any analysis method they wish.

All Whiskers					
Whisker #	Length (um)	Width (um)	Resistance (ohm)	Iteration	
1	2921.423	57.96461	0.1206716	1	
2	5753.229	72.59191	0.1515206	1	
3	6390.803	52.62613	0.32025	1	
4	2826.367	45.6218	0.1884605	1	
5	2061.689	54.36802	0.09679941	1	
6	2657.678	62.38023	0.09478619	1	
7	1675.75	51.51289	0.08764234	1	
8	5369.641	60.25764	0.2052379	1	
9	6449.581	33.88324	0.7796476	1	
10	2589.204	66.38473	0.0815392	1	
11	2007.121	60.37473	0.0764187	1	
12	2974.402	32.73833	0.3851441	1	
13	1055.391	54.58115	0.04916594	1	
14	1913.361	55.89381	0.08499748	1	

Figure 40: List of All Whiskers in Excel

Bridged Whiskers							
Whisker #	Length (um)	Width (um)	Resistance (ohm)	Iteration	Conductor 1	Conductor 2	Bridged Pair
318	3272.759	51.13706	0.1736917	1 GND	Q1		GND + Q1
352	4102.102	55.02091	0.1880561	1 GND	Q1		GND + Q1
26	4421.396	53.90409	0.2111799	1 Q1	GND		Q1 + GND
43	3747.569	56.62005	0.1622355	1 R28	NetD3_1		R28 + NetD3_1
56	2372.752	45.85566	0.1579791	1 C4	NetC4_1		C4 + NetC4_1
198	6977.418	59.15441	0.2767306	1 Q2	NetQ2_2		Q2 + NetQ2_2
117	6817.568	51.01569	0.3635453	1 GND	NetQ1_1		GND + NetQ1_1
9	6449.581	33.88324	0.7796476	1 NetR11_2	C2		NetR11_2 + C2
381	2518.593	63.03785	0.08796133	1 R30	NetR29_2		R30 + NetR29_2
368	1159.62	62.50555	0.04119219	1 R11	NetR11_2		R11 + NetR11_2
66	2761.829	66.48866	0.08670381	1 Q1	NetQ1_2		Q1 + NetQ1_2
274	3592.661	43.09268	0.268501	1 NetR11_2	R15		NetR11_2 + R15
260	3992.121	59.58002	0.156077	1 NetR7_2	NetC1_1		NetR7_2 + NetC1_1
240	2707.194	62.07954	0.09748974	1 R9	NetR8_2		R9 + NetR8_2

Figure 41: List of Bridged Whisker in Excel

Critical Bridged Whiskers							
Whisker #	Length (um)	Width (um)	Resistance (ohm)	Iteration	Conductor 1	Conductor 2	Bridged Pair
120	3336.264	91.95851	0.05475366	4 R35	R36		R35 + R36
112	2761.398	60.62378	0.1042749	9 R36	R35		R36 + R35
167	6544.978	59.42219	0.2572453	14 R35	R36		R35 + R36
206	5232.003	46.96705	0.3291683	15 R35	R36		R35 + R36

Figure 42: List of Critical Bridged Whiskers in Excel

Paste Inputs Below	
Simulation Inputs	
Parameter	Value
Length Mu	6
Length Sigma	1.15
Width Mu	1
Width Sigma	0.67
# Of Whiskers	200
Conductive Material Selection	copper
Whisker Material	Tin
Distribution Selection	LogNormal
Whisker Spawn Point X	0
Whisker Spawn Point Y	0
Whisker Spawn Point Z	0
X-Coord	3
Y-Coord	1
Z-Coord	3
Gravity	Earth
# of Iterations	15
Directory Path	C:\test
Save File Name	DemoDataLog
Rotation-X	-90
Shock Active	N
Vibration Active	N

Figure 43: List of Simulation Inputs

The next sheet is called General Analysis. This sheet contains important analysis features, the first being the probability section (Figure 44). The probability section is divided into two

subsections. The first being Bridged Probability, which is based on the Bridged Whisker table in the Data sheet. The second is Critical Bridged Probability, which is based on the Critical Bridged Whisker table in the Data sheet. Each of these sections contain both Overall Probability and Individual Probability calculations. The Overall Probability is the percent of iterations that have at least one bridge over all iterations that were ran in the simulation. For example, if 10 total iterations ran, and 6 iterations had at least one bridge occur, there is a 60% chance of bridging. The Individual Probability is the percent of whiskers that bridged over the total number of spawned whiskers for each individual iteration. For example, in one iteration, if 5 whiskers bridged out of 100 generated whiskers, the probability of bridging for that specific iteration is 5%.

The General Analysis sheet also contains a Summary Statistics section to help streamline the analytical process (Figure 45). This section shows general statistics such as the average, maximum, and minimum values for whisker length, width, and resistance. These statistics are calculated for each category, including All Whiskers, Bridged Whiskers, and Critical Bridged Whiskers. It also includes additional information about the number of iterations with zero bridges and the number of iterations with at least one bridge. This information is given for the Bridged Whiskers, and Critical Bridged Whisker sections.

Bridged Probability*		Critical Bridged Probability*	
Overall Probability*		Overall Probability*	
100.00%		13.33%	
Individual Probability*		Individual Probability*	
Iteration #	Probability	Iteration #	Probability
1	8.00%	4	0.25%
2	6.25%	9	0.25%
3	8.25%	14	0.25%
4	5.50%	15	0.25%
5	7.50%		
6	6.75%		

Figure 44: List of Bridge Probability and Critical Bridge Probability for Each Iteration

Summary Statistics*			
	All Whiskers	Bridged Whiskers	Critical Bridged Whiskers
Average Length (um)	3386.3	4190.6	4468.7
Average Width (um)	55.8	56.3	64.7
Average Resistance (ohm)	0.2	0.2	0.2
Maximum Length (um)	22522.4	22522.4	6545.0
Maximum Width (um)	112.9	99.8	92.0
Maximum Resistance (ohm)	1.5	1.4	0.3
Minimum Length (um)	426.7	1030.2	2761.4
Minimum Width (um)	24.9	28.1	47.0
Minimum Resistane (um)	0.0118	0.0217	0.0548
		Bridged Whiskers	Critical Bridged whiskers
# of Iterations with 0 Bridges		0	26
# of Iterations with at least 1 Bridge		30	4

Figure 45: List of Summary Statistics

There are another two sheets in this Excel File: Bridged Whisker Analysis and Critical Bridged Whisker Analysis. While these sheets are separate, they both use the same type of

analysis (Figure 46). The first section on both sheets is a Bridged Pair Analysis table which gives a breakdown on specific conductor pairs that have bridged. The table has columns for Bridged Pairs, Count of Bridged Pair, and Distinct Count of Iteration. This gives the user an idea of how often specific conductor pairs are involved in a bridge and how many iterations they appear in. Next to this table are Excel slicers which will allow the user to filter the data to focus on specific conductors and iterations. There is also a plot within this section which uses a combination chart with a bar graph that represents the Count of Bridged pair and a line that represents the Distinct Count of Iteration (Figure 47). This makes it so the user can visualize both frequency and iteration distribution.

Bridged Pair Analysis				Filters	
Bridged Pairs	Count of Bridged Pair	Distinct Count of Iteration		Conductor 1	
Q1 + GND	12	66.67%		C1	
C4 + NetC4_1	4	26.67%		C2	
GND + NetQ1_1	4	20.00%		C3	
C3 + NetC3_2	3	20.00%		C4	
C5 + NetC5_1	3	20.00%		C5	
R39 + GND	2	13.33%		C6	
R16 + NetR8_1	2	13.33%		D2	
NetL2_1 + NetD3_1	2	13.33%		GND	
C5 + NetC5_2	2	13.33%			
R6 + NetC1_2	2	13.33%		Conductor 2	
C6 + NetC6_2	2	13.33%		C5	
Q2 + NetQ2_2	2	13.33%		GND	
R19 + NetD2_1	2	13.33%		NetC1_1	
C3 + NetC3_1	2	13.33%		NetC1_2	
GND + Q1	2	13.33%		NetC2_1	
R29 + NetL2_2	2	6.67%		NetC2_2	
Q2 + R5	1	6.67%		NetC3_1	
NetC5_1 + C5	1	6.67%		NetC3_2	
R16 + NetR16_2	1	6.67%		Iteration	
C4 + NetC4_2	1	6.67%		9	
Q2 + NetQ2_1	1	6.67%		10	
NetQ2_1 + R2	1	6.67%		11	
R12 + NetR12_1	1	6.67%		12	
R28 + NetD3_1	1	6.67%		13	
L2 + NetL2_1	1	6.67%		14	
R3 + NetR3_2	1	6.67%		15	
Q1 + NetQ1_1	1	6.67%		(blank)	
R35 + NetD4_1	1	6.67%			
GND + NetR38_2	1	6.67%			
D2 + NetD2_1	1	6.67%			
R11 + NetC2_1	1	6.67%			
D2 + NetD2_2	1	6.67%			
R15 + NetR15_2	1	6.67%			
R8 + NetR8_1	1	6.67%			
C1 + NetC1_2	1	6.67%			
NetR29_2 + R30	1	6.67%			
R2 + NetQ2_1	1	6.67%			
NetR9_2 + R10	1	6.67%			
C2 + NetC2_2	1	6.67%			
NetR11_2 + R11	1	6.67%			
R30 + NetR29_2	1	6.67%			

Figure 46: Bridged Pair Analysis and Filters

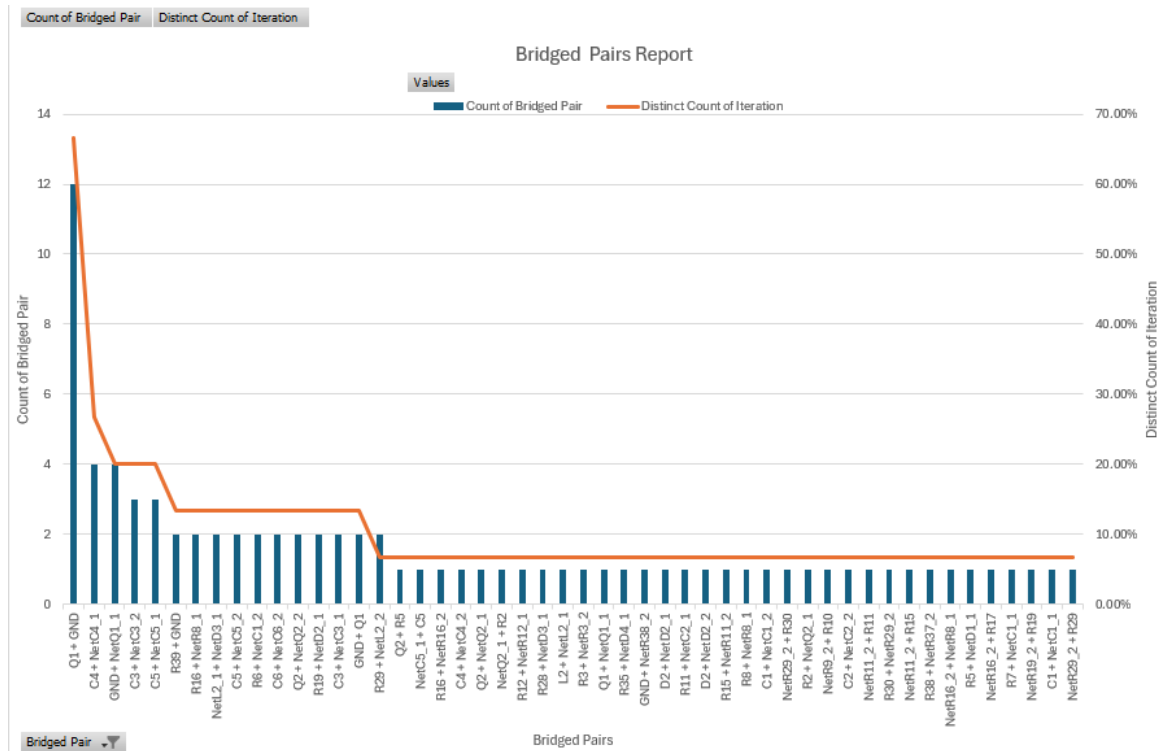


Figure 47: Bridged Pair Plot

The next section on both the Bridged Whisker Analysis and Critical Bridged Whisker Analysis sheet is the Bridged Count Distribution (Figure 48). In this section, the table has two columns, the first showing the number of bridges and the other showing the number of iterations. This table shows the distribution of the bridge counts across all iterations. This gives the user an idea of the frequency at which these bridges occur. Similar to the previous section, there are slicers that will allow the user to filter the data to their liking. The plot in this section is there to visually represent the distribution of bridge counts (Figure 49). These tools will help the user better understand the behavior of whiskers on their CCA and identify patterns in bridging.

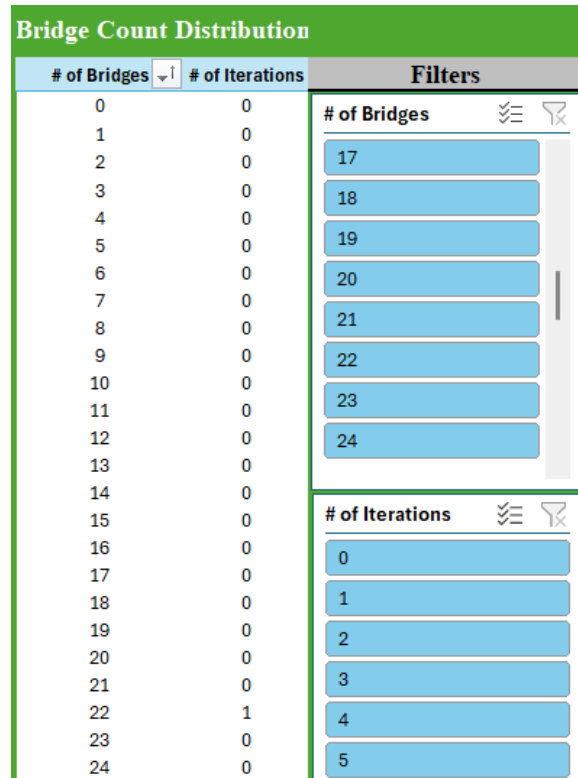


Figure 48: Bridge Count Distribution

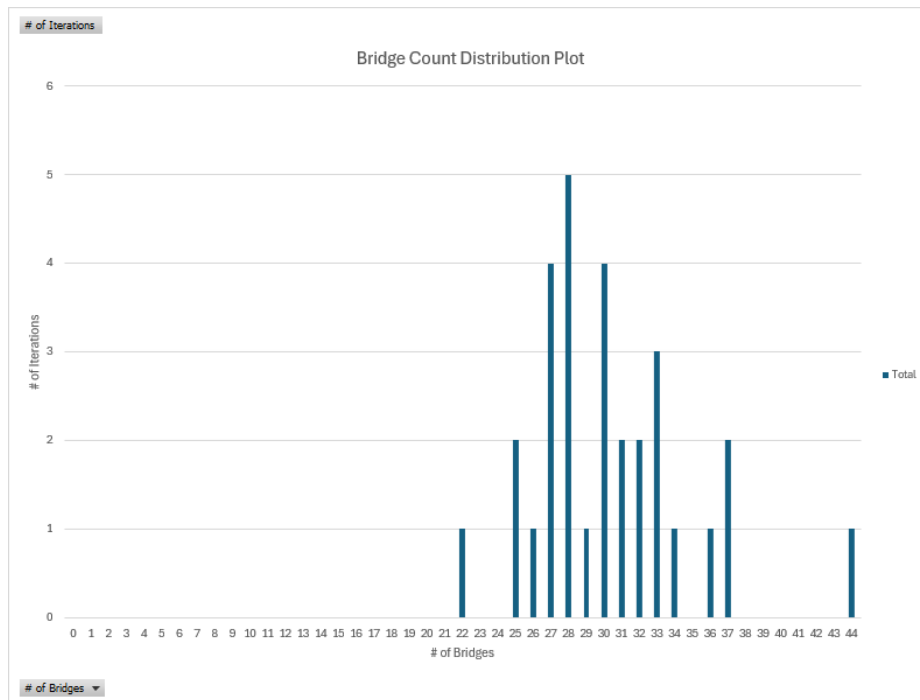


Figure 49: Bridge Count Distribution Plot

Throughout this Excel file there are built-in functions that can further help the user understand how certain sections work and how certain results are calculated. Headers that are marked with an asterisk (*) can be selected to show additional information via built-in tooltips. There is also a helper sheet within this Excel file that helps calculate data within all of the sheets. While it should not be modified, it can help the user gather a better understanding of the formulas used within Excel.

Further Development

Though the project has been thoroughly looked through and tested, there are bound to be errors within the simulation and known issues that could not be fixed within the projects' allotted time span. This section is for discussing improvements that could be made in the future, known issues, and different tasks that should be tackled by the following team to improve the simulation. Functionality and ease of use was the focus of this team's work; however, some functionality was not accomplished and should be looked at further. The main functionality that was not fully achieved was: a screenshot file name and directory input, a screen recording of the simulation, an input for changing the duration of iterations, implementing/uploading of a new PCB into the simulation, and a continuation of a STIG for any code changes.

As mentioned previously, a camera and orbit functionality were added to the simulation; however, a file designation name and directory were not added. This functionality would be helpful to easily save screenshot images of the simulation within a designated directory of the user's choosing. The screenshot code is already capable of naming new screenshots sequentially; however, it currently has a default file name and stores directly to the desktop of the user, rather than a specific folder. This directory would allow for ease of use by storing the screenshot files

within an organized designated directory under a desired file name rather than on the user's desktop.

The screen capture functionality was explored, but no good implementation method was found. The first method explored was finding an in-game Unity feature that allowed for screen capture, but none was found in our Personal Unity License. The second method that was explored involved hard coding, a function that took a screenshot of the simulation every millisecond, then combine each image taken to compile it into a video. This method ended up being extremely processor heavy and inefficient for the simulation, causing it to slow or crash.

Another function that would be great to have for the user would be a new button or text field that allows the user to change how long each iteration runs for. This should be a simple fix, but this was a low priority issue for this project. It may be possible to implement this feature by going to "UIscript.cs" and adding a function to change its duration or creating a separate script for it.

The PCB upload functionality would be the next best function to figure out as it opens the doors to other CCA's the user can test with the simulation. The main issues found reside within the programs themselves. Specifically, Altium did not download into a file version that can be uploaded to Blender without going through an online file converter, and Altium would not save the pad and trace data of a CCA, no matter which file type was used (.step, .obj, .fbx etc.). This made it so that when a board was uploaded to another program, either to a software such as Blender to clean up geometric data, or straight into Unity itself, the CCA did not have pads or traces identifiable as objects for the simulation to detect. The furthest step found to accomplish this task was through a .step file potentially storing the pads and traces as 2D information and then writing a script to read this and extrude it into a 3D form. Additionally, a limitation was

discovered within Unity and .mtl files. A BoM or .mtl file that was uploaded into Unity could not be read by the Unity program in the same way Unity assign material information. The only known way, currently, to assign materials in Unity is by hand through a .mtl file or BoM created within Unity itself. All these issues proved to be too tedious and time-consuming, and the PCB implementation was abandoned as a result.

The STIG is a software security list of governmental code compliance to stop cyberattacks or online threats through the code generated. Team 2 created a STIG using the Stig Viewer 3 software and a Source Code Analysis using the SonarQube software. Any future team that updates this code should complete a STIG and Source Code Analysis for the sponsors using the altered code.

CHAPTER 8 CONCLUSION

This semester Auburn Metal Whiskers Team 2 designed a refinement to the 3D simulation created by Auburn Metal Whiskers Team 1 during the previous semester. Both the Missile Defense Agency and NASA agreed on a list of design specifications that were to be implemented in the 3D simulation. The design specification consisted of 3D Simulation Inputs and Functionality, Identification of Nodes and Whiskers, Stored and Displayed Information, Beta Testing, 3D Simulation Inputs and Functionality, Identification of Nodes and Whiskers. The completion of these tasks was due to a better understanding of the different programs that were needed to complete the 3D simulation after a successful 2D simulation last semester. This was important due to constraints from the customer, who requested that all PCBs be created in Altium and that all simulations be made in Unity Game Engine.

Members of Auburn Metal Whiskers Team 2 have learned how to use C# functions that allow the user to input different desired data and design PCBs in the Altium PCB designer so it can be uploaded to Blender and finally Unity. Throughout the semester Auburn Metal Whiskers Team 2, the customer, and technical advisors have met on multiple conference calls to allow for a shared understanding of what Team 2 has accomplished in the 3D simulation and what design specifications will be met and completed each week. Team 2 faced a couple of different obstacles during the semester. One of those problems was the limitations of importing between Blender and Altium in which the two programs had different files they exported and imported resulting in file formatting complications. To combat this, multiple third-party CAD and PCB exporting programs were used. The other problem faced was each of the two leads had a unique approach to achieving the desired end goal. This required clear communication and collaboration

to develop diverse solutions that aligned with both the customer's objectives and the Tech Advisor's expectations for the project. However, even with these problems Team 2 was not only able to overcome them but also meet the design specifications that were labeled at the start of the semester.

APPENDIX A

Programmatic Questions

- 1. Are there any professional and ethical responsibilities that came up in your project (if any), and what your team did to address them and why.**

There were really two professional and ethical responsibilities that the team faced during the project. The first problem of this project was to ensure the team delivered a product that was precise and user-friendly. It is crucial for the user to accurately assess the risk of a metal whisker causing a bridge, as such a bridge could lead to potentially catastrophic consequences. The second problem was having two different leads that had two different approaches to the desired end goal. This meant that the team had to accurately communicate and come up with diverse solutions that met the main goals of both what the customer and what the Tech Advisor wanted from the project.

- 2. What are the economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability constraints related to your design, and how your team chose to address them.**

Since our project is completely software based there wasn't any the economic, environmental, social, political, ethical, health and safety, manufacturability, and sustainability constraints that directly were influenced by our project. However, the reason our project was created was due to the fact that lead soldering was affecting the environment. Which led to the use of tin solder which typically creates metal whiskers.

3. Lifelong learning is important. What are the new areas and topics that your team had to learn in order to complete your project? What methods did you use to become educated on the things you did not know before?

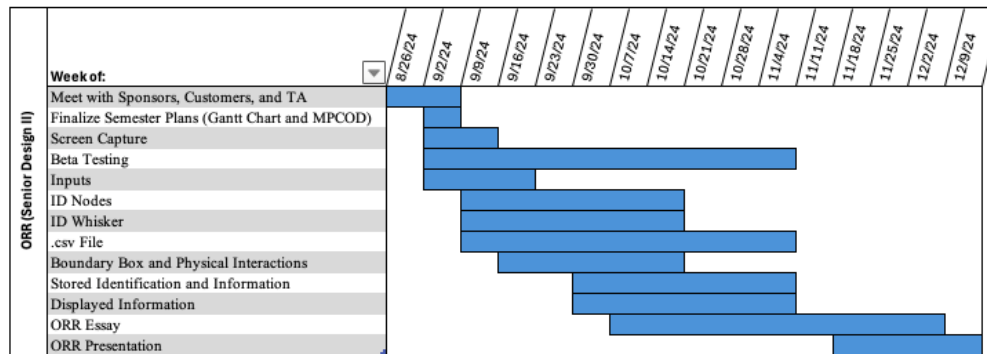
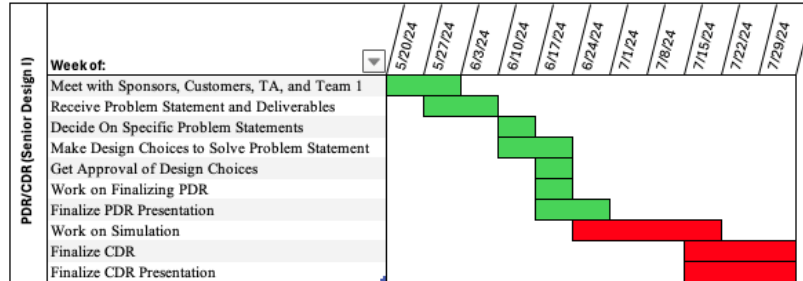
The new areas of that different team's members learn were the software that was required to accomplish the project. Three team members made up the unity team they learned about C# functions and more specifically how to basically build a game in a game engine like unity. Two team members made up the Altium and Blender team where they learned how to create realistic PCB schematics as well as import them into Blender to define geometry. This demonstrates one of the main methods where our team was broken down into different teams. This allowed for team members to focus on specific areas and become experts in there given fields.

4. What are the contemporary issues related to your project and design?

It was essential for the user to reliably assess the risk of a metal whisker causing a bridge, as such a bridge could result in potentially catastrophic outcomes. The project was developed in response to the environmental impact of lead soldering, which led to the adoption of tin solder, a material that commonly produces metal whiskers.

APPENDIX B

Gantt Chart



APPENDIX C

BETA Website

<https://shawn-de89.github.io/3D-Metal-Whisker-Simulation-Beta/>

APPENDIX D

UIScript.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;
using System.IO;
using UnityEngine.UI;
using Unity.VisualScripting;
using System.Data.Common;
using System.Linq;

//This is the main script that controls many of the objects on the interface, namely the user
inputs.
public class UIScript : MonoBehaviour
{
    private System.Random rand = new System.Random();
    //UI elements
    public TMP_InputField lengthMu;
    public TMP_InputField widthMu;
    public TMP_InputField lengthSigma;
    public TMP_InputField widthSigma;
    public TMP_InputField numWhiskers;
    public TMP_InputField xCoord;
    public TMP_InputField yCoord;
    public TMP_InputField zCoord;
    public TMP_InputField numIterations;
    public TMP_InputField shockAmp;
    public TMP_InputField shockFreq;
    public TMP_InputField shockDur;
    public TMP_InputField vibrAmp;
    public TMP_InputField vibrFreq;
    public TMP_InputField vibrDur;
    public TMP_InputField material_input;
    public TMP_InputField WhiskerSpawnPointX;
    public TMP_InputField WhiskerSpawnPointY;
    public TMP_InputField WhiskerSpawnPointZ;
    public TMP_InputField totalRuns;
    public TMP_InputField CircuitRotateX;
    public TMP_InputField CircuitRotateY;
    public TMP_InputField CircuitRotateZ;
    public TMP_InputField SpinRateX;
    public TMP_InputField SpinRateY;
    public TMP_InputField SpinRateZ;
```

```

public TMP_InputField wallHeightInput;
public TMP_Dropdown whiskMat;
public TMP_Dropdown distributionDropdown;
public Toggle isShockActiveToggle;
public Toggle isVibrationActiveToggle;

public GameObject whisker;
public TextMeshProUGUI totalBridges;
public int bridgesDetected;
public TextMeshProUGUI bridgesEachRun;
public int bridgesPerRun;
public TextMeshProUGUI errorMessage;
public float simtimeElapsed;
public bool startSim = false;
public int simIntComplete = 1;
public float simTimeThresh;
public float moveSpeed = 5f;
public DistributionType distributionType = DistributionType.Lognormal;
public bool UIisOn = true;
public GameObject UIui;
public bool GridIsOn = true;
public GameObject grid;
public TextMeshProUGUI iterationCounter;
public TextMeshProUGUI iterationCompleteMessage;
public TriggerControl triggerControl;
public Dictionary<MaterialType, MaterialProperties> materialProperties = new
Dictionary<MaterialType, MaterialProperties>()
{ //density (kg/um^3), resistivity (ohm*um), coefficient of friction (unitless)
  { MaterialType.Tin, new MaterialProperties(7.3e-15f, 1.09e-1f, 0.32f) },
  { MaterialType.Zinc, new MaterialProperties(7.14e-15f, 5.9e-2f, 0.6f) },
  { MaterialType.Cadmium, new MaterialProperties(8.65e-15f, 7.0e-2f, 0.5f) }
};
public MaterialType currentMaterial = MaterialType.Tin;
private List<float> lengths;
private List<float> widths;
private List<float> volumes;
private List<float> masses;
private List<float> resistances;
public WhiskerControl whiskerControl; //new
public bool isVibrationActive = false;
public bool isShockActive = false;
public bool RotateSpinToggle = false;
public VibrationManager vibrationManager;
public ShockManager shockManager;
public float shockPressTimer = 0f;
public float shockPressInterval = 2f;

```

```

    public SimulationController simulationController; //reference to the simulation controller
script
    private int whiskerCounter; //variable to track whisker numbers

    public ScreenshotHandler screenshotManager; // Reference to the Screenshot script

    //references for critical pair UI
    public TMP_Dropdown conductor1Dropdown;
    public TMP_Dropdown conductor2Dropdown;
    public Button addPairButton;
    public Button removePairButton;
    public ScrollRect criticalPairsScrollView;
    public GameObject criticalPairItemPrefab; //prefab for list items
    public HashSet<string> criticalPairs = new HashSet<string>(); //data structure to store crit
pairs
    private List<GameObject> criticalPairItems = new List<GameObject>(); // list to keep track
of UI items
    public Button refreshDropdownsButton;

    private float CircuitRotateXValue;
    private float CircuitRotateYValue;
    private float CircuitRotateZValue;
    private GameObject circuitBoard;
    public WallCreator wallCreator;
    public CircuitBoardFinder circuitBoardFinder;

    private DataManager dataManager;
    private bool dataWritten = false;

    //Sets the lists for the dimensions/data to be stored in, as well as sets material properties from
the dropdown.
    void Start()
    {
        whiskerCounter = 1; //initialize the counter to 1 when the script starts

        lengths = new List<float>();
        widths = new List<float>();
        volumes = new List<float>();
        masses = new List<float>();
        resistances = new List<float>();

        whiskMat.onValueChanged.AddListener(delegate
        {
            WhiskMatDropdownValueChanged(whiskMat);

```

```

});

UpdateMaterialPropertiesUI(currentMaterial);

Time.fixedDeltaTime = 0.005f;
Physics.defaultSolverIterations = 10;
Physics.defaultSolverVelocityIterations = 10;

StartCoroutine(DelayedPopulateConductorDropdowns()); //delay to call code

addPairButton.onClick.AddListener(OnAddPairButtonClicked);
removePairButton.onClick.AddListener(OnRemovePairButtonClicked);
refreshDropdownsButton.onClick.AddListener(OnRefreshButtonClicked);

circuitBoard = GameObject.FindGameObjectWithTag("CircuitBoard");

dataManager = DataManager.Instance;
if (dataManager == null)
{
    Debug.LogError("dataManager instance not found.");
}

if (whiskerControl == null)
{
    whiskerControl = FindObjectOfType<WhiskerControl>();
    if (whiskerControl == null)
    {
        Debug.LogError("WhiskerControl not found in the scene.");
    }
}
}

//Controls the dropdown for material selection.
public void UpdateMaterialPropertiesUI(MaterialType materialType)
{
    switch (materialType)
    {
        case MaterialType.Tin:
            whiskMat.captionText.text = "Tin";
            break;
        case MaterialType.Zinc:
            whiskMat.captionText.text = "Zinc";
            break;
        case MaterialType.Cadmium:
            whiskMat.captionText.text = "Cadmium";
            break;
    }
}

```



```

        default:
            break;
    }

    Debug.Log($"UI updated for material: {materialType}");
}

//Method to handle whiskMat dropdown value change
void WhiskMatDropdownValueChanged(TMP_Dropdown change)
{
    currentMaterial = (MaterialType)change.value;
    UpdateMaterialPropertiesUI(currentMaterial);

    StartCoroutine(CloseDropdownAfterDelay());
}

//Coroutine to close dropdown after a short delay
private IEnumerator CloseDropdownAfterDelay()
{
    yield return null;
    TMP_Dropdown dropdown = whiskMat.GetComponent<TMP_Dropdown>();
    dropdown.Hide();
}

//Controls the iteration counters/bridge counters/applies shock/vibration throughout simulation
void Update()
{
    totalBridges.text = "Total Bridges: " + bridgesDetected.ToString(); //remove if needed.
    bridgesEachRun.text = "Bridges for Current Run: " + bridgesPerRun.ToString();

    float horizontalInput = Input.GetAxis("Horizontal");
    float verticalInput = Input.GetAxis("Vertical");

    Vector3 moveDirection = new Vector3(horizontalInput, 0f, verticalInput).normalized;
    transform.Translate(moveDirection * moveSpeed * Time.deltaTime);

    if (startSim)
    {
        CircuitBoardFinder circuitBoardFinder = FindObjectOfType<CircuitBoardFinder>();

        if (isShockActive && shockManager.shockButton.interactable)
        {
            shockPressTimer += Time.deltaTime;
            if (shockPressTimer >= shockPressInterval)
            {
                shockManager.shockPressed();
            }
        }
    }
}

```

```

        shockPressTimer = 0f;
    }
}

if (isVibrationActive && vibrationManager.vibrateButton.interactable)
{
    vibrationManager.vibratePressed();
}

simtimeElapsed += Time.deltaTime;
if (simIntComplete <= Convert.ToInt32(totalRuns.text) - 1)
{
    iterationCounter.text = "Iteration Counter: " + simIntComplete.ToString();
    if (simtimeElapsed > simTimeThresh)
    {
        simIntComplete++;
        ReloadWhiskersButton();
        simtimeElapsed = 0f;

        if (screenshotManager != null)
        {
            screenshotManager.OnIterationEnd();
        }
        if (circuitBoardFinder != null)
        {
            // Toggle the RotateSpinToggle
            circuitBoardFinder.RotateSpinToggle.isOn =
!circuitBoardFinder.RotateSpinToggle.isOn;
        }
    }
}
else if (simIntComplete == Convert.ToInt32(totalRuns.text))
{
    iterationCounter.text = "Iteration Counter: " + simIntComplete.ToString();
    if (simtimeElapsed > simTimeThresh)
    {
        if (screenshotManager != null)
        {
            screenshotManager.OnIterationEnd();
        }
        iterationCompleteMessage.text = "Simulation Complete!";
        startSim = false;

        if (!dataWritten)
        {
            WriteDataToCSV();
        }
    }
}

```

```

        dataWritten = true; // Prevents multiple writes
    }
}
}
}

//Resets the simulation counters back to original value
public void resetSim()
{
    simIntComplete = 1;
    simtimeElapsed = 0;
    bridgesDetected = 0;
    bridgesPerRun = 0;
    iterationCompleteMessage.text = "";
    iterationCounter.text = "";
    startSim = false;
}

//Takes in Mu and Sigma values for Length to generate values.
public float LengthDistributionGenerate()
{
    float mu = float.Parse(lengthMu.text);
    float sigma = float.Parse(lengthSigma.text);
    float lengthVal;

    if (distributionType == DistributionType.Lognormal)
    {
        lengthVal = GenerateLogNormalValue(mu, sigma);
    }
    else
    {
        lengthVal = GenerateNormalValue(mu, sigma);
    }

    return lengthVal;
}

//Takes in Mu and Sigma values for Width to generate values.
public float WidthDistributionGenerate()
{
    float mu_log = float.Parse(widthMu.text);
    float sigma_log = float.Parse(widthSigma.text);
    float widthVal;

```

```

    if (distributionType == DistributionType.Lognormal)
    {
        widthVal = GenerateLogNormalValue(mu_log, sigma_log);
    }
    else
    {
        widthVal = GenerateNormalValue(mu_log, sigma_log);
    }

    return widthVal;
}

//Generates a lognormal value based off mu/sigma inputs.
private float GenerateLogNormalValue(float mu_log, float sigma_log)
{
    float normalVal = RandomFromDistribution.RandomNormalDistribution(mu_log,
sigma_log);
    float logNormalVal = Mathf.Exp(normalVal);
    return logNormalVal;
}

//Generates a normal value based off mu/sigma inputs.
private float GenerateNormalValue(float mu_norm, float sigma_norm)
{
    return RandomFromDistribution.RandomNormalDistribution(mu_norm, sigma_norm);
}

//Handles many error messages and generates whiskers
public void MakeWhiskerButton()
{
    //error handling | Limits are subject to change
    float mu_log = float.Parse(widthMu.text);
    float sigma_log = float.Parse(widthSigma.text);
    float mu = float.Parse(lengthMu.text);
    float sigma = float.Parse(lengthSigma.text);
    float numWhiskersToCreate = float.Parse(numWhiskers.text);
    float numberIterations = float.Parse(totalRuns.text);

    float x = Convert.ToSingle(xCoord.text);
    float y = Convert.ToSingle(yCoord.text);
    float z = Convert.ToSingle(zCoord.text);

    if (distributionType == DistributionType.Lognormal)
    {
        if (mu_log > 9 || mu_log < 0)

```

```

{
    SetErrorMessage("Width Mu value outside acceptable range.");
    return;
}
if (sigma_log > 4 || sigma_log < 0)
{
    SetErrorMessage("Width Sigma value is outside acceptable range.");
    return;
}
if (mu > 20 || mu < 0)
{
    SetErrorMessage("Length Mu value outside acceptable range.");
    return;
}
if (sigma > 15 || sigma < 0)
{
    SetErrorMessage("Length Sigma value is outside acceptable range.");
    return;
}
}
else
{
    if (mu_log > 10000 || mu_log < 0)
    {
        SetErrorMessage("Width Mu value outside acceptable range.");
        return;
    }
    if (sigma_log > 10000 || sigma_log < 0)
    {
        SetErrorMessage("Width Sigma value is outside acceptable range.");
        return;
    }
    if (mu > 30000 || mu < 0)
    {
        SetErrorMessage("Length Mu value outside acceptable range.");
        return;
    }
    if (sigma > 10000 || sigma < 0)
    {
        SetErrorMessage("Length Sigma value is outside acceptable range.");
        return;
    }
}

if (numWhiskersToCreate > 2000 || numWhiskersToCreate < 0 ||
!Mathf.Approximately(numWhiskersToCreate, Mathf.Round(numWhiskersToCreate)))

```

```

    {
        SetErrorMessage("Whisker count too high or invalid. Limit is 2000 and must be a
positive integer.");
        return;
    }

    if (x < 0 || y < 0 || z < 0)
    {
        SetErrorMessage("Coordinates cannot be negative.");
        return;
    }

    if (numberIterations < 0 || !Mathf.Approximately(numberIterations,
Mathf.Round(numberIterations)))
    {
        SetErrorMessage("Iteration value must be a positive integer.");
        return;
    }

    lengths.Clear();
    widths.Clear();
    volumes.Clear();
    masses.Clear();
    resistances.Clear();

    GameObject whiskerSpawnPoint = GameObject.Find("WhiskerSpawnPoint");

    if (whiskerSpawnPoint == null)
    {
        SetErrorMessage("WhiskerSpawnPoint not found in the scene.");
        return;
    }
    // Move the WhiskerSpawnPoint to (0, 0, 0) before starting
    whiskerSpawnPoint.transform.position = Vector3.zero;

    // Ensure input fields are not null
    if (WhiskerSpawnPointX == null || WhiskerSpawnPointY == null || WhiskerSpawnPointZ
== null)
    {
        SetErrorMessage("One or more input fields are not assigned.");
        return;
    }

    // Read target position from input fields
    if (!float.TryParse(WhiskerSpawnPointX.text, out float WSPX) ||

```

```

        !float.TryParse(WhiskerSpawnPointY.text, out float WSPY) ||
        !float.TryParse(WhiskerSpawnPointZ.text, out float WSPZ))
    {
        SetErrorMessage("Invalid input for target positions.");
        return;
    }

    for (int i = 0; i < numWhiskersToCreate; i++)
    {
        //Generate dimensions and spawn position
        float diameter = WidthDistributionGenerate() / 1000;
        float length = LengthDistributionGenerate() / 1000;
        float spawnPointX = UnityEngine.Random.Range(-float.Parse(xCoord.text) * 10,
float.Parse(xCoord.text) * 10);
        float spawnPointY = UnityEngine.Random.Range(1, Convert.ToInt32(yCoord.text) *
10);
        float spawnPointZ = UnityEngine.Random.Range(-float.Parse(zCoord.text) * 10,
float.Parse(zCoord.text) * 10);

        Vector3 spawnPos = whiskerSpawnPoint.transform.position + new
Vector3(spawnPointX, spawnPointY, spawnPointZ);

        if (whisker == null)
        {
            SetErrorMessage("Whisker prefab is not assigned.");
            return;
        }

        GameObject whiskerClone = Instantiate(whisker, spawnPos,
Quaternion.Euler(UnityEngine.Random.Range(0, 360), UnityEngine.Random.Range(0, 360),
UnityEngine.Random.Range(0, 360)));
        whiskerClone.transform.SetParent(whiskerSpawnPoint.transform);
        whiskerClone.tag = "whiskerClone";

        whiskerClone.name = $"whisker_{whiskerCounter}"; //unique name for each whisker

        // Set up Rigidbody, scaling, and physics properties
        Rigidbody whiskerRigidbody = whiskerClone.GetComponent<Rigidbody>();
        if (whiskerRigidbody == null)
        {
            whiskerRigidbody = whiskerClone.AddComponent<Rigidbody>();
        }

        //Calculates properties
        MaterialProperties currentProps = materialProperties[currentMaterial];
        float volume = Mathf.PI * Mathf.Pow(diameter / 2, 2) * length;

```

```

float mass = volume * currentProps.density;
float resistance = (currentProps.resistivity * length * 1000) / (Mathf.PI *
Mathf.Pow(diameter * 1000 / 2, 2));

//Sets a minimum mass limit
if (mass < 1f)
{
    whiskerRigidbody.mass = 0.22f;
    whiskerRigidbody.drag = 20f;
    whiskerRigidbody.angularDrag = 2f;
}
else
{
    whiskerRigidbody.mass = mass;
    whiskerRigidbody.drag = 20f;
    whiskerRigidbody.angularDrag = 2f;
}

whiskerRigidbody.collisionDetectionMode = CollisionDetectionMode.Continuous;

//UPSIZING WIDTHS
Transform visual = whiskerClone.transform.Find("visual");
Transform collider = whiskerClone.transform.Find("collider");
if (diameter * 1000 < 10)
{
    visual.localScale = new Vector3(10, 1, 10); // scales relative to the parent object
} // so this is saying if the diameter is less than 50, it takes the diameter of the original
whisker and *5.
else
{
    visual.localScale = new Vector3(1.2f, 1, 1.2f);
}

collider.localScale = new Vector3(1, 1, 1);

whiskerClone.transform.localScale = new Vector3(diameter, length / 2, diameter);
lengths.Add(length);
widths.Add(diameter);

Collider whiskerCollider = whiskerClone.GetComponent<Collider>();
if (whiskerCollider == null)
{
    Debug.LogError("Whisker prefab must have a Collider component.");
    return;
}

```



```

//Gets physics material and sets its friction properties
PhysicMaterial whiskerPhysicsMaterial = whiskerCollider.sharedMaterial;
if (whiskerPhysicsMaterial == null)
{
    whiskerPhysicsMaterial = new PhysicMaterial();
    whiskerCollider.sharedMaterial = whiskerPhysicsMaterial;
}
UpdatePhysicsMaterialFriction(whiskerPhysicsMaterial);

if (whiskerControl.confirmGravity)
{
    WhiskerData data = new WhiskerData(whiskerCounter, length, diameter, volume,
mass, resistance, simIntComplete);
    SaveWhiskerData(data);
}

whiskerCounter++; //increment counter for next whisker

// Debug log for verification, currently commented out to avoid log spam
//Debug.Log($"Whisker created with material: {currentMaterial}, Density:
{currentProps.density}, Mass: {mass}, Resistance: {resistance}");
}
// Move the WhiskerSpawnPoint to the desired target position after spawning
Vector3 targetPosition = new Vector3(WSPX * 10, WSPY * 10, WSPZ * 10);
whiskerSpawnPoint.transform.position = targetPosition;
}

//Updates the friction on the whiskers
private void UpdatePhysicsMaterialFriction(PhysicMaterial material)
{
    MaterialProperties currentProps = materialProperties[currentMaterial];
    material.staticFriction = currentProps.coefficientOfFriction;
    material.dynamicFriction = currentProps.coefficientOfFriction;
}

//Controls the Reload whiskers buttons. Clears out whiskers and generates new.
public void ReloadWhiskersButton()
{
    // Clear out all whiskers
    GameObject[] whiskerClones = GameObject.FindGameObjectsWithTag("whiskerClone");
    GameObject[] bridgedWhiskers =
GameObject.FindGameObjectsWithTag("bridgedWhisker");
    GameObject[] allWhiskers = whiskerClones.Concat(bridgedWhiskers).ToArray();
    foreach (GameObject whisk in allWhiskers)
    {
        Destroy(whisk.gameObject);
    }
}

```

```

    }
    bridgesPerRun = 0;

    // Reset the whisker counter before creating new whiskers
    whiskerCounter = 1; //resets counter to 1 for each iteration
    MakeWhiskerButton();
}

//Function that sets the error message.
public void SetErrorMessage(string message)
{
    errorMessage.text = message;
    StartCoroutine(ClearErrorMessageAfterDelay(6f));
}

//Clears the error message
private IEnumerator ClearErrorMessageAfterDelay(float delay)
{
    yield return new WaitForSeconds(delay);
    errorMessage.text = "";
}

//Saving whiskers data;
public void SaveWhiskerData(WhiskerData data)
{
    DataManager.Instance.allWhiskersData.Add(data);

    Debug.Log($"Whisker data saved: {data.WhiskerNumber}");
}

public void WriteDataToCSV()
{
    // Paths for both files
    string customDirectoryPath = whiskerControl.directoryPath;
    string customFilePath = Path.Combine(customDirectoryPath, whiskerControl.fileName +
".csv");
    string fixedFilePath = Path.Combine(Application.dataPath, "bridgeOutput.csv");

    // Write data to custom file path with all data
    WriteDataToCSVFile(customFilePath, includeAllWhiskersData: true);
    //brideOutput.csv file with only bridged data
    WriteDataToCSVFile(fixedFilePath, includeAllWhiskersData: false);
}

private void WriteDataToCSVFile(string filePath, bool includeAllWhiskersData)
{

```

```

if (string.IsNullOrEmpty(filePath))
{
    SetErrorMessage("Failed to save data - File path cannot be empty");
    return;
}

try
{
    if (!Directory.Exists(Path.GetDirectoryName(filePath)))
    {
        Directory.CreateDirectory(Path.GetDirectoryName(filePath));
    }

    using (StreamWriter writer = new StreamWriter(filePath, false)) // Overwrite the file
    {
        // Get data from dataManager
        List<WhiskerData> allWhiskersData = DataManager.Instance.allWhiskersData;
        List<WhiskerData> bridgedWhiskersData =
DataManager.Instance.bridgedWhiskersData;
        List<WhiskerData> criticalBridgedWhiskersData =
DataManager.Instance.criticalBridgedWhiskersData;

        if (includeAllWhiskersData)
        {
            // Write the headers
            writer.WriteLine("All Whiskers,,,,,Bridged Whiskers,,,,,,Critical Bridged
Whiskers,,,,,,Simulation Inputs,");
            writer.WriteLine("Whisker #,Length (um),Width (um),Resistance (ohm),Iteration,"
+ ",Whisker #,Length (um),Diameter (um),Resistance (ohm),Iteration,Conductor
1,Conductor 2,"
+ ",Whisker #,Length (um),Diameter (um),Resistance (ohm),Iteration,Conductor
1,Conductor 2,"
+ ",Parameter,Value");

            //collect sim inputs
            List<KeyValuePair<string, string>> simulationInputs = GetSimulationInputs();

            // Determine the maximum number of rows
            int maxRows = Mathf.Max(allWhiskersData.Count, bridgedWhiskersData.Count,
criticalBridgedWhiskersData.Count, simulationInputs.Count);

            string gap = ",,";
            string gap1 = ",,";
            string gap3 = ",,";

```

```

// Loop through all rows
for (int i = 0; i < maxRows; i++)
{
    string allWhiskerDataLine = "";
    string bridgedWhiskerDataLine = "";
    string criticalBridgedWhiskerDataLine = "";
    string simulationInputLine = "";

    bool hasData = false;

    // Get All Whisker data if available
    if (i < allWhiskersData.Count)
    {
        var whisker = allWhiskersData[i];
        allWhiskerDataLine = $"{whisker.WhiskerNumber},{whisker.Length *
1000},{whisker.Width * 1000},{whisker.Resistance},{whisker.Iteration}";
        hasData = true;
    }
    else
    {
        allWhiskerDataLine = ".,.,.";
    }

    // Get Bridged Whisker data if available
    if (i < bridgedWhiskersData.Count)
    {
        var bridgedWhisker = bridgedWhiskersData[i];
        bridgedWhiskerDataLine =
"${bridgedWhisker.WhiskerNumber},{bridgedWhisker.Length},{bridgedWhisker.Diameter},{b
ridgedWhisker.Resistance},{bridgedWhisker.SimulationIndex},{bridgedWhisker.Conductor1},{
bridgedWhisker.Conductor2}";
        hasData = true;
    }
    else
    {
        bridgedWhiskerDataLine = ".,.,.,.";
    }

    // Get Critical Bridged Whisker data if available
    if (i < criticalBridgedWhiskersData.Count)
    {
        var criticalWhisker = criticalBridgedWhiskersData[i];
        criticalBridgedWhiskerDataLine =
"${criticalWhisker.WhiskerNumber},{criticalWhisker.Length},{criticalWhisker.Diameter},{crit
icalWhisker.Resistance},{criticalWhisker.SimulationIndex},{criticalWhisker.Conductor1},{criti
calWhisker.Conductor2}";
    }
}

```

```

        hasData = true;
    }
    else
    {
        criticalBridgedWhiskerDataLine = ",,,,,";
    }

    // Get Simulation Input data if available
    if (i < simulationInputs.Count)
    {
        var input = simulationInputs[i];
        if (!string.IsNullOrEmpty(input.Key) &&
!string.IsNullOrEmpty(input.Value))
        {
            simulationInputLine = $"{input.Key},{input.Value}";
            hasData = true;
        }
        else
        {
            simulationInputLine = ",";
        }
    }
    else
    {
        simulationInputLine = ",";
    }

    // Add debug statements to trace values
    Debug.Log($"Row {i + 1}:");
    Debug.Log($" AllWhiskerDataLine: '{allWhiskerDataLine}'");
    Debug.Log($" BridgedWhiskerDataLine: '{bridgedWhiskerDataLine}'");
    Debug.Log($" CriticalBridgedWhiskerDataLine:
'{criticalBridgedWhiskerDataLine}'");
    Debug.Log($" SimulationInputLine: '{simulationInputLine}'");
    Debug.Log($" hasData: {hasData}");

    //skip writing line if all data sections are empty
    if (!hasData)
    {
        Debug.Log(" Skipping row because hasData is false.");
        continue;
    }

    // Write the combined data to the CSV file

```

```

        string combinedLine =
        $"{allWhiskerDataLine}{gap1}{bridgedWhiskerDataLine}{gap}{criticalBridgedWhiskerDataL
ine}{gap3}{simulationInputLine}";
        writer.WriteLine(combinedLine);

    }
}
else
{
    // Only write the Bridged Whiskers section
    writer.WriteLine("Bridged Whiskers");
    writer.WriteLine("Whisker #,Length (um),Diameter (um),Resistance
(ohm),Iteration,Conductor 1,Conductor 2");

    foreach (var bridgedWhisker in bridgedWhiskersData)
    {
        string bridgedWhiskerDataLine =
        $"{bridgedWhisker.WhiskerNumber},{bridgedWhisker.Length},{bridgedWhisker.Diameter},{b
ridgedWhisker.Resistance},{bridgedWhisker.SimulationIndex},{bridgedWhisker.Conductor1},{
bridgedWhisker.Conductor2}";
        writer.WriteLine(bridgedWhiskerDataLine);
    }
}
}
Debug.Log($"Data saved successfully to {filePath}");
}
catch (Exception ex)
{
    Debug.LogError($"Failed to save data to {filePath}: {ex.Message}");
}

Debug.Log($"Number of bridged whiskers:
{DataManager.Instance.bridgedWhiskersData.Count}");

}

public TextMeshProUGUI bridgesCount;
public Text conductorName;

//Updates bridge counter
public void UpdateConductorBridge(string conductorName, int pingCount)
{
    bridgesCount.text = conductorName + ": " + pingCount; // prints UI instead of component
name. Correct ping count though.
}

```

//Hides or Shows the UI.

```
public void toggleUI()
{
    if (UIisOn)
    {
        UGui.SetActive(false);
        UIisOn = false;
    }
    else
    {
        UGui.SetActive(!UGui.activeSelf);
        UIisOn = true;
    }
}
```

//Hides or Shows the Scale Grid

```
public void toggleGrid()
{
    if (GridIsOn)
    {
        grid.SetActive(false);
        GridIsOn = false;
    }
    else
    {
        grid.SetActive(!grid.activeSelf);
        GridIsOn = true;
    }
}
```

//Allows users to raise grid to fit their board.

```
public void RaiseGrid()
{
    if (grid != null)
    {
        grid.transform.position += new Vector3(0, 0.1f, 0);
    }
}
```

//Allows for lowering of grid.

```
public void LowerGrid()
{
    if (grid != null)
    {
        grid.transform.position -= new Vector3(0, 0.1f, 0);
    }
}
```

```

}

//Tells the program to apply the shock throughout the simulation
public void toggleShock()
{
    if (!isShockActive)
    {
        isShockActive = true;
    }
    else
    {
        isShockActive = false;
    }
}

//Tells the program to apply the vibration throughout the simulation
public void toggleVibration()
{
    if (!isVibrationActive)
    {
        isVibrationActive = true;
    }
    else
    {
        isVibrationActive = false;
    }
}

//Properties class
public class MaterialProperties
{
    public float density;
    public float resistivity;
    public float coefficientOfFriction;

    public MaterialProperties(float density, float resistivity, float coefficientOfFriction)
    {
        this.density = density;
        this.resistivity = resistivity;
        this.coefficientOfFriction = coefficientOfFriction;
    }
}

public void OnSaveInputButtonClicked()
{
    //calls the savesettings method from SimulationController script to save inputs

```



```

        simulationController.SaveSettings();
    }

    public void OnLoadInputButtonClicked()
    {
        simulationController.LoadSettings(); //Calls the LoadSettings method from
SimulationController
    }

    public void PopulateConductorDropdowns() //Critical Pair UI
    {
        //Find all gameobjects with the tag "ConductorTrigger"
        GameObject[] conductorObjects =
GameObject.FindGameObjectsWithTag("ConductorTrigger");

        //Extract unique conductor names
        List<string> conductorNames = new List<string>();
        foreach (GameObject obj in conductorObjects)
        {
            string name = obj.name.Replace("_ColliderCopy", "");
            if (!conductorNames.Contains(name))
            {
                conductorNames.Add(name);
            }
        }

        //sort names alphabetically
        conductorNames.Sort();

        //clear existing options
        conductor1Dropdown.ClearOptions();
        conductor2Dropdown.ClearOptions();

        //Add options to dropdowns
        conductor1Dropdown.AddOptions(conductorNames);
        conductor2Dropdown.AddOptions(conductorNames);
    }

    public string CreatePairKey(string conductorA, string conductorB) //critical pair UI
    {
        var orderedPair = new[] { conductorA, conductorB }.OrderBy(name => name).ToArray();
        return $"{orderedPair[0]}-{orderedPair[1]}";
    }

    public void OnAddPairButtonClicked() //critical pair UI
    {

```

```

string conductorA = conductor1Dropdown.options[conductor1Dropdown.value].text;
string conductorB = conductor2Dropdown.options[conductor2Dropdown.value].text;

string pairKey = CreatePairKey(conductorA, conductorB);

if (!criticalPairs.Contains(pairKey))
{
    criticalPairs.Add(pairKey);
    AddCriticalPairToUI(conductorA, conductorB);
}
else
{
    Debug.Log("Critical pair already exists.");
}
}

public void OnRemovePairButtonClicked() //critical pair UI
{
    string conductorA = conductor1Dropdown.options[conductor1Dropdown.value].text;
    string conductorB = conductor2Dropdown.options[conductor2Dropdown.value].text;

    string pairKey = CreatePairKey(conductorA, conductorB);

    if (criticalPairs.Contains(pairKey))
    {
        criticalPairs.Remove(pairKey);
        RemoveCriticalPairFromUI(pairKey);
    }
    else
    {
        Debug.Log("Critical pair does not exist.");
    }
}

public void AddCriticalPairToUI(string conductorA, string conductorB)
{
    // Instantiate a new list item under the Content GameObject
    GameObject newItem = Instantiate(criticalPairItemPrefab, criticalPairsScrollView.content);

    // Set the text to display the pair
    TextMeshProUGUI itemText =
newItem.GetComponentInChildren<TextMeshProUGUI>();
    if (itemText != null)
    {
        itemText.text = $"{conductorA} - {conductorB}";
    }
}

```

```

else
{
    Debug.LogError("TextMeshProUGUI component not found in CriticalPairItemPrefab.");
}

// Store the pair key in the item's name for easy removal
newItem.name = CreatePairKey(conductorA, conductorB);

// Add to the list of UI items
criticalPairItems.Add(newItem);
}

public void RemoveCriticalPairFromUI(string pairKey)
{
    // Find the UI item corresponding to the pairKey
    GameObject itemToRemove = criticalPairItems.FirstOrDefault(item => item.name ==
pairKey);

    if (itemToRemove != null)
    {
        criticalPairItems.Remove(itemToRemove);
        Destroy(itemToRemove);
    }
    else
    {
        Debug.Log($"No UI item found with name {pairKey}.");
    }
}

private IEnumerator DelayedPopulateConductorDropdowns()
{
    // Wait until the end of the frame
    yield return new WaitForEndOfFrame();

    // Now populate the conductor dropdowns
    PopulateConductorDropdowns();
}

public void OnRefreshButtonClicked()
{
    PopulateConductorDropdowns();
}

//Code for getting input variables and storing them so they can be inserted into .csv
private List<KeyValuePair<string, string>> GetSimulationInputs()

```

```

{
    List<KeyValuePair<string, string>> inputs = new List<KeyValuePair<string, string>>();

    // Collect inputs from UIScript
    inputs.Add(new KeyValuePair<string, string>("Length Mu", lengthMu.text));
    inputs.Add(new KeyValuePair<string, string>("Length Sigma", lengthSigma.text));
    inputs.Add(new KeyValuePair<string, string>("Width Mu", widthMu.text));
    inputs.Add(new KeyValuePair<string, string>("Width Sigma", widthSigma.text));
    inputs.Add(new KeyValuePair<string, string>("# Of Whiskers", numWhiskers.text));
    inputs.Add(new KeyValuePair<string, string>("Conductive Material Selection",
material_input.text));
    inputs.Add(new KeyValuePair<string, string>("Whisker Material",
whiskMat.options[whiskMat.value].text));
    inputs.Add(new KeyValuePair<string, string>("Distribution Selection",
distributionDropdown.options[distributionDropdown.value].text));
    inputs.Add(new KeyValuePair<string, string>("Whisker Spawn Point X",
WhiskerSpawnPointX.text));
    inputs.Add(new KeyValuePair<string, string>("Whisker Spawn Point Y",
WhiskerSpawnPointY.text));
    inputs.Add(new KeyValuePair<string, string>("Whisker Spawn Point Z",
WhiskerSpawnPointZ.text.Trim()));
    inputs.Add(new KeyValuePair<string, string>("X-Coord", xCoord.text));
    inputs.Add(new KeyValuePair<string, string>("Y-Coord", yCoord.text));
    inputs.Add(new KeyValuePair<string, string>("Z-Coord", zCoord.text));
    inputs.Add(new KeyValuePair<string, string>("Gravity",
whiskerControl.gravity.options[whiskerControl.gravity.value].text));
    inputs.Add(new KeyValuePair<string, string>("# of Iterations", numIterations.text));
    inputs.Add(new KeyValuePair<string, string>("Directory Path",
whiskerControl.directoryPath));
    inputs.Add(new KeyValuePair<string, string>("Save File Name",
whiskerControl.fileName));

    // Add Wall Height from WallCreator
    inputs.Add(new KeyValuePair<string, string>("Wall Height",
wallCreator.wallHeightInput.text));

    // Add Rotations from CircuitBoardFinder
    inputs.Add(new KeyValuePair<string, string>("Rotation-X",
circuitBoardFinder.CircuitRotateX.text));
    inputs.Add(new KeyValuePair<string, string>("Rotation-Y",
circuitBoardFinder.CircuitRotateY.text));
    inputs.Add(new KeyValuePair<string, string>("Rotation-Z",
circuitBoardFinder.CircuitRotateZ.text));

    // Add Spin Rates from CircuitBoardFinder

```

```

        inputs.Add(new KeyValuePair<string, string>("Spin-X",
circuitBoardFinder.SpinRateX.text));
        inputs.Add(new KeyValuePair<string, string>("Spin-Y",
circuitBoardFinder.SpinRateY.text));
        inputs.Add(new KeyValuePair<string, string>("Spin-Z",
circuitBoardFinder.SpinRateZ.text));

        inputs.Add(new KeyValuePair<string, string>("Shock Amplitude", shockAmp.text));

// For checkboxes, output Y/N
string shockActive = isShockActive ? "Y" : "N";
inputs.Add(new KeyValuePair<string, string>("Shock Active", shockActive));

inputs.Add(new KeyValuePair<string, string>("Vibration Amplitude", vibrAmp.text));
inputs.Add(new KeyValuePair<string, string>("Vibration Frequency", vibrFreq.text));
inputs.Add(new KeyValuePair<string, string>("Vibration Duration", vibrDur.text));

string vibrationActive = isVibrationActive ? "Y" : "N";
inputs.Add(new KeyValuePair<string, string>("Vibration Active", vibrationActive));

// Add debug statements to log collected inputs before filtering
Debug.Log("Collected Simulation Inputs (before filtering):");
foreach (var input in inputs)
{
    Debug.Log($"Key: '{input.Key}', Value: '{input.Value}'");
}

//remove empty entries
inputs = inputs.Where(input => !string.IsNullOrEmpty(input.Key) &&
!string.IsNullOrEmpty(input.Value)).ToList();

// Add debug statements to log inputs after filtering
Debug.Log("Simulation Inputs (after filtering empty entries):");
foreach (var input in inputs)
{
    Debug.Log($"Key: '{input.Key}', Value: '{input.Value}'");
}

return inputs;
}

}

```

CallForces.cs

```

using System.Collections;

```

```

using System.Collections.Generic;
using UnityEngine;

//This script is solely used to hide or show the external force UI
public class CallForces : MonoBehaviour
{
    public GameObject VibrationUI;
    public GameObject ShockUI;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }

    public void toggleShockForces()
    {
        ShockUI.SetActive(!ShockUI.activeSelf);
    }

    public void hideShockForces()
    {
        ShockUI.SetActive(false);
    }

    public void toggleVibrForces()
    {
        VibrationUI.SetActive(!VibrationUI.activeSelf);
    }

    public void hideVibrForces()
    {
        VibrationUI.SetActive(false);
    }
}

```

CameraControl.cs

```

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

```

```

using UnityEngine;

public class CameraControl : MonoBehaviour
{
    public float moveSpeed = 10f;
    public float rotationSpeed = 90000000f;
    public GameObject UIObject;
    public UIScript uiScript;

    // Start is called before the first frame update
    void Start()
    {

    }

    //freely moving camera based off user inputs.
    void Update()
    {
        // Rotates the camera based on user input
        if (Input.GetMouseButton(0))
        {
            float horizontalRotation = Input.GetAxis("Mouse X") * rotationSpeed;
            float verticalRotation = Input.GetAxis("Mouse Y") * rotationSpeed;

            transform.Rotate(Vector3.up, horizontalRotation);
            transform.Rotate(Vector3.left, verticalRotation);

            // Moves the camera based on user input
            float horizontalInput = Input.GetAxis("Horizontal");
            float verticalInput = Input.GetAxis("Vertical");

            Vector3 moveDirection = new Vector3(horizontalInput, 0f, verticalInput).normalized;
            transform.Translate(moveDirection * moveSpeed * Time.deltaTime);

            // Moves the camera vertically with Q and E
            if (Input.GetKey(KeyCode.Q))
            {
                transform.Translate(Vector3.down * moveSpeed * Time.deltaTime, Space.World);
            }
            if (Input.GetKey(KeyCode.E))
            {
                transform.Translate(Vector3.up * moveSpeed * Time.deltaTime, Space.World);
            }
        }

        if (Input.GetMouseButtonDown(1))
    }
}

```

```

    {
        Quaternion currentRotation = transform.rotation;
        transform.rotation = Quaternion.Euler(currentRotation.eulerAngles.x,
currentRotation.eulerAngles.y,0);
    }
}

//Preset camera angles for the user to click to if needed.
public void TopDownView()
{
    transform.position = new Vector3(0, 85, 0);
    transform.rotation = Quaternion.Euler(90,180,0);
}

public void StandardView()
{
    transform.position = new Vector3(0, 48, 68);
    transform.rotation = Quaternion.Euler(36,180,0);
}

public void SideView()
{
    transform.position = new Vector3(73,5,0);
    transform.rotation = Quaternion.Euler(5,-90,0);
}
}

```

CameraOrbit.cs

```

using System.Collections.Generic;
using UnityEngine;
using TMPro; // Required for TextMeshPro

public class CameraOrbit : MonoBehaviour
{
    public float orbitSpeed = 1f;           // Speed of orbiting
    public float distanceFromTarget = 5f;    // Distance from the target object
    public Vector3 defaultPosition;          // Default position of the camera
    public Vector3 defaultRotation;          // Default rotation of the camera
    public float speedChangeAmount = 15f;    // Amount to change the orbit speed
    public TextMeshProUGUI OrbitWhiskerName; // Reference to the UI text element for
displaying whisker name
    public TMP_Dropdown whiskerDropdown;     // Reference to the UI dropdown for
whisker selection

    private List<Transform> bridgedWhiskers; // List of game objects with the tag
"bridgedWhisker"

```



```

    private List<string> previousWhiskerNames;    // List to store the last state of whisker names
for comparison
    private int currentTargetIndex = -1;        // Current target index for orbiting
    private bool isOrbiting = false;            // Is the camera currently orbiting
    private bool isMouseControlled = false;      // Is the camera being controlled by the mouse
    private Vector3 lastMousePosition;          // Last position of the mouse during movement

void Start()
{
    // Store the default position and rotation of the camera
    defaultPosition = transform.position;
    defaultRotation = transform.rotation.eulerAngles;

    // Initialize the list of bridged whiskers
    UpdateBridgedWhiskers();

    // Populate the dropdown with the whisker names
    PopulateDropdown();

    // Ensure the text element is cleared at the start
    OrbitWhiskerName.text = "";
}

void Update()
{
    // Continuously update the list of bridged whiskers to check for any changes
    UpdateBridgedWhiskers();

    // Check if the whisker list has changed and update the dropdown if necessary
    if (HasWhiskerListChanged())
    {
        PopulateDropdown();
    }

    // Check for mouse button to take over control of the camera
    HandleMouseControl();

    // Check for the up arrow key to start orbiting to the next whisker
    if (Input.GetKeyDown(KeyCode.UpArrow))
    {
        if (bridgedWhiskers.Count > 0)
        {
            // Move to the next whisker in numerical order
            currentTargetIndex = (currentTargetIndex + 1) % bridgedWhiskers.Count;
            UpdateWhiskerText(bridgedWhiskers[currentTargetIndex]);
            isOrbiting = true; // Start orbiting the selected whisker
        }
    }
}

```

```

    }
}

// Check for the down arrow key to reset the camera
if (Input.GetKeyDown(KeyCode.DownArrow))
{
    ResetCamera();
}

// Check for keys to change the distance from the target
if (Input.GetKey(KeyCode.W)) // Increase distance
{
    distanceFromTarget += Time.deltaTime * orbitSpeed; // Change speed if necessary
}
if (Input.GetKey(KeyCode.S)) // Decrease distance
{
    distanceFromTarget = Mathf.Max(1f, distanceFromTarget - Time.deltaTime *
orbitSpeed); // Prevent going below 1 unit
}

// Adjust orbit speed with [ and ]
if (Input.GetKey(KeyCode.LeftBracket)) // Decrease orbit speed
{
    orbitSpeed = Mathf.Max(1f, orbitSpeed - speedChangeAmount * Time.deltaTime);
}
if (Input.GetKey(KeyCode.RightBracket)) // Increase orbit speed
{
    orbitSpeed += speedChangeAmount * Time.deltaTime;
}

// Handle orbiting the current target if mouse control is not active
if (isOrbiting && !isMouseControlled && currentTargetIndex >= 0 && currentTargetIndex
< bridgedWhiskers.Count)
{
    OrbitAround(bridgedWhiskers[currentTargetIndex]);
}
}

void HandleMouseControl()
{
    // Check if the left mouse button is pressed along with the Right Shift key
    if (Input.GetMouseButtonDown(0) && Input.GetKey(KeyCode.RightShift))
    {
        // Start mouse control when the left mouse button and key are pressed
        isMouseControlled = true;
        isOrbiting = false; // Disable automatic orbit when mouse control begins
    }
}

```

```

        lastMousePosition = Input.mousePosition;
    }

    // Continue handling mouse movement only if the mouse is held down and the key is pressed
    if (Input.GetMouseButton(0) && Input.GetKey(KeyCode.RightShift) &&
isMouseControlled)
    {
        // Calculate mouse movement difference
        Vector3 mouseDelta = Input.mousePosition - lastMousePosition;
        lastMousePosition = Input.mousePosition;

        // Rotate the camera based on mouse movement
        float rotationX = mouseDelta.x * orbitSpeed * Time.deltaTime;
        float rotationY = -mouseDelta.y * orbitSpeed * Time.deltaTime;

        // Apply the rotation around the target
        transform.RotateAround(bridgedWhiskers[currentTargetIndex].position, Vector3.up,
rotationX);
        transform.RotateAround(bridgedWhiskers[currentTargetIndex].position, transform.right,
rotationY);
    }

    // Stop mouse control when the left mouse button is released
    if (Input.GetMouseButtonUp(0))
    {
        isMouseControlled = false;
    }
}

void UpdateBridgedWhiskers()
{
    bridgedWhiskers = new List<Transform>();
    GameObject[] whiskerObjects =
GameObject.FindGameObjectsWithTag("bridgedWhisker");

    foreach (GameObject obj in whiskerObjects)
    {
        bridgedWhiskers.Add(obj.transform);
    }

    // Sort the whiskers by the numeric part in their names
    bridgedWhiskers.Sort((a, b) =>
ExtractNumber(a.name).CompareTo(ExtractNumber(b.name)));
}

int ExtractNumber(string name)
{

```

```

// Assumes the number in the whisker name is the last part after an underscore
string[] parts = name.Split('_');
if (parts.Length > 1 && int.TryParse(parts[1], out int number))
{
    return number; // Return the extracted number
}
return int.MaxValue; // If no valid number found, return max value to sort last
}

bool HasWhiskerListChanged()
{
    List<string> currentWhiskerNames = new List<string>();
    foreach (Transform whisker in bridgedWhiskers)
    {
        currentWhiskerNames.Add(whisker.name);
    }

    // Compare the new list with the previous list of whisker names
    if (previousWhiskerNames == null || currentWhiskerNames.Count !=
previousWhiskerNames.Count)
    {
        previousWhiskerNames = currentWhiskerNames;
        return true; // List has changed
    }

    for (int i = 0; i < currentWhiskerNames.Count; i++)
    {
        if (currentWhiskerNames[i] != previousWhiskerNames[i])
        {
            previousWhiskerNames = currentWhiskerNames;
            return true; // List has changed
        }
    }

    return false; // No changes detected
}

void PopulateDropdown()
{
    if (bridgedWhiskers == null || bridgedWhiskers.Count == 0)
    {
        Debug.LogWarning("No bridged whiskers found to populate the dropdown.");
        return; // Early exit if there are no whiskers to add
    }

    // Clear the current options in the dropdown

```

```

whiskerDropdown.ClearOptions();

// Create a list of whisker names
List<string> whiskerNames = new List<string>();
foreach (Transform whisker in bridgedWhiskers)
{
    whiskerNames.Add(whisker.name);
}

// Add the whisker names to the dropdown
whiskerDropdown.AddOptions(whiskerNames);

// Add a listener to handle when the user selects a new option
whiskerDropdown.onValueChanged.AddListener(OnDropdownValueChanged);

Debug.Log("Dropdown successfully populated with whisker names.");
}

void OnDropdownValueChanged(int selectedIndex)
{
    if (selectedIndex >= 0 && selectedIndex < bridgedWhiskers.Count)
    {
        // Update the camera to focus on the selected whisker
        currentTargetIndex = selectedIndex;
        UpdateWhiskerText(bridgedWhiskers[currentTargetIndex]);
        isOrbiting = true; // Start orbiting the selected whisker
        Debug.Log($"Dropdown selection changed to:
{bridgedWhiskers[selectedIndex].name}");
    }
}

void OrbitAround(Transform target)
{
    // Calculate the new position of the camera based on the target
    Vector3 direction = (transform.position - target.position).normalized;
    Quaternion rotation = Quaternion.Euler(0, orbitSpeed * Time.deltaTime, 0);
    Vector3 newPosition = target.position + rotation * direction * distanceFromTarget;

    // Update the camera's position and rotation
    transform.position = newPosition;
    transform.LookAt(target.position);
}

void ResetCamera()
{
    transform.position = defaultPosition;
}

```

```

transform.rotation = Quaternion.Euler(defaultRotation);
isOrbiting = false;
currentTargetIndex = -1; // Reset the target index
distanceFromTarget = 5f; // Reset distance to default if desired

// Clear the text when resetting
OrbitWhiskerName.text = "";
}

void UpdateWhiskerText(Transform target)
{
    Debug.Log($"Updating text for target: {target.name}"); // Debug statement

    // Always display the static text
    string staticText = "Red = Bridged\nOrange = Momentary\n";

    // Update the text with the name of the current target whisker
    OrbitWhiskerName.text = staticText + target.gameObject.name;
}
}

```

DataManager.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DataManager : MonoBehaviour
{
    public static DataManager Instance { get; private set; }

    public List<WhiskerData> allWhiskersData = new List<WhiskerData>();
    public List<WhiskerData> bridgedWhiskersData = new List<WhiskerData>();
    public List<WhiskerData> criticalBridgedWhiskersData = new List<WhiskerData>(); // For
critical pairs

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
            DontDestroyOnLoad(gameObject); // Optional: Keeps DataManager persistent across
scenes
        }
        else
        {
            Destroy(gameObject);
        }
    }
}

```

```

    }
}

```

DataSaveManager.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DataSaveManager : MonoBehaviour
{
    public static int CurrentRowIndex { get; set; } = -1;
}

```

DistSelect.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

//used to allow the dropdown list to change between the lognormal/normal distributions, applies
the selection in the UIScript for selection
public class DistributionSelector : MonoBehaviour
{
    public TMP_Dropdown dropdown; // Dropdown reference
    public UIScript uiScript; // Reference to your UIScript

    void Start()
    {
        dropdown.onValueChanged.AddListener(delegate {
            DropdownValueChanged(dropdown);
        });
    }

    void DropdownValueChanged(TMP_Dropdown change)
    {
        switch (change.value)
        {
            case 0:
                uiScript.distributionType = DistributionType.Lognormal;
                break;
            case 1:
                uiScript.distributionType = DistributionType.Normal;
                break;
        }
    }
}

```

```

    }
}

public enum DistributionType
{
    Normal,
    Lognormal
}

```

HeatMap.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//This scripts controls the heatmap feature.
public class HeatMap : MonoBehaviour
{
    public Material triggerMaterial;
    public GameObject HeatUI;
    public bool toggleColorOn = false;
    public float minPing = 0f;
    public float maxPing = 10f;

    //Button to the tell the heatmap to turn on.
    public void heatMapButton()
    {
        if(toggleColorOn)
        {
            heatMapToggleOFF();
            HeatUI.SetActive(false);
        }
        else
        {
            HeatUI.SetActive(!HeatUI.activeSelf);
            toggleColorOn = true;
        }
    }
    void Start()
    {
        CalculateColorGradient();
    }
    void Update()
    {
        if(toggleColorOn)
        {
            heatMapToggleON();
        }
    }
}

```



```

    }
    //once heatmap is on, this will run constantly, and updates the colors as bridges form.
    public void heatMapToggleON()
    {
        GameObject[] triggerObjects =
        GameObject.FindGameObjectsWithTag("ConductorTrigger");

        foreach (GameObject triggerObject in triggerObjects)
        {
            TriggerTracker triggerTracker = triggerObject.GetComponent<TriggerTracker>();

            if (triggerTracker != null)
            {
                int pingCount = triggerTracker.GetPingCount();
                Renderer renderer = triggerObject.GetComponent<Renderer>();

                if (renderer != null)
                {
                    if (pingCount >= 5)
                    {
                        renderer.material.color = Color.red;
                    }
                    else if (pingCount >= 4)
                    {
                        renderer.material.color = colorGradient[4];
                    }
                    else if (pingCount >= 3)
                    {
                        renderer.material.color = colorGradient[3];
                    }
                    else if (pingCount >= 2)
                    {
                        renderer.material.color = Color.yellow;
                    }
                    else if (pingCount >= 1)
                    {
                        renderer.material.color = colorGradient[2];
                    }
                    else
                    {
                        renderer.material.color = Color.green;
                    }
                }
            }
        }
        toggleColorOn = true;
    }

```

```

    }

    //Turns heatmap off.
    public void heatMapToggleOFF()
    {
        GameObject[] triggerObjects =
        GameObject.FindGameObjectsWithTag("ConductorTrigger");
        foreach (GameObject triggerObject in triggerObjects)
        {
            Renderer renderer = triggerObject.GetComponent<Renderer>();
            renderer.material = triggerMaterial;
        }
        toggleColorOn = false;
    }
    Color[] colorGradient = new Color[5];
    void CalculateColorGradient()
    {
        Color startColor = Color.green;
        Color middleColor = Color.yellow;
        Color endColor = Color.red;
        colorGradient[0] = Color.Lerp(startColor, middleColor, 0.25f); // Green to Yellow (25%)
        colorGradient[1] = Color.Lerp(startColor, middleColor, 0.5f); // Green to Yellow (50%)
        colorGradient[2] = Color.Lerp(startColor, middleColor, 0.75f); // Green to Yellow (75%)
        colorGradient[3] = Color.Lerp(middleColor, endColor, 0.25f); // Yellow to Red (25%)
        colorGradient[4] = Color.Lerp(middleColor, endColor, 0.5f); // Yellow to Red (50%)
    }
}

```

MatSelect.cs

```

using System;
using UnityEngine;

//This allows the user to select between the three materials of Tin, Zinc, and Cadmium from the
dropdown list.
public enum MaterialType
{
    Tin,
    Zinc,
    Cadmium
}

public class MatSelect : MonoBehaviour
{
    public MaterialType selectedMaterial = MaterialType.Tin;
}

```

```

void Start()
{
    selectedMaterial = MaterialType.Tin; // Default selection
}
}

```

ResetBoard.cs

```

using UnityEngine;
using UnityEngine.UI;

public class ToggleButton : MonoBehaviour
{
    public Toggle RotateSpinToggle; // Reference to the Toggle UI element
    public Button StartSimulation; // Reference to the Button UI element

    void Start()
    {
        RotateSpinToggle.isOn = !RotateSpinToggle.isOn; // Toggle the state
    }

    // Method to toggle the toggle state
    void ToggleState()
    {
        RotateSpinToggle.isOn = !RotateSpinToggle.isOn; // Toggle the state
    }
}

```

ShockManager.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;
using UnityEngine.UI;
using System;

//This script controls one of the external forces: Shock
public class ShockManager : MonoBehaviour
{
    public TMP_InputField amplitudeInputField;
    public Transform circuitBoard;
    private Rigidbody circuitBoardRigidbody;
    private Vector3 originalBoardPosition;
    private float amplitude;
    private float duration = 0.1f; // Hardcoded duration of the shock
    private float frequency = 5f; // Hardcoded frequency of the shock
    private float startTime;
}

```

```

private bool isShockingX = false;
private bool isShockingY = false;
private bool isShockingZ = false;
public TMP_Dropdown shockAxis;
public Button shockButton;
public UIScript uiScript;

private void Start()
{

}

//Runs the shock force every 2 seconds based on what axis the user has selected.
private void FixedUpdate()
{
    if (isShockingX)
    {
        float elapsedTime = Time.time - startTime;
        if (elapsedTime < duration)
        {
            float sineValue = Mathf.Sin(elapsedTime * frequency * 2 * Mathf.PI);
            float offsetX = sineValue * amplitude;

            Vector3 targetBoardPosition = new Vector3(originalBoardPosition.x + offsetX,
originalBoardPosition.y, originalBoardPosition.z);
            if (circuitBoardRigidbody != null)
            {
                circuitBoardRigidbody.MovePosition(targetBoardPosition);
            }
        }
        else
        {
            isShockingX = false;
            Debug.Log("Shock finished.");
            ShockFinished();
            if (circuitBoard != null)
            {
                circuitBoard.position = originalBoardPosition;
            }
        }
    }

    if (isShockingY)
    {
        float elapsedTime = Time.time - startTime;
        if (elapsedTime < duration)

```

```

{
    float sineValue = Mathf.Sin(elapsedTime * frequency * 2 * Mathf.PI);
    float offsetY = sineValue * amplitude;

    Vector3 targetBoardPosition = new Vector3(originalBoardPosition.x,
originalBoardPosition.y + offsetY, originalBoardPosition.z);
    if (circuitBoardRigidbody != null)
    {
        circuitBoardRigidbody.MovePosition(targetBoardPosition);
    }
}
else
{
    isShockingY = false;
    Debug.Log("Shock finished.");
    ShockFinished();

    if (circuitBoard != null)
    {
        circuitBoard.position = originalBoardPosition;
    }
}
}

if (isShockingZ)
{
    float elapsedTime = Time.time - startTime;
    if (elapsedTime < duration)
    {
        float sineValue = Mathf.Sin(elapsedTime * frequency * 2 * Mathf.PI);
        float offsetZ = sineValue * amplitude;

        Vector3 targetBoardPosition = new Vector3(originalBoardPosition.x,
originalBoardPosition.y, originalBoardPosition.z + offsetZ);
        if (circuitBoardRigidbody != null)
        {
            circuitBoardRigidbody.MovePosition(targetBoardPosition);
        }
    }
}
else
{
    isShockingZ = false;
    Debug.Log("Shock finished.");
    ShockFinished();
}
}

```

```

        if (circuitBoard != null)
        {
            circuitBoard.position = originalBoardPosition;
        }
    }
}

//controls to tell the code which one of the previous functions to run.
public void StartShockX()
{
    circuitBoardRigidbody = circuitBoard.GetComponent<Rigidbody>();
    originalBoardPosition = circuitBoard.position;

    if (float.TryParse(amplitudeInputField.text, out amplitude))
    {
        Debug.Log("Starting shock with parameters:");
        Debug.Log("Amplitude: " + amplitude);
        Debug.Log("Duration: " + duration);
        Debug.Log("Frequency: " + frequency);

        startTime = Time.time;
        isShockingX = true;
        Debug.Log("Shock started.");
    }
}

public void StartShockY()
{
    circuitBoardRigidbody = circuitBoard.GetComponent<Rigidbody>();
    originalBoardPosition = circuitBoard.position;

    if (float.TryParse(amplitudeInputField.text, out amplitude))
    {
        Debug.Log("Starting shock with parameters:");
        Debug.Log("Amplitude: " + amplitude);
        Debug.Log("Duration: " + duration);
        Debug.Log("Frequency: " + frequency);

        startTime = Time.time;
        isShockingY = true;
        Debug.Log("Shock started.");
    }
}

public void StartShockZ()

```

```

{
    circuitBoardRigidbody = circuitBoard.GetComponent<Rigidbody>();
    originalBoardPosition = circuitBoard.position;

    if (float.TryParse(amplitudeInputField.text, out amplitude))
    {
        Debug.Log("Starting shock with parameters:");
        Debug.Log("Amplitude: " + amplitude);
        Debug.Log("Duration: " + duration);
        Debug.Log("Frequency: " + frequency);

        startTime = Time.time;
        isShockingZ = true;
        Debug.Log("Shock started.");
    }
}

//Button to begin or apply the Shock/Impact force to the board.
public void shockPressed()
{
    if(Convert.ToInt32(amplitudeInputField.text) > 25)
    {
        uiScript.SetErrorMessage("Shock Amplitude is too high.");
        return;
    }
    GameObject circuitBoardObject = GameObject.FindGameObjectWithTag("CircuitBoard");
    if (circuitBoardObject != null)
    {
        circuitBoard = circuitBoardObject.transform;
    }
    shockButton.interactable = false;
    int selectedAxis = shockAxis.value;
    switch (selectedAxis)
    {
        case 0:
            StartShockX();
            break;
        case 1:
            StartShockY();
            break;
        case 2:
            StartShockZ();
            break;
    }
}

private void ShockFinished()

```

```

    {
        shockButton.interactable = true;
    }
}

```

TriggerControl.cs

```

using System.Collections;
using System.Collections.Generic;
using Unity.VisualScripting;
/*using UnityEditor.Callbacks;*/
using UnityEngine;
using TMPro;
using System.Linq;
using JetBrains.Annotations;

//This script sets up the circuit board in order to be simulated.
/*It does this by applying colliders to the base components of the board for physical interations,
and then copies the colliders to an empty object which is placed slightly above the original object
in order to act as the trigger which actually allows the whiskers to tell if a bridge has occured.*/
public class TriggerControl : MonoBehaviour
{
    public List<GameObject> targetObjects = new List<GameObject>();
    public Rigidbody rb;
    public Vector3 offset = new Vector3(0, 0, 0.005f);
    public int maxDepth = 5;
    public Vector3 newScale = new Vector3(10, 10, 10);
    public TMP_InputField num_mat;
    public string material_text;
    public int num_mat_int;
    public Material triggerMaterial;
    public TMP_InputField material_input;
    public List<GameObject> objectvalues_ = new List<GameObject>();
    public List<GameObject> testerfill = new List<GameObject>();

    //Accesses the circuit board and applys proper settings in order for the simulation to run.
    //Additionally, this runs the scripts that applys colliders and triggers to the components of the
    board.
    void Start()
    {
        gameObject.tag = "CircuitBoard";

        Transform topTransform = transform.Find("Top");
        if (topTransform != null)
        {

```



```

        Destroy(topTransform.gameObject);
    }

    rb = gameObject.AddComponent<Rigidbody>();
    rb.isKinematic = true;
    rb.constraints = RigidbodyConstraints.FreezeRotation;
    rb.collisionDetectionMode = CollisionDetectionMode.Continuous;
    transform.localScale = newScale;
    transform.position = Vector3.zero;

    AddMeshCollidersRecursively(transform);
    StartCoroutine(CopyMeshCollidersToEmptyObjects(transform, 0));

    GameObject[] allObjects = GameObject.FindObjectsOfType<GameObject>();
    material_input.onValueChanged.AddListener(delegate { checkifconduct(testerfill,
objectvalues_, material_input.text); } );
}

// Update is called once per frame
void Update()
{
}

//Applies colliders to the child components on the board.
public void AddMeshCollidersRecursively(Transform parent)
{
    foreach (Transform child in parent)
    {
        if (child.GetComponent<Collider>() == null)
        {
            MeshCollider meshCollider = child.gameObject.AddComponent<MeshCollider>();
            meshCollider.convex = true;
        }
        AddMeshCollidersRecursively(child);
    }
}

//Copies the colliders to the copied objects, which are triggers for the whiskers
IEnumerator CopyMeshCollidersToEmptyObjects(Transform parent, int depth)
{
    if (depth > maxDepth)
    {
        yield break;
    }

    List<Transform> children = new List<Transform>();
    foreach (Transform child in parent)

```

```

    {
        children.Add(child);
    }
    foreach (Transform child in children)
    {
        if (child == null || child.GetComponent<MeshCollider>() == null)
        {
            continue;
        }

        MeshCollider originalMeshCollider = child.GetComponent<MeshCollider>();
        if (originalMeshCollider != null)
        {
            testerfill.Add(child.gameObject);
            GameObject emptyObject = new GameObject(child.name + "_ColliderCopy");

            emptyObject.transform.position = child.TransformPoint(offset);
            emptyObject.transform.rotation = child.rotation;
            emptyObject.transform.localScale = child.lossyScale;

            MeshCollider newMeshCollider = emptyObject.AddComponent<MeshCollider>();
            newMeshCollider.sharedMesh = originalMeshCollider.sharedMesh;
            newMeshCollider.convex = originalMeshCollider.convex;
            newMeshCollider.isTrigger = true;

            emptyObject.transform.SetParent(parent, true);

            Renderer renderer = emptyObject.AddComponent<MeshRenderer>();
            MeshFilter meshFilter = emptyObject.AddComponent<MeshFilter>();
            meshFilter.mesh = originalMeshCollider.sharedMesh;
            renderer.material = new Material(triggerMaterial);

            emptyObject.AddComponent<TriggerTracker>();
            objectvalues_.Add(emptyObject);
        }

        yield return StartCoroutine(CopyMeshCollidersToEmptyObjects(child, depth + 1));
    }
}

//Checks if object is a conductor by comparing materials to the open given by the users.
public void checkifconduct(List<GameObject> values, List<GameObject> emptylist, string
text_value)
{
    Renderer[] allRenderers = FindObjectsOfType<Renderer>();
    foreach (Renderer renderer in allRenderers)

```

```

{
    foreach (Material material in renderer.materials)
    {
        string materialName = material.name.Replace(" (Instance)", "");
        if (materialName.Equals(text_value, System.StringComparison.OrdinalIgnoreCase))
        {
            GameObject obj = renderer.gameObject;
            obj.tag = "Conductor";
            for (int i = 0; i < emptylist.Count; i++)
            {
                if (emptylist[i].name.StartsWith(obj.name))
                {
                    emptylist[i].tag = "ConductorTrigger";
                    break;
                }
            }
            break;
        }
    }
}
}
}
}

```

TriggerTracker.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPro;
using Unity.VisualScripting;
using UnityEngine;

```

//This script tracks how many times each trigger has made contact with a whisker and formed a bridge

```

public class TriggerTracker : MonoBehaviour

```

```

{
    private int pingCount = 0;
    public UIScript uiScript;

    void Start()
    {
        uiScript = FindObjectOfType<UIScript>();
    }
    public void IncrementPingCount()
    {
        pingCount++;
        //Debug.Log($"Ping count for {gameObject.name}: {pingCount}"); //log to notify when
        //whiskers hit nodes, can uncomment whenever
    }
}

```

```

    }

    public int GetPingCount()
    {
        return pingCount;
    }
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("whiskerClone"))
        {
            StartCoroutine(CheckConnectionDelayed(other));
        }
    }

    private IEnumerator CheckConnectionDelayed(Collider other)
    {
        yield return new WaitForSeconds(0.1f);

        WhiskerControl whiskerControl = other.GetComponent<WhiskerControl>();
        if (whiskerControl != null && whiskerControl.haveLoggedConnection)
        {
            IncrementPingCount();
        }
    }
}

```

VibrationManager.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPPro;
using UnityEngine.UI;
using System;

//Controls the External Force: Vibration
public class VibrationManager : MonoBehaviour
{
    public TMP_InputField amplitudeInputField; // Input field for vibration amplitude
    public TMP_InputField durationInputField; // Input field for vibration duration
    public TMP_InputField frequencyInputField; // Input field for vibration frequency
    public Transform circuitBoard; // Reference to the circuit board transform
    private Rigidbody circuitBoardRigidbody; // Rigidbody of the circuit board
    private Vector3 originalBoardPosition; // Original position of the circuit board
    private float amplitude;
    private float duration;
    private float frequency;

```

```

private float startTime;
private bool isVibratingX = false;
private bool isVibratingY = false;
private bool isVibratingZ = false;
public TMP_Dropdown fAxis;
public Button vibrateButton;
public UIScript uiScript;

private void Start()
{

}

//Applies the correct vibration in the appropriate axis depending on the user inputs
private void FixedUpdate()
{
    if (isVibratingX)
    {
        float elapsedTime = Time.time - startTime;
        if (elapsedTime < duration)
        {
            float sineValue = Mathf.Sin(elapsedTime * frequency * 2 * Mathf.PI);
            float offsetX = sineValue * amplitude;

            Vector3 targetBoardPosition = new Vector3(originalBoardPosition.x + offsetX,
originalBoardPosition.y, originalBoardPosition.z);
            if (circuitBoardRigidbody != null)
            {
                circuitBoardRigidbody.MovePosition(targetBoardPosition);
            }
        }
        else
        {
            isVibratingX = false;
            Debug.Log("Vibration finished.");
            VibrationFinished();

            if (circuitBoard != null)
            {
                circuitBoard.position = originalBoardPosition;
            }
        }
    }

    if (isVibratingY)
    {

```

```

float elapsedTime = Time.time - startTime;
if (elapsedTime < duration)
{
    float sineValue = Mathf.Sin(elapsedTime * frequency * 2 * Mathf.PI);
    float offsetY = sineValue * amplitude;

    Vector3 targetBoardPosition = new Vector3(originalBoardPosition.x,
originalBoardPosition.y + offsetY, originalBoardPosition.z);
    if (circuitBoardRigidbody != null)
    {
        circuitBoardRigidbody.MovePosition(targetBoardPosition);
    }
}
else
{
    isVibratingY = false;
    Debug.Log("Vibration finished.");
    VibrationFinished();

    if (circuitBoard != null)
    {
        circuitBoard.position = originalBoardPosition;
    }
}
}

if (isVibratingZ)
{
    float elapsedTime = Time.time - startTime;
    if (elapsedTime < duration)
    {

        float sineValue = Mathf.Sin(elapsedTime * frequency * 2 * Mathf.PI);
        float offsetZ = sineValue * amplitude;

        Vector3 targetBoardPosition = new Vector3(originalBoardPosition.x,
originalBoardPosition.y, originalBoardPosition.z + offsetZ);
        if (circuitBoardRigidbody != null)
        {
            circuitBoardRigidbody.MovePosition(targetBoardPosition);
        }
    }
    else
    {
        isVibratingZ = false;
        Debug.Log("Vibration finished.");
    }
}

```

```

        VibrationFinished();

        if (circuitBoard != null)
        {
            circuitBoard.position = originalBoardPosition;
        }
    }
}

//These functions actually run the vibration
public void StartVibrationX()
{
    circuitBoardRigidbody = circuitBoard.GetComponent<Rigidbody>();
    originalBoardPosition = circuitBoard.position;

    if (float.TryParse(amplitudeInputField.text, out amplitude) &&
        float.TryParse(durationInputField.text, out duration) &&
        float.TryParse(frequencyInputField.text, out frequency))
    {
        Debug.Log("Starting vibration with parameters:");
        Debug.Log("Amplitude: " + amplitude);
        Debug.Log("Duration: " + duration);
        Debug.Log("Frequency: " + frequency);

        startTime = Time.time;
        isVibratingX = true;
        Debug.Log("Vibration started.");
    }
}

public void StartVibrationY()
{
    circuitBoardRigidbody = circuitBoard.GetComponent<Rigidbody>();
    originalBoardPosition = circuitBoard.position;

    if (float.TryParse(amplitudeInputField.text, out amplitude) &&
        float.TryParse(durationInputField.text, out duration) &&
        float.TryParse(frequencyInputField.text, out frequency))
    {
        Debug.Log("Starting vibration with parameters:");
        Debug.Log("Amplitude: " + amplitude);
        Debug.Log("Duration: " + duration);
    }
}

```

```

        Debug.Log("Frequency: " + frequency);

        startTime = Time.time;
        isVibratingY = true;
        Debug.Log("Vibration started.");
    }

}

public void StartVibrationZ()
{
    circuitBoardRigidbody = circuitBoard.GetComponent<Rigidbody>();
    originalBoardPosition = circuitBoard.position;

    if (float.TryParse(amplitudeInputField.text, out amplitude) &&
        float.TryParse(durationInputField.text, out duration) &&
        float.TryParse(frequencyInputField.text, out frequency))
    {
        Debug.Log("Starting vibration with parameters:");
        Debug.Log("Amplitude: " + amplitude);
        Debug.Log("Duration: " + duration);
        Debug.Log("Frequency: " + frequency);

        startTime = Time.time;
        isVibratingZ = true;
        Debug.Log("Vibration started.");
    }

}

//This runs whenever the vibration button is pressed or called.
public void vibratePressed()
{
    if(float.Parse(amplitudeInputField.text) > 35)
    {
        uiScript.SetErrorMessage("Vibration Amplitude is too high.");
        return;
    }
    if(float.Parse(frequencyInputField.text) > 150)
    {
        uiScript.SetErrorMessage("Vibration frequency is too high.");
        return;
    }
    if(float.Parse(durationInputField.text) > 30)
    {
        uiScript.SetErrorMessage("Vibration Amplitude is too high.");
    }
}

```



```

        return;
    }

    GameObject circuitBoardObject = GameObject.FindGameObjectWithTag("CircuitBoard");
    if (circuitBoardObject != null)
    {
        circuitBoard = circuitBoardObject.transform;
    }
    vibrateButton.interactable = false;

    int selectedAxis = fAxis.value;
    switch (selectedAxis)
    {
        case 0:
            StartVibrationX();
            break;
        case 1:
            StartVibrationY();
            break;
        case 2:
            StartVibrationZ();
            break;
    }
}
private void VibrationFinished()
{
    vibrateButton.interactable = true;
}
}

```

WhiskerControl.cs

```

using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using TMPro;
using UnityEngine;
using UnityEngine.UI;

//Handles most of script that goes along with the whiskers, such as setting colliders and handling
their collisions.
public class WhiskerControl : MonoBehaviour
{
    public Dropdown gravity;
    private ConstantForce cForce;
    private Vector3 forceDirection;

```

```

public int selectedIndex = 0;
public Material targetMaterial;
public int bridges = 0;
public bool confirmGravity;
public GameObject UIObject;
public UIScript uiScript;
public TMP_InputField filePathInputField;
public TMP_InputField fileNameInputField;
public Button saveButton;
public string directoryPath;
public string fileName;
public InputField customGravityInputX;
public InputField customGravityInputY;
public InputField customGravityInputZ;

//Initializes the UiScript
void Start()
{
    uiScript = UIObject.GetComponent<UIScript>();
    if (uiScript == null)
    {
        Debug.LogError("UIScript not found on UIObject.");
    }
    if (!bridgesPerConductor.ContainsKey(gameObject.name))
    {
        bridgesPerConductor[gameObject.name] = 0;
    }
}

public float detectionRadius = 0.5f;
public float rayDistance = 1.0f;
public LayerMask conductorLayer;

//Turns on the gravity
void Update()
{
    if (confirmGravity)
    {
        GetGravitySelection(gravity.value);
    }
}

//Confirm Gravity Button
public void ConfirmButtonPressed()
{

```

```

    GetGravitySelection(gravity.value);
    confirmGravity = true;
    uiScript.ReloadWhiskersButton();
    UIObject.GetComponent<UIScript>().startSim = true;
}

//Gets gravity selection from dropdown
public void GetGravitySelection(int val)
{
    ApplyGravity(val);
}

//Applies correct gravity
public void ApplyGravity(int val)
{
    // Find all objects with the tag "whiskerClone"
    GameObject[] objectsWithTag = GameObject.FindGameObjectsWithTag("whiskerClone");

    Vector3 forceDirection = Vector3.zero;

    // Set default values to "0" if the InputFields are null or empty
    if (customGravityInputX == null || string.IsNullOrEmpty(customGravityInputX.text))
    {
        customGravityInputX.text = "0";
    }

    if (customGravityInputY == null || string.IsNullOrEmpty(customGravityInputY.text))
    {
        customGravityInputY.text = "0";
    }

    if (customGravityInputZ == null || string.IsNullOrEmpty(customGravityInputZ.text))
    {
        customGravityInputZ.text = "0";
    }

    // Validate the input fields
    if (customGravityInputX != null && customGravityInputY != null &&
        customGravityInputZ != null)
    {
        if (float.TryParse(customGravityInputX.text, out float CGIX) &&
            float.TryParse(customGravityInputY.text, out float CGIY) &&
            float.TryParse(customGravityInputZ.text, out float CGIZ))
        {
            // Proceed with gravity application
            switch (val)

```

```

{
    case 0:
        forceDirection = new Vector3(0, -100, 0);
        // This acceleration is 0.1m/s^2 any faster and the whisker fly through the board

        break;
    case 1:
        forceDirection = new Vector3(0, -20, 0);
        // This acceleration is 0.02m/s^2 d
        break;
    case 2:
        forceDirection = new Vector3(0, -40, 0);
        // This acceleration is 0.04m/s^2
        break;
    case 3:
        forceDirection = new Vector3(CGIX*10, CGIY*10, CGIZ*10);
        break;
    default:
        Debug.LogWarning("Invalid gravity value provided.");
        return;
}

// Apply the force to each object
foreach (GameObject obj in objectsWithTag)
{
    ConstantForce cForce = obj.GetComponent<ConstantForce>() ??
obj.AddComponent<ConstantForce>();
    cForce.force = forceDirection;
}
}
else
{
    Debug.LogError("Failed to parse custom gravity inputs.");
}
}
else
{
    Debug.LogError("One or more custom gravity input fields are not assigned.");
}
}

//Resets gravity /ResetButton
public void ResetGravity()
{
    GameObject[] objectsWithTag = GameObject.FindGameObjectsWithTag("whiskerClone");
    foreach (GameObject obj in objectsWithTag)

```

```

    {
        cForce = GetComponent<ConstantForce>();
        forceDirection = new Vector3(0, 0, 0);
        cForce.force = forceDirection;
    }
    confirmGravity = false;
}

public bool haveLoggedConnection;
public bool haveMadeOneConnection;
public string firstConnection;
public bool haveMadeSecondConnection;
public string secondConnection;
public int connectionsMade;
public static Dictionary<string, int> bridgesPerConductor = new Dictionary<string, int>();
private Renderer objectRenderer;
public Color bridgeDetectedColor = Color.red;
public Color defaultColor = Color.white;
public HashSet<string> currentConnections = new HashSet<string>();
public Color[] colors;
private Dictionary<GameObject, int> triggerInteractionCounts = new
Dictionary<GameObject, int>();
public bool haveBridgedBefore;

// BRIDGING DETECTION
private void OnTriggerStay(Collider trigger)
{
    if (trigger.gameObject.CompareTag("ConductorTrigger"))
    {
        currentConnections.Add(trigger.gameObject.name);

        if (currentConnections.Count >= 2)
        {
            if (!haveLoggedConnection && !haveBridgedBefore)
            {
                Transform visualChild = transform.Find("visual");
                Renderer childRenderer = visualChild.GetComponent<Renderer>();

                objectRenderer = GetComponent<Renderer>();
                bridgesPerConductor[gameObject.name]++;
                UIObject.GetComponent<UIScript>().bridgesDetected++;
                UIObject.GetComponent<UIScript>().bridgesPerRun++;
                childRenderer.material.color = bridgeDetectedColor;
                haveLoggedConnection = true;

                TrackBridgedWhiskers(gameObject, currentConnections.ToList());
            }
        }
    }
}

```

```

        haveBridgedBefore = true; //uncomment for limit of one bridge.
    }
}
}

private void OnTriggerExit(Collider trigger)
{
    if (trigger.gameObject.CompareTag("ConductorTrigger"))
    {
        currentConnections.Remove(trigger.gameObject.name);

        if (currentConnections.Count < 2 && haveLoggedConnection)
        {
            ResetConnectionState();
        }
    }
}

//Resets connection whenever the whisker stops being in contact
private void ResetConnectionState()
{
    Transform visualChild = transform.Find("visual");
    Renderer childRenderer = visualChild.GetComponent<Renderer>();

    currentConnections.Clear();
    haveLoggedConnection = false;
    bridgesPerConductor[gameObject.name]--;
    childRenderer.material.color = new Color(1.0f, 0.647f, 0.0f);
}

//Saves Bridged Whiskers
public void TrackBridgedWhiskers(GameObject whisker, List<string> conductorNames)
{
    //Debug.Log($"TrackBridgedWhiskers called for whisker {whisker.name}");

    Vector3 scale = whisker.transform.localScale;
    float length = scale.y * 2;
    float diameter = (scale.x + scale.z) / 2;

    UIScript.MaterialProperties currentProps =
    uiScript.materialProperties[uiScript.currentMaterial];

    float resistance = CalculateResistance(length, diameter, currentProps);

```

```

// Extract whisker number from whisker's name
string whiskerName = whisker.name;
int whiskerNumber = int.Parse(whiskerName.Split('_')[1]);

//Get conductor name and remove _ColliderCopy from names
string conductor1 = conductorNames.Count > 0 ?
CleanConductorName(conductorNames[0]) : "";
string conductor2 = conductorNames.Count > 1 ?
CleanConductorName(conductorNames[1]) : "";

// Collect bridged whisker data
WhiskerData data = new WhiskerData(whiskerNumber, length * 1000, diameter * 1000,
resistance, uiScript.simIntComplete, conductor1, conductor2);

//Add to datamangers bridgedwhiskerdata list
DataManager.Instance.bridgedWhiskersData.Add(data);

Debug.Log($"Bridged whisker added: {data.WhiskerNumber}");

// Set the tag of the whisker to "bridgedWhisker"
whisker.tag = "bridgedWhisker";

// Handle critical pairs
string pairKey = uiScript.CreatePairKey(conductor1, conductor2);
if (uiScript.criticalPairs.Contains(pairKey))
{
    // If this is a critical pair, save it separately
    SaveCriticalBridgedWhiskerData(data);
}
}

//Calculates resistances
private float CalculateResistance(float length, float diameter, UIScript.MaterialProperties
materialProps)
{
    float area = Mathf.PI * Mathf.Pow((diameter * 1000) / 2, 2);
    return materialProps.resistivity * (length * 1000) / area;
}

//Saves bridged whisker datas into the CSV files.
private void SaveBridgedWhiskerData(WhiskerData data)
{
    // Save to the custom CSV file specified by directoryPath and fileName
    string directoryPath = uiScript.whiskerControl.directoryPath;

```

```

        string customFilePath = Path.Combine(directoryPath, uiScript.whiskerControl.fileName +
        ".csv");

        // Save to the bridgeOutput.csv file in Application.dataPath
        string fixedFilePath = Path.Combine(Application.dataPath, "bridgeOutput.csv");

        // Save to both files
        SaveBridgedWhiskerDataToFile(data, customFilePath);
        SaveBridgedWhiskerDataToFile(data, fixedFilePath);
    }

    // Helper method to save data to a specified file
    private void SaveBridgedWhiskerDataToFile(WhiskerData data, string filePath)
    {
        try
        {
            if (!Directory.Exists(Path.GetDirectoryName(filePath)))
            {
                Directory.CreateDirectory(Path.GetDirectoryName(filePath));
            }

            var lines = File.Exists(filePath) ? File.ReadAllLines(filePath).ToList() : new
            List<string>();

            // If the file is empty, write the headers
            if (lines.Count == 0)
            {
                // Write the column titles
                lines.Add("All Whiskers,,,,,Bridged Whiskers");
                lines.Add("Whisker #,Length (um),Width (um),Resistance (ohm),Iteration,Whisker
                #,Length (um),Width (um),Resistance (ohm),Iteration,Conductor 1,Conductor 2");
            }

            // Ensure each line has at least 12 columns
            for (int i = 0; i < lines.Count; i++)
            {
                var columns = lines[i].Split(',').ToList();
                while (columns.Count < 12)
                {
                    columns.Add(string.Empty);
                }
                lines[i] = string.Join(",", columns);
            }

            // Find the line where the whisker number matches and insert bridged data
            bool dataInserted = false;

```



```

// The bridged whisker number
int bridgedWhiskerNumber = data.WhiskerNumber;

// Start from line 2 because lines 0 and 1 are headers
for (int i = 2; i < lines.Count; i++)
{
    var columns = lines[i].Split(',').ToList();

    // Check if the All Whiskers column has this whisker number
    if (columns[0] == bridgedWhiskerNumber.ToString())
    {
        // Fill in the bridged whisker data starting from column index 5
        columns[5] = data.WhiskerNumber.ToString();
        columns[6] = data.Length.ToString();
        columns[7] = data.Diameter.ToString();
        columns[8] = data.Resistance.ToString();
        columns[9] = data.SimulationIndex.ToString();
        columns[10] = data.Conductor1;
        columns[11] = data.Conductor2;

        lines[i] = string.Join(",", columns);
        dataInserted = true;
        break;
    }
}

if (!dataInserted)
{
    // If no matching whisker number is found, append a new line with empty All Whiskers
data
    string newLine =
    $",,,,,{data.WhiskerNumber},{data.Length},{data.Diameter},{data.Resistance},{data.Simulation
Index},{data.Conductor1},{data.Conductor2}";
    lines.Add(newLine);
}

// Write the updated lines back to the file
File.WriteAllLines(filePath, lines);

Debug.Log($"Bridged whisker data saved successfully to {filePath}");
}
catch (Exception ex)
{
    Debug.LogError($"Failed to save bridged whisker data to {filePath}: {ex.Message}");
}

```

```

    }

    // Saves the directory/filename
    public void SaveButtonClicked()
    {
        directoryPath = filePathInputField.text;
        fileName = fileNameInputField.text;
    }

    private void SaveCriticalBridgedWhiskerData(WhiskerData data) //crit pairs UI
    {
        // Add to DataManager's critical bridged whiskers list
        DataManager.Instance.criticalBridgedWhiskersData.Add(data);

        // Optionally, you can log or handle critical bridged whiskers here
        Debug.Log($"Critical bridged whisker detected: {data.WhiskerNumber}");
    }

    private string CleanConductorName(string name)
    {
        return name.Replace("_ColliderCopy", "");
    }
}

```

WhiskerData.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class WhiskerData : MonoBehaviour
{
    public int WhiskerNumber { get; set; }
    public float Length { get; set; }
    public float Width { get; set; } // For all whiskers
    public float Diameter { get; set; } // For bridged whiskers
    public float Volume { get; set; } // For all whiskers
    public float Mass { get; set; } // For all whiskers
    public float Resistance { get; set; }
    public int Iteration { get; set; } // For all whiskers
    public int SimulationIndex { get; set; } // For bridged whiskers
    public string Conductor1 { get; set; } // For bridged whiskers
    public string Conductor2 { get; set; } // For bridged whiskers

    // Constructor for all whiskers
    public WhiskerData(int whiskerNumber, float length, float width, float volume, float mass,
        float resistance, int iteration)
    {

```

```

        WhiskerNumber = whiskerNumber;
        Length = length;
        Width = width;
        Volume = volume;
        Mass = mass;
        Resistance = resistance;
        Iteration = iteration;
    }

    // Constructor for bridged whiskers
    public WhiskerData(int whiskerNumber, float length, float diameter, float resistance, int
simulationIndex, string conductor1, string conductor2)
    {
        WhiskerNumber = whiskerNumber;
        Length = length;
        Diameter = diameter;
        Resistance = resistance;
        SimulationIndex = simulationIndex;
        Conductor1 = conductor1;
        Conductor2 = conductor2;
    }
}

```

CircuitBoardFinder.cs

```

using UnityEngine;
using TMPro;
using UnityEngine.UI;

public class CircuitBoardFinder : MonoBehaviour
{
    // References to the TMP input fields for rotation
    public TMP_InputField CircuitRotateX;
    public TMP_InputField CircuitRotateY;
    public TMP_InputField CircuitRotateZ;

    // References to the TMP input fields for spin rates
    public TMP_InputField SpinRateX;
    public TMP_InputField SpinRateY;
    public TMP_InputField SpinRateZ;

    // Reference to the RotateSpinToggle for resetting
    public Toggle RotateSpinToggle;

    private Quaternion initialRotation;
    public float xSpinRate, ySpinRate, zSpinRate;
    private GameObject circuitBoard;
}

```

```

private Coroutine spinCoroutine;

public GameObject basePlane;

void Start()
{
    circuitBoard = GameObject.FindGameObjectWithTag("CircuitBoard");
    if (circuitBoard != null)
    {
        initialRotation = circuitBoard.transform.rotation;
    }

    // Add listeners to input fields
    CircuitRotateX.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
    CircuitRotateY.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
    CircuitRotateZ.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
    SpinRateX.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
    SpinRateY.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
    SpinRateZ.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
    RotateSpinToggle.onValueChanged.AddListener(delegate { UpdateCircuitBoard(); });
}

public void UpdateCircuitBoard()
{
    ResetCircuitBoard(); // Reset before updating

    if (circuitBoard != null)
    {
        // Get the rotation values
        float xRotation = GetInputValue(CircuitRotateX);
        float yRotation = GetInputValue(CircuitRotateY);
        float zRotation = GetInputValue(CircuitRotateZ);

        // Set the rotation of the circuitBoard
        circuitBoard.transform.rotation = Quaternion.Euler(xRotation, yRotation, zRotation);
        Debug.Log($"CircuitBoard rotated to: X={xRotation}, Y={yRotation}, Z={zRotation}");

        // Get the spin rates
        xSpinRate = GetInputValue(SpinRateX);
        ySpinRate = GetInputValue(SpinRateY);
        zSpinRate = GetInputValue(SpinRateZ);

        // Start spinning the circuitBoard
        if (spinCoroutine != null) StopCoroutine(spinCoroutine);
        spinCoroutine = StartCoroutine(SpinBasePlane(basePlane, xSpinRate, ySpinRate,
zSpinRate));
    }
}

```

```

    }
    else
    {
        Debug.Log("No CircuitBoard found.");
    }
}

public float GetInputValue(TMP_InputField inputField)
{
    if (inputField == CircuitRotateX && string.IsNullOrEmpty(inputField.text))
    {
        return -90f; // Default to -90 if CircuitRotateX is empty
    }

    if (float.TryParse(inputField.text, out float result))
    {
        return result;
    }

    return 0f; // Default to 0 if input is invalid for other fields
}

private System.Collections.IEnumerator SpinBasePlane(GameObject basePlane , float
xSpinRate, float ySpinRate, float zSpinRate)
{
    while (true)
    {
        basePlane.transform.Rotate(xSpinRate * Time.deltaTime, ySpinRate * Time.deltaTime,
zSpinRate * Time.deltaTime);
        yield return null; // Wait for the next frame
    }
}

public void ResetCircuitBoard()
{
    if (RotateSpinToggle != null && RotateSpinToggle.isOn)
    {
        float xRotation = GetInputValue(CircuitRotateX);
        float yRotation = GetInputValue(CircuitRotateY);
        float zRotation = GetInputValue(CircuitRotateZ);

        circuitBoard.transform.rotation = Quaternion.Euler(xRotation, yRotation, zRotation);
        circuitBoard.transform.position = Vector3.zero; // Reset position if needed
        basePlane.transform.position= Vector3.zero;
    }
}

```

```

        if (spinCoroutine != null) StopCoroutine(spinCoroutine);
        Debug.Log("CircuitBoard and BasePlane reset to initial rotation and spin.");
    }
}
}

```

csvReader.cs

```

using System;
using System.IO;
using System.Linq;
using UnityEngine;
using TMPro;

public class LoadBridgedWhiskerData : MonoBehaviour
{
    public TextMeshProUGUI outputText; // Attach this to the TMP text object in your Scroll
    View
    public RectTransform contentRectTransform; // Reference to the Content RectTransform
    private string filePath;

    void Start()
    {
        // Set the file path to the saved CSV file in the Assets folder
        filePath = Path.Combine(Application.dataPath, "bridgeOutput.csv");
    }

    // Method to load and display CSV data, now public so it can be called by a UI button
    public void LoadCSVData()
    {
        try
        {
            // Check if the file exists
            if (File.Exists(filePath))
            {
                // Read all lines from the CSV file
                string[] lines = File.ReadAllLines(filePath);

                // Check for empty file or insufficient rows
                if (lines.Length < 2)
                {
                    outputText.text = "CSV file is empty or missing header row.";
                    return;
                }

                // Parse header row (second row) and calculate column widths based on header cells
                string[] header = lines[1].Split(',');
            }
        }
    }
}

```

```

int[] columnWidths = header.Select(h => h.Length).ToArray();

// Get the total width in characters that the TextMeshProUGUI can display
float totalTextWidth = outputText.rectTransform.rect.width;
float characterWidth = outputText.fontSize * 0.5f; // Estimate character width as half
of the font size
int totalColumnsWidth = Mathf.FloorToInt(totalTextWidth / characterWidth);

// Distribute the total width across columns proportionally based on the header
int headerWidthSum = columnWidths.Sum();
for (int i = 0; i < columnWidths.Length; i++)
{
    columnWidths[i] = Mathf.FloorToInt((float)columnWidths[i] / headerWidthSum *
totalColumnsWidth);
}

// Clear the existing text
outputText.text = "";

// Display the header row with center alignment and adjusted column widths
for (int i = 0; i < header.Length; i++)
{
    outputText.text += CenterAlignText(header[i], columnWidths[i]) + "  ";
}
outputText.text += "\n" + new string('-', totalColumnsWidth + (header.Length - 1) * 4)
+ "\n";

// Process each data row starting from the third line and format it based on the
calculated column widths
for (int row = 2; row < lines.Length; row++)
{
    string[] rowData = lines[row].Split(',');

    for (int i = 0; i < columnWidths.Length; i++)
    {
        // Center-align each cell in the row based on header-defined width
        string cellData = i < rowData.Length ? rowData[i] : ""; // Fallback to empty if
data is missing
        outputText.text += CenterAlignText(cellData, columnWidths[i]) + "  ";
    }
    outputText.text += "\n"; // New line for each row
}

// Adjust content size to make it scrollable
AdjustContentHeight();
}

```

```

        else
        {
            outputText.text = "No CSV file found.";
        }
    }
    catch (Exception ex)
    {
        Debug.LogError($"Failed to load CSV data: {ex.Message}");
    }
}

// Method to center-align text by adding spaces to both sides
private string CenterAlignText(string text, int width)
{
    int padding = width - text.Length;
    int padLeft = padding / 2 + text.Length;
    return text.PadLeft(padLeft).PadRight(width);
}

// Adjusts the content height based on the text size
private void AdjustContentHeight()
{
    // Reset the Content position to align with the top of the Viewport
    contentRectTransform.anchoredPosition = Vector2.zero;

    // Calculate the total preferred height of the text in outputText
    float textHeight = outputText.preferredHeight;

    // Set the Content height to match the height of the text
    contentRectTransform.sizeDelta = new Vector2(contentRectTransform.sizeDelta.x,
textHeight);
}
}

```

DropDownHandler.cs

```

using UnityEngine;
using UnityEngine.UI;
using TMPro; // Include this for TextMeshPro

public class DropdownHandler : MonoBehaviour
{
    public Dropdown dropdown; // Reference to your TMP Dropdown
    public InputField inputFieldX; // Reference to your TMP Input Field

    public InputField inputFieldY; // Reference to your TMP Input Field
}

```



```

public InputField inputFieldZ; // Reference to your TMP Input Field

private void Start()
{
    // Initialize the Input Field as disabled
    inputFieldX.gameObject.SetActive(false);
    inputFieldY.gameObject.SetActive(false);
    inputFieldZ.gameObject.SetActive(false);

    // Add listener for when the dropdown value changes
    dropdown.onValueChanged.AddListener(OnDropdownValueChanged);
}

private void OnDropdownValueChanged(int index)
{
    // Assuming the option you want to check is at index 1
    if (index == 3) // Change this to the index of your desired option
    {
        inputFieldX.gameObject.SetActive(true); // Show the input field
        inputFieldY.gameObject.SetActive(true);
        inputFieldZ.gameObject.SetActive(true);
    }
    else
    {
        inputFieldX.gameObject.SetActive(false); // Hide the input field
        inputFieldY.gameObject.SetActive(false);
        inputFieldZ.gameObject.SetActive(false);
    }
}
}

```

InputFieldToFloat.cs

```

using UnityEngine;
using UnityEngine.UI;

public class InputFieldToFloat : MonoBehaviour
{
    public InputField inputField; // Drag your Input Field here in the inspector
    public float result; // This will hold the converted float

    public void ConvertInputToFloat()
    {
        // Try to parse the text from the Input Field
        if (float.TryParse(inputField.text, out result))
        {
            Debug.Log("Converted to float: " + result);
        }
    }
}

```

```

    }
    else
    {
        Debug.Log("Invalid input, please enter a valid float.");
    }
}
}

```

PositionUpdate.cs

```

using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class PositionUpdater : MonoBehaviour
{
    public TMP_InputField WhiskerSpawnPointX;
    public TMP_InputField WhiskerSpawnPointY;
    public TMP_InputField WhiskerSpawnPointZ;
    public GameObject WhiskerSpawnPoint;

    void Start()
    {
        // Add listener to each input field to call UpdatePosition on value change
        WhiskerSpawnPointX.onValueChanged.AddListener(delegate { UpdatePosition(); });
        WhiskerSpawnPointY.onValueChanged.AddListener(delegate { UpdatePosition(); });
        WhiskerSpawnPointZ.onValueChanged.AddListener(delegate { UpdatePosition(); });
    }

    void UpdatePosition()
    {
        // Try to parse input field values to float
        if (float.TryParse(WhiskerSpawnPointX.text, out float x) &&
            float.TryParse(WhiskerSpawnPointY.text, out float y) &&
            float.TryParse(WhiskerSpawnPointZ.text, out float z))
        {
            // Set the position of the object
            WhiskerSpawnPoint.transform.position = new Vector3(x*10, y*10, z*10);
        }
    }
}

```

TabControllerFront.cs

```

using UnityEngine;

```

```

using UnityEngine.UI;

public class TabControllerFront : MonoBehaviour
{
    public RectTransform tab; // The RectTransform of the tab you want to move
    public Button moveButton; // The button that triggers the movement
    public Vector2 targetPosition; // The position to move the tab to

    private void Start()
    {
        // Ensure the button has a listener for the click event
        moveButton.onClick.AddListener(MoveTab);
    }

    private void MoveTab()
    {
        // Move the tab to the specified position
        tab.anchoredPosition = targetPosition;

        // Bring the tab to the front by setting its sibling index
        tab.SetAsLastSibling();
    }
}

```

TabControllerStay.cs

```

using UnityEngine;
using UnityEngine.UI;

public class TabControllerStay : MonoBehaviour
{
    public RectTransform tab; // The RectTransform of the tab you want to move
    public Button moveButton; // The button that triggers the movement
    public Vector2 targetPosition; // The position to move the tab to

    private void Start()
    {
        // Ensure the button has a listener for the click event
        moveButton.onClick.AddListener(MoveTab);
    }

    private void MoveTab()
    {
        // Move the tab to the specified position
        tab.anchoredPosition = targetPosition;
    }
}

```

WhiskerCubeScaler.cs

```
using UnityEngine;
using TMPro; // Include this namespace for TextMeshPro

public class MovePlaneTMP : MonoBehaviour
{
    public TMP_InputField inputFieldX; // Assign this in the Inspector
    public TMP_InputField inputFieldY; // Assign this in the Inspector
    public TMP_InputField inputFieldZ; // Assign this in the Inspector

    void Update()
    {
        // Check if all input fields are not empty
        if (!string.IsNullOrEmpty(inputFieldX.text) &&
            !string.IsNullOrEmpty(inputFieldY.text) &&
            !string.IsNullOrEmpty(inputFieldZ.text))
        {
            // Try to parse the text as floats
            if (float.TryParse(inputFieldX.text, out float xValue) &&
                float.TryParse(inputFieldY.text, out float yValue) &&
                float.TryParse(inputFieldZ.text, out float zValue))
            {
                // Update the plane's position
                transform.localScale = new Vector3(xValue * 20f, yValue * 10f, zValue * 20f);
                transform.localPosition = new Vector3 (0,yValue/.2f ,0);
            }
        }
    }

    public void ClearInputs()
    {
        // Optionally, clear the input fields when needed
        inputFieldX.text = string.Empty;
        inputFieldY.text = string.Empty;
        inputFieldZ.text = string.Empty;
    }
}
```

WhiskerSpawnCubeVisibility.cs

```
using UnityEngine;
using UnityEngine.UI;

public class ToggleVisibility : MonoBehaviour
```

```

{
    public GameObject cube; // Reference to the cube
    public Toggle toggle; // Reference to the toggle

    void Start()
    {
        // Ensure the toggle's on value is synced with the cube's initial visibility
        toggle.isOn = cube.activeSelf;
        toggle.onValueChanged.AddListener(OnToggleChanged);
    }

    void OnToggleChanged(bool isOn)
    {
        cube.SetActive(isOn); // Set the cube's active state based on toggle value
    }
}

```

BoardDetector.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using TMPro;

public class BoardDetector : MonoBehaviour
{
    public GameObject detectedBoard { get; private set; } // Stores the detected board

    private IEnumerator Start()
    {
        yield return new WaitForEndOfFrame();

        // Find the imported model tagged as "CircuitBoard"
        GameObject importedModel = GameObject.FindWithTag("CircuitBoard");

        if (importedModel != null)
        {
            // Immediately set BasePlane as the parent of the imported model
            GameObject basePlane = GameObject.Find("BasePlane");
            if (basePlane != null)
            {
                importedModel.transform.SetParent(basePlane.transform);
                Debug.Log("CircuitBoard parent set to BasePlane.");
            }
        }
        else
        {

```

```

        Debug.LogError("BasePlane not found.");
    }

    // Now find and process the detected board
    detectedBoard = FindBoard(importedModel);

    if (detectedBoard != null)
    {
        Debug.Log("Board detected: " + detectedBoard.name);
        Renderer boardRenderer = detectedBoard.GetComponent<Renderer>();
        // Uncomment if you need to log bounds
        // Debug.Log("Detected Board Bounds: " + boardRenderer.bounds);

        GameObject whiskerSpawnPoint = GameObject.Find("WhiskerSpawnPoint");
        if (whiskerSpawnPoint != null)
        {
            whiskerSpawnPoint.transform.SetParent(detectedBoard.transform);
        }
        else
        {
            Debug.LogError("WhiskerSpawnPoint not found.");
        }
    }
    else
    {
        Debug.LogError("Board could not be detected.");
    }
}
else
{
    Debug.LogError("No GameObject with tag 'CircuitBoard' found.");
}
}

// Method to find the board based on largest object in heirachy
private GameObject FindBoard(GameObject rootObject)
{
    GameObject largestFlatObject = null;
    float largestSurfaceArea = 0;

    // Iterate over all the children of the imported model (all the components)
    foreach (Renderer renderer in rootObject.GetComponentsInChildren<Renderer>())
    {
        Bounds bounds = renderer.bounds;

        //calculates the surface area

```

```

float surfaceArea = bounds.size.x * bounds.size.z;

//Debug.Log($"Checking Object: {renderer.gameObject.name}, Surface Area:
{surfaceArea}, Thickness (Y): {bounds.size.y}, Center Y: {bounds.center.y}");
//check if the object is relatively flat by comparing its height (Y) with its surface area
// Example: thickness should be relatively small compared to surface area
if (surfaceArea > largestSurfaceArea && bounds.size.y < (0.3f *
Mathf.Min(bounds.size.x, bounds.size.z)))
{
    largestSurfaceArea = surfaceArea;
    largestFlatObject = renderer.gameObject;

    //Debug.Log($"Potential Board Detected: {largestFlatObject.name}, Surface Area:
{surfaceArea}, thickness (Y): {bounds.size.y}");
}
}

if (largestFlatObject != null)
{
    //Debug.Log("Largest Flat Object Detected: " + largestFlatObject.name);
}
else
{
    Debug.LogWarning("No flat object detected as the board.");
}
return largestFlatObject;
}
}

```

WallCreator.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using TMPro;

public class WallCreator : MonoBehaviour
{
    public GameObject wallPrefab; //prefab for the wall
    public GameObject ceilingPrefab; //prefab for ceiling
    public Button wallButton; //reference to the create walls button
    public TMP_InputField wallHeightInput; //Reference to the input field for wall height
    public Toggle ceilingToggle; //Reference to the toggle button for the ceiling
    public Material transparentWallMaterial;

    private BoardDetector boardDetector; //to detect the board

```

```

private GameObject[] currentWalls; //stores the currently created walls
private GameObject currentCeiling; //stores currently created ceiling

private void Start()
{
    // find the button and add a listener for the onClick event
    if (wallButton != null)
    {
        wallButton.onClick.AddListener(OnWallButtonPressed);
    }

    // Find the BoardDetector component in the scene
    boardDetector = GetComponent<BoardDetector>();

    if (boardDetector == null)
    {
        Debug.LogError("BoardDetector component is missing!");
    }
}

//called when the button is pressed
public void OnWallButtonPressed()
{
    DeleteExistingWallsAndCeiling();

    //Get the detected board from the BoardDetector script
    GameObject detectedBoard = boardDetector.detectedBoard;

    if (detectedBoard != null)
    {
        Renderer boardRenderer = detectedBoard.GetComponent<Renderer>();
        Bounds boardBounds = boardRenderer.bounds;

        //Debug.Log("Detected Board for Wall Placement: " + detectedBoard.name);
        //Debug.Log("detected Board Bounds for Wall PLacement: " + boardBounds);

        // The input from the text box to get the wall height
        float wallHeight;
        if (float.TryParse(wallHeightInput.text, out wallHeight))
        {
            // creates walls around the detected board
            CreateWalls(boardBounds, detectedBoard, wallHeight);

            //check if ceiling toggle is on
            if (ceilingToggle.isOn)
            {

```



```

        CreateCeiling(boardBounds, detectedBoard, wallHeight);
    }
}
else
{
    Debug.LogError("Invalid input for wall height. Please enter a valid number.");
}
}
else
{
    Debug.LogError(" No board detected.");
}
}

```

```

private void CreateWalls(Bounds boardBounds, GameObject detectedBoard, float wallHeight)
{
    Vector3 boardSize = boardBounds.size;
    Vector3 boardCenter = boardBounds.center;

    //ensures bottom of walls align with bottom of the board
    float wallBottomY = boardBounds.min.y;

    // Log the wall position and size
    //Debug.Log($"Creating Walls at Bottom Y: {wallBottomY}, Wall Height: {wallHeight}");

    // Wall dimensions
    float wallThickness = 0.1f;

    // positions and scales the walls
    Vector3 leftWallPos = new Vector3(boardCenter.x - (boardSize.x / 2) - (wallThickness / 2),
wallBottomY + (wallHeight / 2), boardCenter.z);
    Vector3 rightWallPos = new Vector3(boardCenter.x + (boardSize.x / 2) + (wallThickness /
2), wallBottomY + (wallHeight / 2), boardCenter.z);
    Vector3 frontWallPos = new Vector3(boardCenter.x, wallBottomY + (wallHeight / 2),
boardCenter.z + (boardSize.z / 2) + (wallThickness / 2));
    Vector3 backWallPos = new Vector3(boardCenter.x, wallBottomY + (wallHeight / 2),
boardCenter.z - (boardSize.z / 2) - (wallThickness / 2));

    // Creates an array to hold references for the new walls
    currentWalls = new GameObject[4];

    // Instantiate walls
    currentWalls[0] = CreateWall(leftWallPos, new Vector3(wallThickness, wallHeight,
boardSize.z), detectedBoard);

```

```

        currentWalls[1] = CreateWall(rightWallPos, new Vector3(wallThickness, wallHeight,
boardSize.z), detectedBoard);
        currentWalls[2] = CreateWall(frontWallPos, new Vector3(boardSize.x, wallHeight,
wallThickness), detectedBoard);
        currentWalls[3] = CreateWall(backWallPos, new Vector3(boardSize.x, wallHeight,
wallThickness), detectedBoard);
    }

    private void CreateCeiling(Bounds boardBounds, GameObject detectedBoard, float
wallHeight)
    {
        Vector3 boardSize = boardBounds.size;
        Vector3 boardCenter = boardBounds.center;

        float ceilingThickness = 0.1f;

        //Positions ceiling at the top of the walls
        float ceilingY = boardBounds.min.y + wallHeight + (ceilingThickness / 2);

        //Debug.Log($"Creating Ceiling at Y: {ceilingY}, Board Center: {boardCenter}");

        Vector3 ceilingPos = new Vector3(boardCenter.x, ceilingY, boardCenter.z);

        // Create the ceiling
        currentCeiling = Instantiate(ceilingPrefab, ceilingPos, Quaternion.identity);
        currentCeiling.transform.localScale = new Vector3(boardSize.x, 0.1f, boardSize.z);
//adjusts size and thickness
        currentCeiling.transform.SetParent(detectedBoard.transform); //attaches ceiling to the board

        // Apply the transparent material to the ceiling
        Renderer ceilingRenderer = currentCeiling.GetComponent<Renderer>();
        if (ceilingRenderer != null)
        {
            ceilingRenderer.material = transparentWallMaterial;
        }
    }

    private GameObject CreateWall(Vector3 position, Vector3 scale, GameObject detectedBoard)
    {
        GameObject wall = Instantiate(wallPrefab, position, Quaternion.identity);
        wall.transform.localScale = scale;
        wall.transform.SetParent(detectedBoard.transform); //attaches wall to the board

        //Apply transparent material to the wall
        Renderer wallrenderer = wall.GetComponent<Renderer>();
        if (wallrenderer != null)

```

```

    {
        wallrenderer.material = transparentWallMaterial;
    }
    return wall; //return reference to newly created wall
}

private void DeleteExistingWallsAndCeiling()
{
    //Checks for existing walls
    if (currentWalls != null)
    {
        // Loop through the walls and destroy them
        for (int i=0; i < currentWalls.Length; i++)
        {
            if (currentWalls[i] != null)
            {
                Destroy(currentWalls[i]);
            }
        }
    }

    //check and delete the ceiling if it exists
    if (currentCeiling != null)
    {
        Destroy(currentCeiling);
    }
}
}

```

SaveManager.cs

```

using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;

public class SaveManager : MonoBehaviour
{
    private string saveFilePath;

    private void Awake()
    {
        //set the file path to save the input data
        saveFilePath = Path.Combine(Application.dataPath + "/simulationInputData.json");
    }
}

```

```

public void SaveSimulationData(SimulationData data)
{
    string json = JsonUtility.ToJson(data);
    File.WriteAllText(saveFilePath, json);
    Debug.Log("Simulation data saved to " + saveFilePath);
}

public SimulationData LoadSimulationData()
{
    if (File.Exists(saveFilePath))
    {
        string json = File.ReadAllText(saveFilePath);
        SimulationData data = JsonUtility.FromJson<SimulationData>(json);
        Debug.Log("Simulation data loaded from " + saveFilePath);
        return data;
    }
    else
    {
        Debug.LogWarning("No saved simulation data found at " + saveFilePath);
        return null;
    }
}
}

```

SimulationController.cs

```

using System.Collections;
using System.Collections.Generic;
using TMPro;
using UnityEngine;
using UnityEngine.Rendering;

public class SimulationController : MonoBehaviour
{
    public TMP_InputField lengthMuInput;
    public TMP_InputField widthMuInput;
    public TMP_InputField lengthSigmaInput;
    public TMP_InputField widthSigmaInput;
    public TMP_InputField numWhiskersInput;
    public TMP_InputField numIterationsInput;
    public TMP_InputField xCoordInput;
    public TMP_InputField yCoordInput;
    public TMP_InputField zCoordInput;
    public TMP_Dropdown whiskerMatDropdown;
    public TMP_Dropdown distributionDropdown;

    public SaveManager saveManager;
}

```

```

private SimulationData currentSimulationData;

public WhiskerControl whiskerControl; //reference to get gravity from WhiskerControl

private void Start()
{
    // Try to find SaveManager if not set in the Inspector
    if (saveManager == null)
    {
        saveManager = FindObjectOfType<SaveManager>();
        if (saveManager == null)
        {
            Debug.LogError("SaveManager not found in the scene. Make sure there is a
GameObject with the SaveManager script attached.");
            return; // Exit early to avoid null reference
        }
    }

    currentSimulationData = new SimulationData();

    // Check if saveManager is still null
    if (saveManager != null)
    {
        LoadSettings();
    }
    else
    {
        Debug.LogError("SaveManager is null. Cannot load settings.");
    }
}

public void SaveSettings()
{
    if (currentSimulationData == null)
    {
        Debug.LogError("currentSimulationData is null.");
        return;
    }

    // Update current simulation data from UI inputs
    currentSimulationData.lengthMu = float.Parse(lengthMuInput.text);
    currentSimulationData.widthMu = float.Parse(widthMuInput.text);
    currentSimulationData.lengthSigma = float.Parse(lengthSigmaInput.text);
    currentSimulationData.widthSigma = float.Parse(widthSigmaInput.text);
    currentSimulationData.numWhiskers = int.Parse(numWhiskersInput.text);
    currentSimulationData.numIterations = int.Parse(numIterationsInput.text);
}

```

```

currentSimulationData.xCoord = float.Parse(xCoordInput.text);
currentSimulationData.yCoord = float.Parse(yCoordInput.text);
currentSimulationData.zCoord = float.Parse(zCoordInput.text);
currentSimulationData.whiskMat = whiskerMatDropdown.value;
currentSimulationData.distributionDropdown = distributionDropdown.value;
currentSimulationData.gravity = whiskerControl.gravity.value;

if (saveManager != null)
{
    saveManager.SaveSimulationData(currentSimulationData);
}
else
{
    Debug.LogError("SaveManager is null. Cannot save settings.");
}
}

public void LoadSettings()
{
    if (saveManager != null)
    {
        SimulationData loadedData = saveManager.LoadSimulationData();
        if (loadedData != null)
        {
            currentSimulationData = loadedData;
            UpdateUIWithLoadedData(currentSimulationData);
        }
    }
    else
    {
        Debug.LogError("SaveManager is null. Cannot load settings.");
    }
}

private void UpdateUIWithLoadedData(SimulationData data)
{
    lengthMuInput.text = data.lengthMu.ToString();
    widthMuInput.text = data.widthMu.ToString();
    lengthSigmaInput.text = data.lengthSigma.ToString();
    widthSigmaInput.text = data.widthSigma.ToString();
    numWhiskersInput.text = data.numWhiskers.ToString();
    numIterationsInput.text = data.numIterations.ToString();
    xCoordInput.text = data.xCoord.ToString();
    yCoordInput.text = data.yCoord.ToString();
    zCoordInput.text = data.zCoord.ToString();
    whiskerMatDropdown.value = data.whiskMat;
}

```

```

        distributionDropdown.value = data.distributionDropdown;
        whiskerControl.gravity.value = data.gravity;
    }
}

```

SimulationData.cs

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SimulationData
{
    public float lengthMu;
    public float widthMu;
    public float lengthSigma;
    public float widthSigma;
    public int numWhiskers;
    public int numIterations;
    public float xCoord;
    public float yCoord;
    public float zCoord;
    public int whiskMat;
    public int distributionDropdown;
    public int gravity;
}

```

RandomFromDistribution.cs

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

public static class RandomFromDistribution {

    //-----
    // Normal Distribution
    //-----

    public enum ConfidenceLevel_e { _60=0, _80, _90, _95, _98, _99, _998, _999 };

    /// <summary>
    /// Get a random number from a normal distribution in [min,max].
    /// </summary>
    /// <description>

```

```

    /// Get a random number between min [inclusive] and max [inclusive] with probability
matching
    /// a normal distribution along this range. The width of the distribution is described by the
    /// confidence_level_cutoff, which describes what percentage of the bell curve should be
over
    /// the provided range. For example, a confidence level cutoff of 0.999 will result in a bell
    /// curve from min to max that contains 99.9% of the area under the complete curve. 0.80
gives
    /// a curve with 80% of the distribution's area.
    /// Because a normal distribution flattens out towards the ends, this means that 0.80 will
have
    /// more even distribution between min and max than 0.999.
    /// </description>
    /// <returns>
    /// A random number between min [inclusive] and max [inclusive], with probability
described
    /// by the distribution.
    /// </returns>
    /// <param name="min">The min value returned [inclusive].</param>
    /// <param name="max">The max min value returned [inclusive].</param>
    /// <param name="confidence_level_cutoff">
    /// The percentage of a standard normal distribution that should be represented in the
range.
    /// </param>
    public static float RandomRangeNormalDistribution(float min, float max,
        ConfidenceLevel_e confidence_level_cutoff /*, float
confidence_level_cutoff*/) {

        float mean = 0.5f * (min + max);

        // TODO formula for this?
        float z_score_cutoff = confidence_to_z_score[(int)confidence_level_cutoff];

        float new_width = (max - min) / 2.0f;
        float sigma = new_width / z_score_cutoff;

        // Get random normal from Normal Distribution that's within the confidence level
cutoff requested
        float random_normal_num;
        do {
            random_normal_num = RandomNormalDistribution(mean, sigma);

        } while (random_normal_num > max || random_normal_num < min);

        // now you have a number selected from a bell curve stretching from min to max!

```



```

        return random_normal_num;

    }

    /// <summary>
    /// Get a random number from a normal distribution with given mean and standard
deviation.
    /// </summary>
    /// <description>
    /// Get a random number with probability following a normal distribution with given
mean
    /// and standard deviation. The likelihood of getting any given number corresponds to
    /// its value along the y-axis in the distribution described by the parameters.
    /// </description>
    /// <returns>
    /// A random number between -infinity and infinity, with probability described by the
distribution.
    /// </returns>
    /// <param name="mean">The Mean (or center) of the normal distribution.</param>
    /// <param name="std_dev">The Standard Deviation (or Sigma) of the normal
distribution.</param>
    public static float RandomNormalDistribution(float mean, float std_dev) {

        // Get random normal from Standard Normal Distribution
        float random_normal_num = RandomFromStandardNormalDistribution();

        // Stretch distribution to the requested sigma variance
        random_normal_num *= std_dev;

        // Shift mean to requested mean:
        random_normal_num += mean;

        // now you have a number selected from a normal distribution with requested
mean and sigma!
        return random_normal_num;

    }

    /// <summary>
    /// Get a random number from the standard normal distribution.
    /// </summary>
    /// <returns>
    /// A random number in range [-inf, inf] from the standard normal distribution (mean ==
1, stand deviation == 1).
    /// </returns>

```

```

public static float RandomFromStandardNormalDistribution() {

    // This code follows the polar form of the muller transform:
    // https://en.wikipedia.org/wiki/Box%E2%80%93Muller_transform#Polar_form
    // also known as Marsaglia polar method
    // https://en.wikipedia.org/wiki/Marsaglia_polar_method

    // calculate points on a circle
    float u, v;

    float s; // this is the hypotenuse squared.
    do {
        u = Random.Range (-1f, 1f);
        v = Random.Range (-1f, 1f);
        s = (u * u) + (v * v);
    } while (!(s != 0 && s < 1)); // keep going until s is nonzero and less than one

    // TODO allow a user to specify how many random numbers they want!
    // choose between u and v for seed (z0 vs z1)
    float seed;
    if (Random.Range(0,2) == 0) {
        seed = u;
    }
    else {
        seed = v;
    }

    // create normally distributed number.
    float z = seed * Mathf.Sqrt(-2.0f * Mathf.Log(s) / s);

    return z;
}

private static float[] confidence_to_z_score = {
    0.84162123f,
    1.28155156f,
    1.64485363f,
    1.95996399f,
    2.32634787f,
    2.57582931f,
    3.0902323f,
    3.29052673f
};

```

```

//-----
// Sloped Distribution (sec^2 distribution)
//-----

public enum Direction_e { Right, Left };

/// <summary>
/// Returns a random number in range [min,max] from a curved slope following sec^2(x).
/// </summary>
/// <returns>Random in range [min,max] from a curved left slope.</returns>
/// <param name="skew">The difference in height between max and min of
curve.</param>
public static float RandomRangeSlope(float min, float max, float skew, Direction_e
direction) {
    return min + RandomFromSlopedDistribution(skew, direction) * (max-min);
}

/// <summary>
/// Returns random in range [0,1] from a curved right slope.
/// </summary>
/// <returns>Random in range [0,1] from a curved right slope.</returns>
/// <param name="skew">The difference in height between max and min of
curve.</param>
public static float RandomFromSlopedDistribution(float skew, Direction_e direction) {

    // the difference in scale is just the same as the max y-value..
    float max_y = skew;

    // our curve will go from 0 to max_x.
    float max_x = Inverse_Sec_Sqrd(max_y);

    float max_cdf = Sec_Sqrd_CumulativeDistributionFunction(max_x);

    float u = Random.Range(0.0f, max_cdf);
    float x_val = Sec_Sqrd_InverseCumulativeDistributionFunction(u);

    // scale to [0,1]
    float value = x_val / max_x;

    if (direction == Direction_e.Left) {
        value = 1.0f - value;
    }
}

```

```

        return value;
    }

    /// <summary>
    /// The inverse of the sec^2 function.
    /// </summary>
    /// <param name="y">The y coordinate. if y < 1, returns NaN. </param>
    private static float Inverse_Seqd(float y) {

        // Note: arcsec(x) = arccos(1/x)

        // return arcsec(sqrt(y))
        return Mathf.Acos(1.0f / Mathf.Sqrt(y));
    }

    // The integral of sec^2
    private static float Seqd_CumulativeDistributionFunction(float x) {

        // The cumulative distribution function for sec^2 is just the definite integral of
        sec^2(x) = tan(x) - tan(0) = tan(x)

        return Mathf.Tan(x);
    }

    // The inverse of the integral of sec^2
    private static float Seqd_InverseCumulativeDistributionFunction(float x) {

        // The cumulative distribution function for sec^2 is just the definite integral of
        sec^2(x) = tan(x) - tan(0) = tan(x)
        // Then the Inverse cumulative distribution function is just atan(x)

        return Mathf.Atan(x);
    }

    //-----
    // Linear Distribution
    //-----

    // Returns random in range [min, max] with linear distribution of given slope.
    public static float RandomRangeLinear(float min, float max, float slope) {

        float val = RandomLinear(slope);

```

```

        return min + (max-min) * val;
    }

    // Returns random in range [0,1] with linear distribution of given slope.
    public static float RandomLinear(float slope) {

        float abs_value = RandomFromLinearWithPositiveSlope(Mathf.Abs(slope));
        if (slope < 0) {
            return 1 - abs_value;
        }
        else {
            return abs_value;
        }
    }
}

```

```

    // Returns random in range [0,1] with linear distribution of given slope.
    private static float RandomFromLinearWithPositiveSlope(float slope) {

        if (slope == 0) {
            return Random.Range(0.0f, 1.0f);
        }

        float x, y;
        do {
            x = Random.Range(0.0f, 1.0f);
            y = Random.Range(0.0f, 1.0f);
            if (slope < 1) {
                y -= (1 - slope) / 2.0f;
            }
        } while (y > x * slope);

        return x;
    }
}

```

```

//-----
// Exponential Distribution
//-----

```

```

/// <summary>
/// Returns a random number in range [min,max] from an exponential distribution.
/// </summary>

```

```

    /// <returns>Random number in range [min,max] from given exponential
distribution.</returns>
    /// <param name="min">Minimum random number (inclusive).</param>
    /// <param name="max">Maximum random number (inclusive).</param>
    /// <param name="exponent">
    /// Exponent for distribution. Must be >= 0.
    /// 0 will be uniform distribution; 1 will be linear distribution w/ slope 1.
    /// </param>
    /// <param name="direction">The direction for the curve (right/left).</param>
    public static float RandomRangeExponential(float min, float max, float exponent,
Direction_e direction) {
        return min + RandomFromExponentialDistribution(exponent, direction) * (max -
min);
    }

    /// <summary>
    /// Returns a random number in range [0,1] from an exponential distribution.
    /// </summary>
    /// <returns>Random number in range [0,1] from given exponential
distribution.</returns>
    /// <param name="exponent">
    /// Exponent for distribution. Must be >= 0.
    /// 0 will be uniform distribution; 1 will be linear distribution w/ slope 1.
    /// </param>
    /// <param name="direction">The direction for the curve (right/left).</param>
    public static float RandomFromExponentialDistribution(float exponent, Direction_e
direction) {

        // our curve will go from 0 to 1.
        float max_cdf = ExponentialRightCDF(1.0f, exponent);

        float u = Random.Range(0.0f, max_cdf);
        float x_val = EponentialRightInverseCDF(u, exponent);

        if (direction == Direction_e.Left) {
            x_val = 1.0f - x_val;
        }

        return x_val;
    }

    // The inverse of the curve.
    private static float ExponentialRightInverse(float y, float exponent) {

        return Mathf.Pow(y, 1.0f/exponent);
    }

```

```

    }

    // The integral of the exponent curve.
    private static float ExponentialRightCDF(float x, float exponent) {

        float integral_exp = exponent+1.0f;
        return (Mathf.Pow(x, integral_exp)) / integral_exp;
    }

    // The inverse of the integral of the exponent curve.
    private static float EponentialRightInverseCDF(float x, float exponent) {

        float integral_exp = exponent+1.0f;
        return Mathf.Pow(integral_exp * x, 1.0f/integral_exp);
    }

    //-----
    // User-Defined Probability Distribution Function
    //-----

    /// <summary>
    /// Return an index randomly chosen following the distribution specified by a list of
probabilities
    /// </summary>
    /// <returns>
    /// An index in range [0, probabilities.Length) following the distribution specified in
probabilites.
    /// </returns>
    /// <param name="probabilities">
    /// A list of probabilities from which to choose an index. All values must be >= 0!
    /// </param>
    public static int RandomChoiceFollowingDistribution(List<float> probabilities) {

        // Sum to create CDF:
        float[] cdf = new float[probabilities.Count];
        float sum = 0;
        for (int i = 0; i < probabilities.Count; ++i) {
            cdf[i] = sum + probabilities[i];
            sum = cdf[i];
        }

        // Choose from CDF:
        float cdf_value = Random.Range(0.0f, cdf[probabilities.Count-1]);
        int index = System.Array.BinarySearch(cdf, cdf_value);

```

```
        if (index < 0) {  
            index = ~index;    // if not found (probably won't be) BinarySearch  
returns bitwise complement of next-highest index.  
        }  
  
        return index;  
    }  
}
```


LIST OF REFERENCES

- [1] G. Duke, M. Osborne, A. Smith, and C. Massey, “3D Modeling of Detached Metal Whiskers.” Auburn University, Auburn (accessed May 29, 2024)
- [2] U. Technologies, “GameObject,” Unity Documentation, <https://docs.unity3d.com/Manual/class-GameObject.html> (accessed June 17, 2024).
- [3] M. Sampson and H. Leidecker, “Basic info on tin whiskers,” NASA, <https://nepp.nasa.gov/whisker/background/index.htm> (accessed June 17, 2024).
- [4] “Mechanism of Tin Whisker Growth in Electronics,” Mechanism of tin whisker growth in Electronics, <https://cave.auburn.edu/rsrch-thrusts/mechanism-of-tin-whisker-growth-in-electronics.html> (accessed June 17, 2024).
- [5] iXie, “Top 5 coding languages compatible with the Unity Game Development Engine,” iXie Gaming, <https://www.ixiegaming.com/blog/top-coding-languages-unity-game-development/> (accessed June 17, 2024).
- [6] W. Kenton, “Monte Carlo Simulation: History, how it works, and 4 key steps,” Investopedia, <http://www.investopedia.com/terms/m/montecarlosimulation.asp> (accessed June 17, 2024).
- [7] J. Brusse, H. Leidecker, L. Panashchenko, “The Art of Metal Whisker Appreciation: A Practical Guide for Electronics Professionals” NASA, https://nepp.nasa.gov/whisker/reference/tech_papers/2012-Panashchenko-IPC-Art-of-Metal-Whisker-Appreciation.pdf (accessed June 17, 2024).
- [8] J. Brusse, “Electrical failure of an accelerator pedal position sensor ...,” 2011 NASA-GSFC whisker failure app sensor, https://nepp.nasa.gov/whisker/reference/tech_papers/2011-NASA-GSFC-whisker-failure-app-sensor.pdf (accessed Dec. 3, 2024).
- [9] DISA, Stig Viewer 3.x user guide, https://dl.dod.cyber.mil/wp-content/uploads/stigs/pdf/U_STIG_Viewer_3-x_User_Guide_V1R4.pdf (accessed Dec. 3, 2024).
- [10] “National Vulnerability Database,” NVD, <https://nvd.nist.gov/vuln> (accessed Dec. 2, 2024).
- [11] DoD, “Application Security STIG,” DoD Cyber Exchange, <https://public.cyber.mil/?s=Application%2BSecurity> (accessed Dec. 2, 2024).
- [12] N. Jadhav, “Tin Whisker,” YouTube, <https://www.youtube.com/watch?v=MJjIclFh7hY> (accessed Dec. 2, 2024).
- [13] Dgraal, “Unity licensing 101: From personal to pro,” GAME DEVELOPERS HUB, <https://www.game-developers.org/unity-licensing-101-from-personal-to-pro> (accessed Dec. 2, 2024).

[14] “Tin Whisker.” *YouTube*, YouTube, www.youtube.com/watch?v=MJjIclFh7hY. Accessed 2 Dec. 2024.

[15] “Unity editor software terms,” Unity, <https://unity.com/legal/editor-terms-of-service/software> (accessed Dec. 2, 2024).

[16] U. Technologies, “GameObject,” Unity Documentation, <https://docs.unity3d.com/Manual/class-GameObject.html> (accessed Dec. 2, 2024).

[20] “Creating and Using Materials,” Unity Documentation, <https://docs.unity3d.com/2019.3/Documentation/Manual/Materials.html> (accessed Apr. 15, 2024)

[21] P. Salmony, “Altium Designer Quick-Start Tutorial with Phil Salmony from Phil’s Lab,” YouTube, https://www.youtube.com/watch?v=YTGzncKU5RY&t=1228s&ab_channel=AltiumAcademy (accessed Dec. 2, 2024).