Rochester Institute of Technology
RIT Digital Institutional Repository

Theses

11-13-2023

BioDOV: A Tool Used to Encourage Discovery in Bioinformatics Through Gamification

Gregory D. Johnson Jr.
gdj5109@g.rit.edu

Follow this and additional works at: https://repository.rit.edu/theses

Recommended Citation
Johnson, Gregory D. Jr., "BioDOV: A Tool Used to Encourage Discovery in Bioinformatics Through Gamification" (2023). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

BioDOV: A Tool Used to Encourage Discovery in Bioinformatics Through Gamification

Gregory D. Johnson Jr

Rochester Institute of Technology

Committee Members

Dr. Gary Skuse

Dr. Gordon Broderick

Dr. Dina Newman

A Thesis submitted in Partial Fulfillment of the Requirements of the

Degree of Master of Science in Bioinformatics

Thomas H. Gosnell School of Life Science

College of Science

Rochester Institute of Technology

Rochester, NY

November 13, 2023

Abstract

Expressing big data in a meaningful manner is exceptionally difficult. Scientists often struggle with the visualization of data due to several limitations. Currently, many scientists use two-dimensional figures to display and analyze meaningful data. However, I believe using game design theory and game technologies, we can create more robust visualizations. These visualizations will allow scientists to easily communicate science to the masses and make better discoveries. Therefore, we introduce BioDOV (Biological Data Oriented Visualization). This tool will utilize game design principles to expand upon current visualization methods in the scientific community. Through this proof of concept, we will create the groundwork for better visualization technologies.

Table of Contents

BioDOV: A Tool Used to Encourage Discovery in Bioinformatics Through Gamification

Bioinformatics is an important field of discovery that will enable scientists and clinicians to better understand biological processes and their applications in medicine. Clinicians require bioinformaticians to assemble and present complex data for deep understanding. You can only interpret so much of the collected data. 2-D graphs and figures only allow clinicians to see from a fixed view. So, the question becomes, is there a way bioinformaticians can present data that allows for deeper dimensionality? This is where game design theory can be of use. We can use game design theory to change how individuals conceptualize and visualize information. We can take advantage of all the research conducted through video games and structure it to work with biological data.

Limitations of Current Approaches

Researchers have used games to display complex research educationally many times before. It is almost impossible to simplify the structure and dynamics of macromolecules, so it is essential to have many other scientific disciplines to assist (Kadir et al., 2021). Using games is a better way to illustrate the data from other scientific disciplines. This will allow scientists to make more profound deductions, and laymen to better understand science. Furthermore, many industries are starting to understand the potential of game engines.

The University of British Columbia used this methodology to portray its scientific findings to non-scientists. Using the Blender Game Engine, they have produced a simulation called OcenaViz (Steenbeek et al., 2021). Through their endeavors, they successfully constructed a brilliant example of marine-life ecosystems (Figure 1). They could communicate science to the masses by displaying this simulation. With the power game engines have, it’s no wonder scientists and other industry specialists are beginning to use them. Big car companies such as

Ford are beginning to use unreal engines due to the power and beauty of the graphics. It has even been used to craft exciting movies. So, what is stopping scientists from wielding the power of games? To answer that question, let us look at some pitfalls of 2D simulations of biological data.

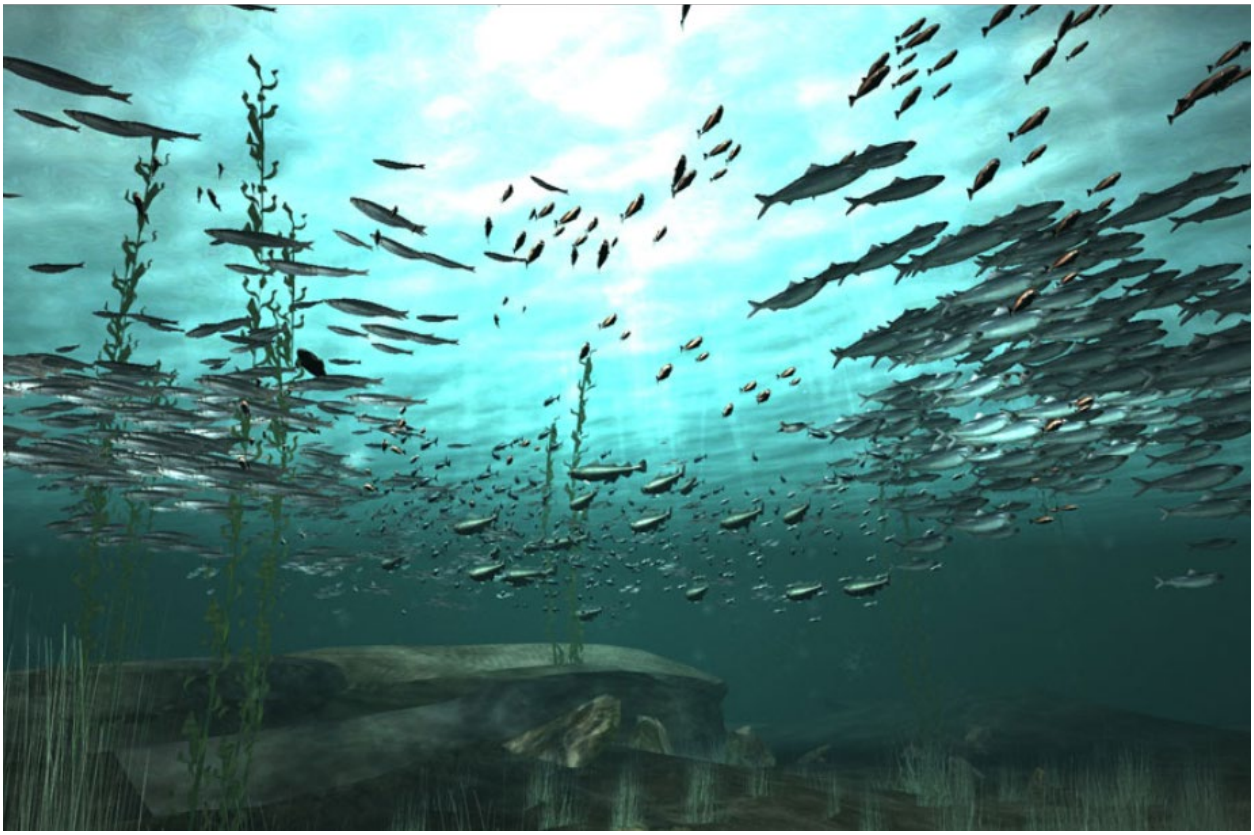

Figure 1: This is OceanViz. We are using this as an inspiration for the tool we plan to develop.

Cytoscape (https://cytoscape.org/), CellNetVis (https://github.com/heberleh/cellnetvis), and Coremine (https://www.coremine.com/) can provide crucial maps of related information. They can allow you to create maps of biological information to find references and relations of certain data. The web tool, CellNetVis can create a fascinating simulation that helps scientists research biological networks (Heberle et al., 2017) (Figure 2). Being able to increase our knowledge of cellular components dynamically will intrigue anyone. When you begin to look into the future, you must ask, is this future proof? A 2D simulation can only provide limited

information, especially considering the limits of lower dimensionality. When Dimensions and layers of information are added, you can visualize data that was never seen.

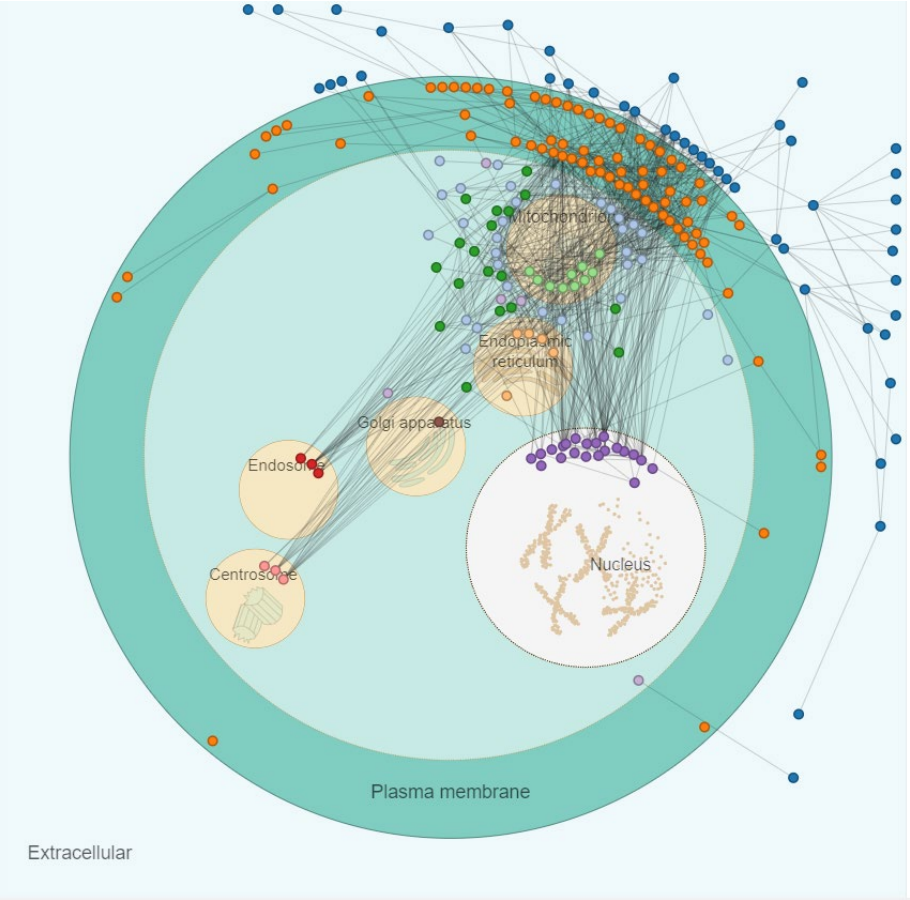

Figure 2: A visualization of CellNetVis, the web tool. It's a dynamic 2D simulation that shows how cells interact.

Organisms are three-dimensional and the creation of 3D visualization tools will aid researchers and scientists alike. If the Game community constantly pushes for better optimization strategies, better renderer graphics, and ways to teach end-users while playing their game, why can't scientists do the same (Lv et al., 2013)? There are so many multidisciplinary fields that are required when creating a game. From programmers to artists, they are all required to make the game great. Scientists will be able to answer many unanswered questions with the creation of a biological visualization such as this.

Even with all this information and all the questions we can answer, the most relevant concern is how we visualize the data. How do scientists and non-science individuals understand the data alike? Is it possible to create a system that will allow scientists to expand their knowledge in ways they never have before? As we explore this idea of visualization this question will be answered. Then we will be able to acknowledge the strengths and weaknesses of this approach. Through proper research of visualization methodologies and deeper understanding, we can truly make a difference.

In this paper, we will research by exploring molecular interactions revealed by cytokine storms. The cytokine storm interactions result from SARS-COV-2. We will accomplish this by using successful and previously tested tools and utilizing current visualization technology to study and analyze failures and make sustainable improvements.

Background

Literature often describes gamification as many tools used to engage participants. Users can experience innovative ideologies based on the use of the software and how developers use it. Gamifying school environments will keep students more engaged and willing to learn. When applied to business environments workers become more motivated and pleased to work. If real life causes boredom, anxiety, moral issues, and other pressing issues, why don't we just structure life closer to that of a game? Games are consistent and efficient sources of joyous experiences, so why don't we use them to make our lives more enjoyable (McGonigal, 2011)? McGonigal explains through literature that gamification can improve the enjoyment of life and allow us to work to our full potential. Therefore, intrinsic rewards can aid in teaching individuals by keeping them happy.

Zhihan Lv et al. explored how we can use video games to make better visualizations. Unity3D provides an easy-to-use interface to develop 3D applications using JavaScript, Boo, and CSharp code. Reflecting on how they used Unity to create a visualization and how powerful game engines are will advance the field. And, with all the features that were already available in Unity and the amazing community, it was only a matter of time before they could fully understand the potential of Unity. Massive success has been created through crowdsourcing big data and combining biology and game development (Lv et al., 2013). Looking at McGonigal ideals, she believes that games help generate enjoyment. With McGonigal's ideas, we can create a remarkable tool to showcase beautiful visualizations. Companies have created massive 3D simulations with McGonigal's ideals.

Oceanviz is a 3D simulation of underwater environments. They created Oceanviz to deal with the challenges created by fisheries (Steenbeek et al., 2021). This simulation allows users to directly see marine ecosystems in an environment that closely resembles ecosystems. Programmers and artists were required to create this project. What’s unique about this is the idea of a 3D simulation compared to that of 2D. 3D simulations show much more information and do not falter from information overload. Information overload occurs when audiences lose track of overarching questions due to the exposure of too many details (Steenbeek et al., 2021). An ecosystem will have many events occurring in it. This makes it harder for users to see this in 2D space or as static images. A great example of this is the simulation tool CellNetVis.

CellNetVis is an open-source software created in JavaScript and Hyper Text Markup Language (HTML). CellNetVis was developed as a plugin for Cytoscape but was eventually created to be a tool (Heberle et al., 2017). For a normal understanding of biological networks, or a quick understanding, this tool is perfect. However, when we get data that is unknown and has

no clear patterns, two-dimensional data becomes useless. A 2D Graph can show basic expression information of genes and proteins. However, it would be challenging to understand their interactions. The human body is complex and vast, how could a 2D static visualization or even dynamic simulation describe it? This is why being able to describe the pitfalls of this tool would be useful in conceptualizing the idea of dynamic 3D simulations.

Nanoscape is a dynamic simulation that simulates cell environments (Kadir et al., 2021). The only issue with tools like Nanoscape and Oceanviz is how they visualize information. Showing a 3D simulation of environments is hard to accomplish, but it is even harder to display information adequately. Looking at a simulation can provide rich data, but numerous people don't know anything about cells, genes, or even marine ecosystems. Before one creates the tool or simulation, one must ask themselves, who is your target audience? Most of these tools are created to display the power of gamification but do not target an audience. During this research, the question will be, how do we tailor this visualization towards scientists and non-scientists alike? Conversely, there was one piece of literature that used newer technology to visualize the information a bit differently.

VRNetzer is a Virtual Reality (VR) platform that displays a network representation of big data. They were able to create powerful visualizations of biological data with Unreal Engine and other tools (Pirch et al., 2021). This sounds like the other simulations or visualization but uses VR to display web-like images or graphs. VRNetzer is designed modularly, which allows for user customization and extensions for data analysis. It also allowed for visualization to be displayed separately (Pirch et al., 2021). Thus, although it still shows 2D representation in 3D space, it allows for fully dynamic data and visualizations. It furthermore gives users the power to customize this program as they see fit. This tool has a problem with data expressed in tabular and

old formats. VRNetzer can be prevalent because of how efficiently it can display data. However, VRNetzer fails to use Virtual Reality meaningfully.

What we can learn from these tools is how powerful but unpolished they are. If we could create a more dynamic feel to VRNetzer, and still allow for the mass amount of customization, we could visualize data in a way that has rarely been achieved. If we took lessons from Nanoscape and OceanViz to develop a tool that uses these simulations and allows scientists and non-scientists to understand, we would have an exemplary tool. Additionally, by adding more enjoyable features that allowed individuals to become more engaged, we could turn this from a good tool to a successful tool.

	CellNetVis	OceanViz	VRNetzer	Nanoscape	Proposed Tool
Goal	✗	✗	✓	✓	✓
Rules	✗	✗	✓	✓	✓
Feedback System	✗	✗	✗	✗	✓
Voluntary Participation	✓	✓	✓	✓	✓
Dynamic	✗	✓	✓	✓	✓

Figure 3: This chart shows each simulation and tool described in the literature.

Video Games

Video games are works created by large teams to entice millions of people. However, a game does not have to necessarily be in the form of a traditional video game. We can gamify

certain tasks using a technique known as gamification. Gamification is the psychological use of intrinsic rewards, created by happiness engineers to create an application that allows fun failures, increased motivation, and rewarding work through the self-creation of fiero and eustress (McGonigal, 2011). This quote directly describes the psychology of gamification. We want to create visualizations that keep people playing and motivated. This is an extremely hard task to implement properly. Most people can immediately tell the difference between a game that is fun and a game that is not fun. The difference between the two can be equivalent to a canyon, or as simple as walking. There are no right or wrong answers, and that is what makes it difficult.

This is why I have created a paradigm that follows three principles to create a visualization that appeals to scientists but also keeps the non-scientific crew engaged. The pillars are research, education, and technology. The research pillar focuses on the improvement of research. Using Gamification to speed up the way scientists research. The education pillar gives us the insight to create an educational tool. Making sure the tool itself can be used to teach individuals. The pillar of technology is concentrated on how we create the tool. Using the proper techniques to create a robust and manageable tool.

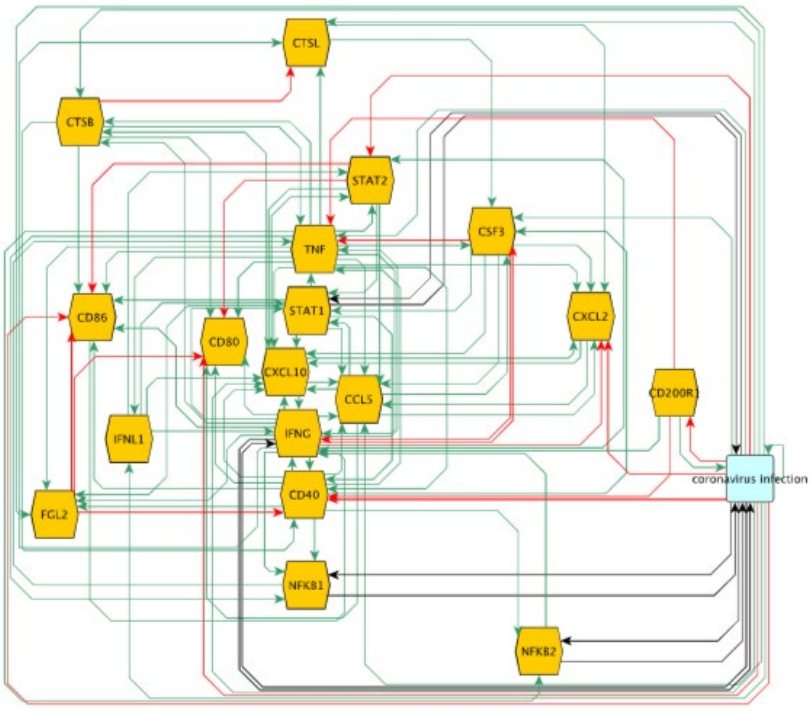

Figure 4: This is the simulation data currently being used in our prototype. This data represents the interactions between genes by showing how they are up and downregulated. This figure is a good representation of how difficult it is to get lost in data and shows the need for more 3D visualizations. Morris et al. paper explains more about the pathways of this graph.

Questions to Answer

The basis of this project has been stated many times. We want to create a system that allows for engaging research and stunning visualizations. However, some questions must be answered to understand why we believe our design is correct. One of the most important questions when designing research tools is explainability. How does our game explain and express gene/protein expression? Does it make sense when people find solutions? The core of our game is to allow individuals to create their paths and find unique solutions to the puzzle. Therefore, any dataset that is used in this project that users solve will give them ownership of the solution. They used tools available inside the game to solve the problem. This also allows them to document their steps for coming up with a solution. It's also important to understand

exactly what a game is. A game is a combination of four unique systems: a goal, rules, a feedback system, and voluntary participation (McGonigal, 2011) (Figure 3). Using these systems will allow us to answer the question, is this a game, or a "glorified game."

Arguably the most important question to answer is why. Why are we creating this game? Do we need to create better ways to visualize big data? The answer to this is relatively simple, we want to break down the wall between scientists and non-scientists, the wall between amateurs and professionals. After living through the COVID-19 pandemic, I am sure most people understand the tension between scientists and non-scientists. Scientists were branded as fakes, frauds, political, etc. It truly was the pandemic of misinformation, but if we were able to create something that non-scientists can use with scientists, we could break the wall that separates the two. This would give non-scientists the potential to replicate everything scientists do to understand science better.

The Hypothesis

Why do we think games would work so well for scientific communication? What exactly is gamification? Gamification is the psychological use of intrinsic rewards, created by happiness engineers to create an application that allows fun failures, increased motivation, and rewarding work through the self-creation of fiero and eustress (McGonigal, 2011). If gamification has been used time and time again to create joyful and meaningful experiences, let's take advantage of this. We plan to create a simulation that will allow users to transform their 2D data into a 3D visualization. This proof of concept will allow scientists in the field to make stronger guests about their data and allow non-scientists to understand it. Using several tools, we can create something efficient that has the potential to be extended. It may not work for every case but creating a proof of concept will allow scientists to understand the potential

strength of gamification. Using McGonigal's ideas as a base for what a game is, we can stay within the bounds of a game to truly demonstrate how powerful gamification is. We will start the creation of this proof of concept by turning Figure 4 into a 3D representation.

Gene Networks

A strong component of this project is the idea of gene networks. A gene network is a collection of genes and reflections of the relationships between them. They can be used to show relationships between proteins and RNA as well. Gene networks also regulate molecular genetics, biochemistry, and physical processes (Kolchanov et al., 2000). This gives us a deeper understanding of the downstream effects of genes. Look at Figure 4, this is a gene network related to cytokine storm. It shows the relationship between genes and how they affect each other. If one gene is upregulated, you can make a prediction (using the figure), to understand how other genes will be affected in the network. Biologically, you can understand more about the processes and functions of these genes. With an understanding of that, you could make inferences about why this gene being differentially regulated, affects another gene. It allows scientists to understand biology more deeply. And, using computational technologies, we can use analysis techniques to deeply analyze these networks to understand more (Kolchanov et al., 2000).

For this research, we wanted to expand upon the ideas of gene networks. Traditional, gene networks are displayed in 2D figures and shapes. They are novel and display expressive information but are limited by the way they are developed. So, by transforming it into a three-dimensional figure (or visualization), we can communicate and research the network more easily. Therefore, people would be able to see the downstream effects of other genes, and they could

make predictions about the functions, processes, and jobs of the genes as they affect others in the network.

Cytokine Storm Use Cases

Cytokine storm is when the immune system becomes overactive. Immune cells and cytokines are hyperactive, and your inflammatory response system flares out of control. Cytokine Storm tends to occur in many influenza-like diseases (Tisoncik et al., 2012). The dataset we are using is based on cytokine storm and acute respiratory distress (ARD). The cytokine storm dataset displays nineteen immune mediators and their involvement with cytokine storm (Morris et al., 2020). This allows us to see the relationship between the immune mediators and their involvement in cytokine storms based on the severity of COVID-19. Morris et al., created a 2D figure to display all the data, but by using more gamified methods have the potential to answer additional questions that were not necessarily answered.

In a 3D figure, we can see relationships in a much easier and dynamic manner. We can understand more about the information and explain it more easily. Morris et al. were able to make impressive predictions, but what if we had the potential to make predictions with the click of a button? What if we could program in statistical methods and other ideologies to understand the network more deeply? Since we propose to convert a 2D network into a 3D network, we already are a step ahead. We can see the downstream effects of immune mediators and understand more about why this happens before we test it inside the lab. Let us see how we were able to make a tool as powerful as this.

Materials and Methods

To develop software as complex as this we had to employ the use of multiple tools. We used the Unity Engine Game Engine for the development of the tool. Inside this engine, we used many statistical methods to display relationships and to create biological networks. We also had to create code in a neat and modular manner for extension in the future. Since the Unity Engine is a game engine, we were able to easily add gamification methods and create a simple proof of concept that can be used for future development. Before we talk about the use of unity, let's describe how the game was designed.

Game Elements

Players will explore the phenomenal world of genetics through gene expression. The idea is to give users the ability to play around with the expression level of certain genes. These genes will affect other genes depending on how highly or lowly expressed they are. Furthermore, we will accomplish this using an approach based on the flow of electricity. Using the high throughput of electricity depending on how strongly expressed certain genes are and extruding the areas will allow show users which genes are the most expressed.

The game will be composed into levels based on a threshold meter. The threshold meter will work like a gauge. The idea for each level is to require users to reach a minimum threshold to accomplish the level. Down the line, there is an opportunity to add higher points based on how full your gauge is. The idea is to use extrusion to display how expressed certain genes are. If you look at Figure 5 you can see extrusion on the motherboard image. We are going to take advantage of this idea to represent gene expression levels. While the flow of electricity (the lines), will represent what genes are connected. This gives us the flexibility to make complex levels for teaching, and the ability to create a tool that allows experts to make rational decisions.

Figure 5: This image provided inspiration for how the game will look. The motherboard shows certain integrated chips taller than others and shows pathways similar to a typical motherboard.

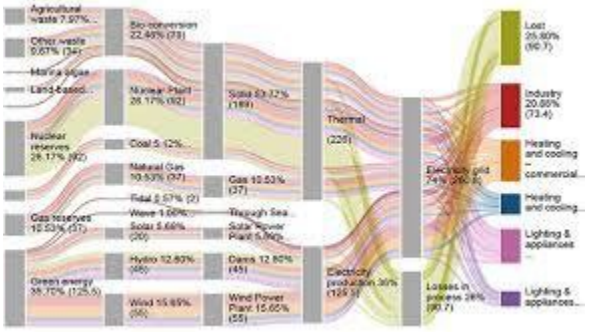

Figure 6: A 2D image of a flow chart. This will show players a more simplistic view of the current connections they are forming. This gives players a more simplistic chart to make decisions.

Gameplay

The objective of the game is to educate players and help researchers conduct research more engagingly. Gameplaywise, players will be mostly using point-and-click interaction to move around pieces on a motherboard and modify genes. Depending on which genes are pressed will determine the expression level of the pressed gene (if the gene state becomes zero, no expression, at gene state one, highly expressed). Other genes that are affected and connected to that gene will have changed expression levels as well. This can cause a domino effect if the user is not careful. To help the user see exactly what is being modified by the changes, we are introducing a map of expression. This map will showcase the levels of expressions and which

pathways are more heightened than others. The gameplay is relatively simple, but the execution is the most important part of this project. To be able to firmly teach individuals while keeping them engaged will be difficult, be manageable if done correctly.

Level Design

The levels will be manually created at the beginning of the project. We will develop them by using scriptable objects and modular objects. This will allow for simple 3D structures of our envisioned level. Going with the circuit board approach, we will create nodes at different locations based on the level a player is on. This will force players to think creatively about the problem at hand. We want to increase the difficulty level after each level to impose a challenge on the player. But we also want players to have ownership of their solution, as it could be used in future research.

Art and Audio

Due to the project's timeline, it would be impossible to develop or find audio for the project. We will be using stock art such as 3D cubes. We are building a proof of concept; therefore, we need to efficiently showcase it.

Story and Characters

There is no story based on the project. As it is an educational project. The story of the project is most closely related to how you solve the problem. How users can develop solutions to immune response drugs in a coronavirus-induced cytokine storm is the story.

Medical Objective vs. Game Objective

What are we going to accomplish with this project? From a medical point of view, we want to proliferate the speed at which we can analyze data. The quicker we can visualize the data, the sooner we can make inferences and diagnose problems based on the results. And, with a

project as experimental as this, the sky is the limit. But there are a few steps that must be taken. This is where the Game Objective comes into play. The reason why this is a game is to promote engagement. The development of this tool does not necessarily expand upon research or move immovable mountains, but it has the potential to. Therefore, if we use game theory properly, we can motivate players, engage players, and convey complex scientific data to individuals who do not understand it. So, by mixing these objectives, we can effectively create software that educates the masses and aids in the research of complex topics.

Unity Engine

The Unity Game Engine is powerful and has been used for the creation of a multitude of video games. In our case, we used Unity as a base for the creation of our tool. Unity has a visual interface and numerous preset objects already included (Figure 7). In Unity, an object that is seen inside of this visual interface is called a Game Object (GO). GOs are what makes Unity, Unity. We can manipulate and change these in the base interface. This includes scaling, rotating, and changing their position in a 3D space. Furthermore, we can also manipulate game objects using scripts.

Unity uses C# as a scripting language. This allows us to create C# code to manipulate GOs dynamically in Code. Moreover, every GO has a set of “Components” attached to them. Unity uses a component-based system to control how we interact with GOs. This includes the transform and material components. The Transform component allows us to change the rotation, scale, and position in a 3D space, and the material component allows us to change the material (such as the color, and resolution). What’s more, is that we can also create our components to manipulate the GOs and implement our logic in Unity. If we wanted to make a component that would move an object to the right every frame, we could create a script or “MonoBehaviour” to

add some vector to the GO position every frame. It is very simple and easy to understand after continued use. Therefore, using this as the base to create a proof of concept was an easy choice.

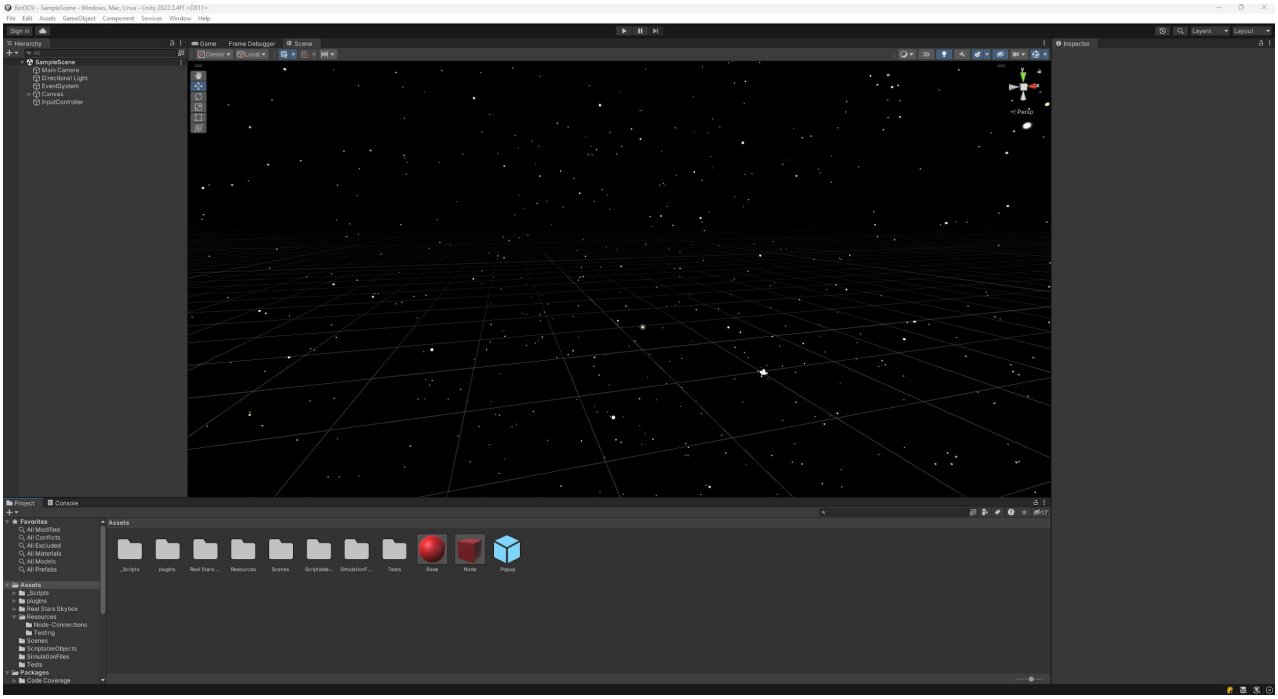

Figure 7: An image of how the Unity engine looks.

Code Architecture

The plan was to create software that could be extended and used for future research. To accomplish this, we had to create software that was clean, maintainable, easy to read, and easy to extend. The SOLID principles helped us immensely when developing this software. In the SOLID principles, S stands for Single Responsibility Principle, O stands for Open and Close Principle, L stands for Liskov Substitution Principle, I stands for Interface Segregation Principle, and D stands for Dependency Inversion Principle. In short, the Single Responsibility Principle tells us that each class should only have one responsibility (Ingeno, 2018). Essentially every time you create a class, it should only do one thing. Therefore, when you need to make changes, the only thing that it affects is itself. This helps prevent downstream issues inside of the code base.

The Open and Close Principle tells us that classes should be open for modification but closed for an extension. Once you create the final version of your method or class, you should not have to modify the class itself anymore because it is well-tested. However, the class should allow you the opportunity to extend new functionality to it. This reduces the number of changes you must make to existing code and allows you to quickly add new features. The Liskov Substitution Principle states that subtypes should be able to be substitutable from their base class. So, if class A is inheriting class B, class A should be a perfect substitute for class B without having any issues.

The Interface Segregation Principle states that we should have multiple interfaces instead of one big interface. Essentially ensure that they only have the functionality required. An interface is a contract with the compiler that states that the class or object will provide certain functionality. Therefore, ensuring that it only has the features it needs reduces the amount of error. Finally, the Dependency Inversion Principle tells us how to deal with tightly coupled classes. A tightly coupled class is a class that depends on others to work. It's a direct violation of the Single Responsibility Principle also because the class has more than one responsibility. Therefore, it tells us that classes should depend on abstraction or be giving the information they need upfront instead of directly referencing each other. And, because I was developing decent-sized software I had to follow a few rules to keep my code readable and maintainable. When it came to architecture, I used an event-based approach.

Event-Based Architecture

Event-based coding architectures are common, and even more common in the game community. This is heavily due to the need to create performant and efficient games. In Unity, if you want to detect key input, most will do it using the Update Function. The Update function is

called every frame, and you are allowed to check for changes in the world state through it. This function is a double-edged sword, especially for the newer programmers. It is very easy to use and allows for quick iterations on small projects, but as the project grows it becomes very oversaturated. To many using the Update function will cause a massive amount of overhead, and a lot of newer developers do not know how to deal with this. This is why I created an event architecture. The C# Language already allows for the creation of events or as they call them, delegates, and Actions. Delegates are like function pointers, where they are initialized by being referenced to a function (Figure 8). You can use them to call functions and pass them around like variables. Actions, the approach I used, is also created similarly. Like a function pointer but can be more flexible and easier to use.

```
namespace _Scripts.ScriptableObjects
{
    public class ActionEventClass
    {
        //Actions
        public System.Action action;

        //Delegates
        delegate void OnCalled();

        //This is how we invoke Actions
        public void CallAction()
        {
            action.Invoke();
        }
        public void OnAction()
        {
            System.Console.Write("Hello from actions");
        }

        //This is how we invoke delegates
        public void CallDelegate()
        {
            OnCalled onCalledDelegate = new OnCalled(OnEventCalled);
            onCalledDelegate.Invoke();
        }
        public void OnEventCalled()
        {
            System.Console.Write("Hello from delegates");
        }
    }
}
```

Figure 8: This figure represents the differences between actions and delegates. They can be used interchangeably, but they have different properties.

```
using System;
using System.Collections.Generic;
using _Scripts.Interface;
using UnityEngine;

namespace _Scripts.ScriptableObjects
{
    [CreateAssetMenu(fileName = "BaseEvents", menuName =
"ScriptableObjects/CreateBaseEvent", order = 1)]
    public class BaseEventScriptableObject : ScriptableObject
    {
        private List<IEventReactor> _eventsToRaise = new
List<IEventReactor>();
        //Objects that are called when the OnEventRaised Function is called
        public void Subscribe(IEventReactor eventReactor)
        {
            _eventsToRaise.Add(eventReactor);
        }
        //Remove objects to be called from the OnEventRaised Function
        public void UnSubscribe(IEventReactor eventReactor)
        {
            _eventsToRaise?.Remove(eventReactor);
        }
        //Raises the events, and calls the event function
        public void OnEventRaised(object objectToSend)
        {
            foreach (IEventReactor eventReactor in _eventsToRaise)
            {
                eventReactor.Execute(objectToSend);
            }
        }
    }
}
```

Figure 9: This figure shows the basic code used to create our event architecture. By combining a concept in unity called scriptable objects and a basic subscribe and unsubscribe system, we were able to create a simple event system. From this, you can also see that the BaseEventScriptablObject class inherits the ScriptableObject (SO), class.

For this project created a class that would control the flow of events. We used an event-based approach to declare and execute events. To accomplish this, we created a simple class called EventBase (Figure 9). However, to create a standalone object that did not have to be declared inside of the game at runtime, we created an object known as a ScriptableObject (SO). In Unity, a SO is a data container that is used to store information that does not have to be created on runtime. It is like a static database instance. So, using this with the EventBase class allowed us to create a standalone database instance that is not affected by anything inside of the

game. The EventBase Object has three functions: Subscribe, Unsubscribe, and OnEventRaised. It also stores a list of objects known as IEventReactors. The IEventReactor is an interface that contains the function Execute (Figure 10). This means any class that inherits the IEventReactor interface will have an execute function. So, when users wanted to subscribe to the EventBase class, the class would send a parameter of itself (since it inherits IEventReactor), and that would be added to the IEventReactor list. Consequently, if they wanted to be removed from the list, they would call Unsubscribe. From there, any class that has a reference to this SO would be able to call the OnEventRaised function. This would raise (or call) all the functions that are added to the IEventReactor list. What's more, is that because I can create many instances of this outside of runtime, I can have infinitely many BaseEvent types to control the event flow. So, if I wanted only one to control the flow from data insertion to data mapping, and another for another event, we could do that! In our case, you can see how we handled the flow of our events in the figures below (Figure 10).

```
namespace _Scripts.Interface
{
    public interface IEventReactor
    {
        public void Execute(object obj);
    }
}
```

Figure 10: This is the basic structure of the creation of the IEventReactor Interface. It forces classes to inherit an Execute function.

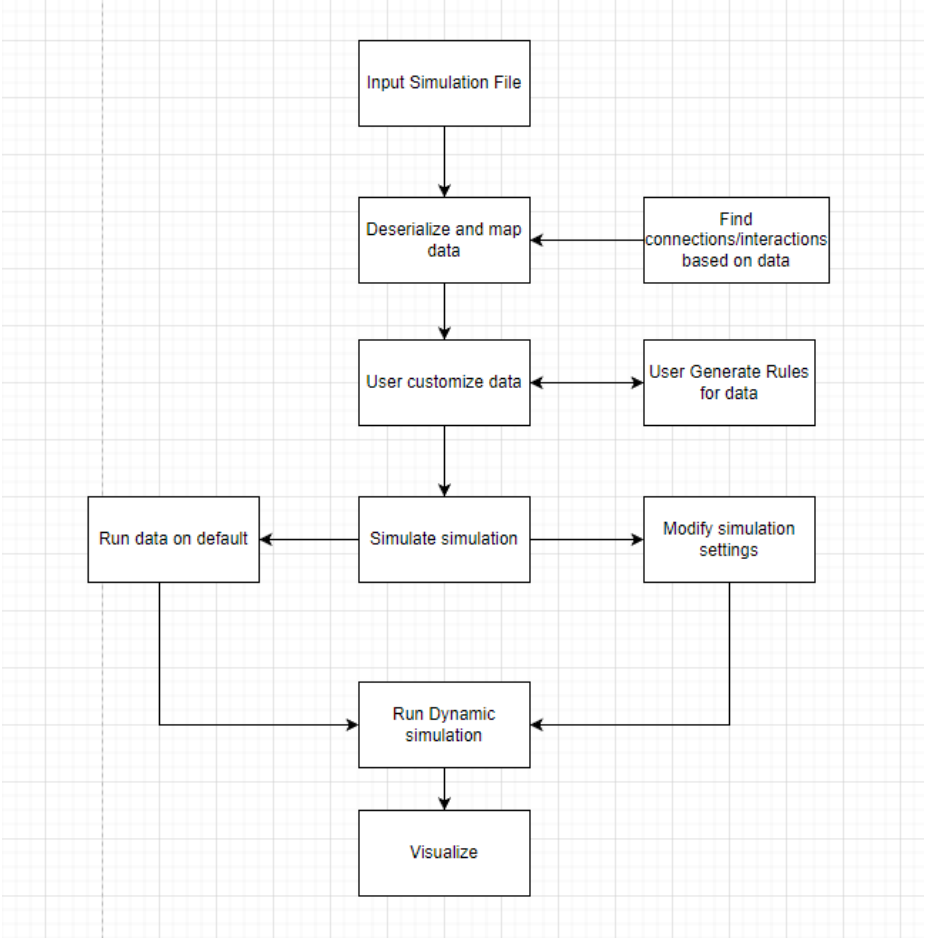

Figure 11: The basic flow of code. It displays in a flowchart how each task is done. This is how we handled our event execution.

Class Architecture

In cadence with the SOLID principles, we also have a myriad of classes that each affect the simulation. Each main class is as follows, DataInserter, LevelController, SimulationManager, and Simulator. Each one of these has a distinct purpose and controls one part of the code. Many supporting classes take upon smaller roles in the class as well to maintain code continuity. Let's delve deeper into the main classes.

The DataInserter class is responsible for the insertion of data. It consists of two functions, the Start Function, and the OnInsertEvent function. The Start function is a built-in Unity function that is always called upon object creation. MonoBehaviours are objects that can be added to GOs

and modify them. We used the Start function to call the OnInsertEvent function to start the event flow. Once the DataInserter class finishes reading the data, we use the SO event-based architecture to invoke an event. This event invokes the LevelController class.

The LevelController class is responsible for creating levels. In the case of this game, a level is displayed by the number of nodes on the screen. The more nodes that are displayed, the higher the level and the more complex it is to advance. The LevelController class has four functions, Awake, Init (Initialize), CreateLevel, and Execute. Like the Start function, the Awake function is called the moment an object is initialized, however, the Awake function is called before the Start function. The Init function is how we initialize the required data for the function to work properly. The CreateLevel function is used to create the level by determining how many nodes there are. Finally, the execute function is the event function. Because the LevelCreator class inherits the IEventReactor interface, it is required to have this function to react to events. As you can imagine, there is more that goes on in the background to ensure all the data for the level creation class is being read properly. Therefore, we have a few supporting classes to help, the LevelCreator class, and the LevelDataScriptableObject class.

The LevelCreator is a simple class that is only responsible for creating a class based on the data inserted from the DataInserter. It has a constructor to initialize the class, and a function named CreateLevel. The CreateLevel function creates several nodes based on the number of data it reads for the CSV class (Figure 12). This was created to ensure there is no violation of the SOLID principles. The LevelDataScriptableObject class is a class that references the SO class. This allows it to be an editable instance inside of the program.

```
namespace _Scripts.LevelCreation
{
    public class LevelCreator
    {
        private readonly Csv _csv;
        public LevelCreator(Csv csv)
        {
            _csv = csv;
        }
        public Csv CreateLevel(LevelConfig config)
        {
            Csv csv = new Csv();
            for (int i = 0; i < config.numberOfNodes; i++)
            {
                CsvNode node = _csv.Data[i];
                node.CurrentState = config.nodeStateState;
                csv.Data.Add(node);
            }
            return csv;
        }
    }
}
```

Figure 12: A representation of how we constructed the level creation class. It was created to be a standalone class that can be easily tested with test-driven development methodologies.

We can modify and change the values of the SO outside of the program to change the output of variables inside the engine. This maintains the separation of code and allows us to make changes without writing more code. In the case of LevelDataScritableObject, it is used to determine how we create the levels. It creates a public instance of the LevelConfig class. The LevelConfig class is a class I created to hold three variables. The number of nodes to create, the threshold to complete the level, and the states of the nodes (Figure 13). This class is declared inside the SO object as a public class to allow for individual instances of the class. Therefore, we can create many instances of this SO object and store them inside of a list. The list then is read like a queue, and once you fulfill the requirements to pass the level (threshold meter), you can continue to the next level.

```
namespace _Scripts.LevelCreation
{
    [System.Serializable]
    public class LevelConfig
    {
        public int numberOfNodes; // Nodes to be created
        public float thresholdToCompleteLevel; // Threshold of nodes that
need to be at a stable state to complete the level
        public int nodeStateState; // The start state of all the nodes
    }
}
namespace _Scripts.LevelCreation
{
    [CreateAssetMenu(fileName = "LevelData", menuName =
"ScriptableObjects/LevelData", order = 1)]
    public class LevelDataScriptableObject : ScriptableObject
    {
        [SerializeField]
        public LevelConfig levelConfig;
    }
}
```

Figure 13: This is the configuration of how we create levels. Just define these values and a different level will be created. You can also see the LevelDataScriptableObject class. This class can be directly created in unity as an object and can be easily modified and used.

The SimulationonManager class is used to manage the simulations. There are two kinds of simulation inside of the code base, the DefaultSimulation, and the CustomSimulation. The Default simulation is used to run simulations in default settings, while the custom allows for more modification. The DefaultSimulation classes inherit the ISimulator interface (Figure 14). It also contains eight functions and a constructor. The constructor requires a SimulationConfig object. This object keeps track of all changes to the data visualizations and allows us to make changes. The functions of the DefaultSimulation class are as follows: Simulate, Initialize, ExecuteCommand, ToDocumentation, UndoCommand, SetAsCurrentSimulator, Reset, and FinishSimulaton. Simulate is a public function that initializes the data. It requires a SimulationConfig to be passed along with it and it will call the Initialize function. The Initialize function is a private function that requires a list of Simulation Objects.

SimulationObjects is a class that I created to separate object interaction from other parts of the code. It is used to represent each node of the data. It then creates the objects and positions them properly. The ExecuteCommand function controls the flow of interactions during the simulation. Since users can interact and modify the nodes in the visualization, we needed to create a pattern to allow for a higher level of separation. Therefore, we created a command pattern (Figure 12).

```
namespace _Scripts.Interface
{
    public interface ISimulator
    {
        public SimulationConfig Config { get; }
        public bool Simulate(SimulationConfig config);
        public bool ExecuteCommand(List<ICommand> commands, SimulationObject
simulationObject);
        public bool UndoCommand();
        public bool Reset();
        public bool FinishSimulation();
    }
}
```

Figure 14: The basic ISimulator Interface. Each ISimulator is required to have each one of these functions. A function to simulate the data do and undo commands, reset its data, and finish the simulation.

The ToDocumentation function takes advantage of a saved list of commands. It turns the list into a string and writes it to a document (Figure 15). This allows users to understand more deeply what is going on in the black box (behind the scenes). The UndoCommand function allows us to undo the last command we processed from the ExecuteCommand function. The SetAsCurrentSimulator is an event function. It is used to notify the SimulationManager that we have created a simulator. The Reset function is used to reset the simulator (clear all the data). The FinishSimulation function is used when the simulation is finished and outputs the documentation data, like the ToDocumentation function. To Understand this deeply, we need to investigate the

use of the supporting classes MapSimulationObjects, the Command Programming Pattern, ICommands, and SimulationInvoker.

```
CD200R1 was interacted with, and stated changed from 0->1
CCL5 changed color to: RGBA(0.000, 1.000, 0.000, 1.000)
CD200R1 changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CD40 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CD80 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CD86 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CSF3 changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CTSB changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CTSL changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CXCL10 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CXCL2 changed color to: RGBA(1.000, 0.000, 0.000, 1.000)

CCL5 Prediction, OldState: 2, NewState: 2CD200R1 Prediction, OldState: 0,
 NewState: 0CD40 Prediction, OldState: 1, NewState: 0CD80 Prediction,
OldState: 0, NewState: 0CD86 Prediction, OldState: 1, NewState: 0CSF3 Prediction,
 OldState: 1, NewState: 1CTSB Prediction, OldState: 1, NewState: 1CTSL Prediction,
OldState: 2, NewState: 1CXCL2 Prediction, OldState: 2, NewState: 1
CCL5 changed color to: RGBA(0.000, 1.000, 0.000, 1.000)
CD200R1 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CD40 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CD80 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CD86 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CSF3 changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CTSB changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CTSL changed color to: RGBA(1.000, 0.000, 0.000, 1.000)
CXCL10 changed color to: RGBA(1.000, 1.000, 1.000, 1.000)
CXCL2 changed color to: RGBA(1.000, 0.000, 0.000, 1.000)

CCL5 was extruded from (0.30, 2.30, 0.30) -> (0.30, 2.05, 0.30)
CD200R1 was extruded from (0.30, 0.30, 0.30) -> (0.30, 0.36, 0.30)
CD40 was extruded from (0.30, 0.98, 0.30) -> (0.30, 0.30, 0.30)
CD80 was extruded from (0.30, 0.73, 0.30) -> (0.30, 0.58, 0.30)
CD86 was extruded from (0.30, 1.00, 0.30) -> (0.30, 0.35, 0.30)
CSF3 was extruded from (0.30, 1.30, 0.30) -> (0.30, 1.30, 0.30)
CTSB was extruded from (0.30, 1.34, 0.30) -> (0.30, 1.21, 0.30)
```

Figure 15: An example of the data AutoDocumentation generates. This can be used to trace back your steps.

MapSimulationObjects is a class that positions nodes in a three-dimensional space based on how closely related the nodes are. If they are related, we use space for them more closely together than if they are not. It also manages the creation of relationship lines. These lines are created using a Relationship Renderer class. The relationship renderer class only has one function called connect lines, and it takes two LineRenderers. It then connects those lines to depict a relationship. The MapSimulationObject class contains two functions:

MapBasedOnRelationship and CleanUp. The MapBasedOnRelationship function maps the objects together and the CleanUp function allows us to remove the depiction of relationships and clean up the created lines. The CleanUp function makes it easier for us to switch between levels.

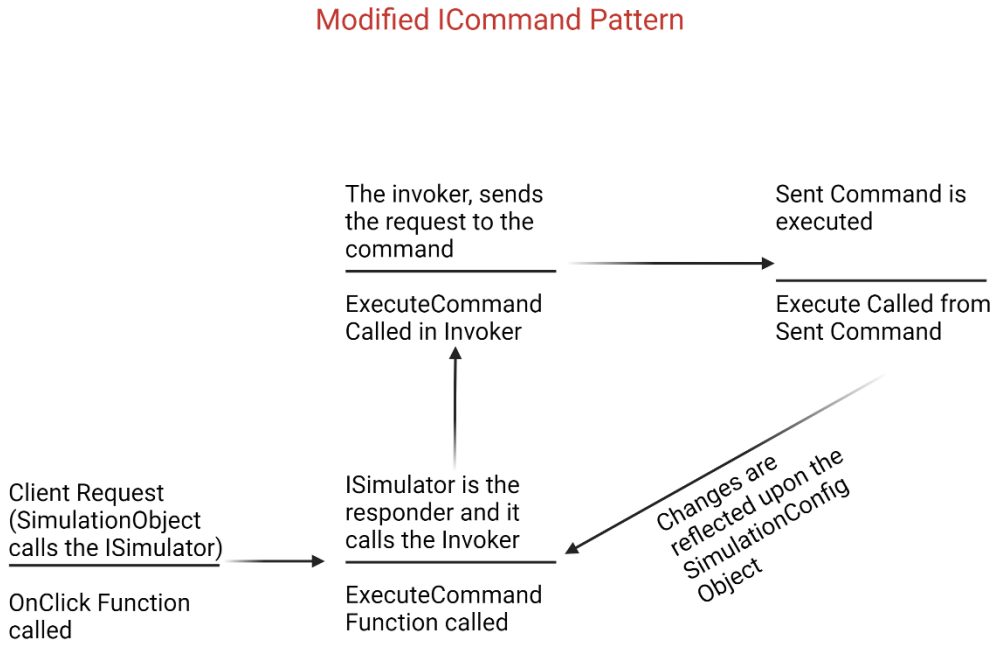

Figure 16: Our modified version of an ICommand pattern. It uses the pattern for the basic structure, but to make proper modifications for our purposes, we had to add additional information. We needed to pass a reference of the SimulationConfig so we could change the objects in the scene.

```
namespace _Scripts.Commands
{
    public class ClickDetectionCommand : ICommand // Concrete Commands
    {

        public SimulationData Data { get; private set; }
        private KeyValuePair<int, double> _prevState = new KeyValuePair<int,
double>();
        private string _docString;
        public bool Execute()
        {
            _docString = "";
            SimulationData currentData = Data;
            double[] states = Data.CurrentStates;
            List<SimulationObject> allObjects = Data.AllCurrentObjects;
            SimulationObject currentObject = Data.CurrentInteractedObject;
            int index = allObjects.IndexOf(currentObject);
            if (index == -1)
                return false;
            double oldVal = states[index];
            _prevState = new KeyValuePair<int, double>(index, oldVal);
            double newVal = (oldVal + 1) % 3;
            states[index] = newVal;
            currentObject.Node.CurrentState = newVal;
            currentData.CurrentStates = states;
            Data = currentData;
            _docString = $"{currentObject.Node.Name} was interacted with," +
                                $" and stated changed from {oldVal}->{newVal}";
            return true;
        }
        public bool Undo()
        {
            SimulationData oldData = Data;
            oldData.CurrentStates[_prevState.Key] = _prevState.Value;
            Data.AllCurrentObjects[_prevState.Key].Node.CurrentState =
_prevState.Value;
            Data = oldData;
            return true;
        }

        public void Set(SimulationData data)
        {
            Data = data;
        }
        public override string ToString()
        {
            return _docString;
        }
    }
}
```

Figure 17: An example of how we implemented and created command classes. We can create more commands using this structure.

Command Pattern

As we continued through the methods, there was another approach I took to maintain clean and scalable code. I used a programmatic pattern called the command pattern (Figure 16). The command pattern takes advantage of loosely coupled code to create command-like classes

(Figure 17). It is defined as a behavioral programming pattern that allows you to encapsulate many requests as objects and supports undoing requests (Gamma et al., 1994). For this project, we used this for all our interactions. When we wanted to interact with objects, it was a command, this goes for changing the color and size of objects as well. We made modifications to the structure of the command pattern, but it works just the same. We started by creating the ICommand interface.

```
    public interface ICommand
    {
        public SimulationData Data { get; }
        public bool Execute();
        public bool Undo();
        public void Set(SimulationData data);
    }
}
```

Figure 18: This was how we structured our command classes. They are created with the ability to execute and undo commands. We had to add the set function, so we knew exactly what data we were working with to modify.

The ICommand interface is defined with a public variable of SimulationData and three functions: Execute, Undo, and Set (Figure 18). The Execute function is how we execute the commands, the Undo function allows us to redo the last command we called, and the Set function is how we acquire the data necessary to complete the commands. Most command patterns only require an Execute and Undo function, but in our cases, we need to modify the structure. Following this we needed to create an invoker to control the flow of commands. Therefore, we created the SimulationInvoker class (Figure 19).

```
public class SimulationInvoker
{
    private Stack<List<ICommand>> _commands = new Stack<List<ICommand>>();
    public SimulationData ExecuteCommand(List<ICommand> commands, ref
SimulationData data)
    {
        foreach (var command in commands)
        {
            command.Set(data);
            command.Execute();
            data = command.Data;
        }
        _commands.Push(commands);
        return data;

    }

    public bool RemoveRecentCommand()
    {
        if (_commands.Count <= 0)
            return false;
        _commands.Pop();
        return true;
    }
    public bool UndoCommands(ref SimulationData data)
    {
        if (_commands.Count <= 0)
            return false;
        List<ICommand> commands = _commands.Pop();
        foreach (var command in commands)
        {
            command.Set(data);
            command.Undo();
            data = command.Data;
        }
        return true;
    }
    public SimulationData UndoAllCommands(ref SimulationData data)
    {
        int count = _commands.Count;
        for (int i = 0; i < count; i++)
        {
            List<ICommand> commands = _commands.Pop();
            foreach (var command in commands)
            {
                command.Set(data);
                command.Undo();
                data = command.Data;
            }
        }

        return data;
    }
}
```

Figure 19: The SimulationInvoker class is responsible for the invocation of our command request. It processes them and makes the calls. It can also undo commands at will.

The SimulationInvoker class is responsible for the execution of command requests. It contains 4 functions: ExecuteCommand, RemoveReccentCommand, UndoCommand, UndoAllCommands. The ExecuteCommand function takes a list of commands and executes them all together. The RemoveReventCommand removes the latest command from the commands stack. The commands stack is used to hold a reference of all the events that occur. The UndoCommand function undoes the last command requests. It does this by removing the command from the stack and calling the undo command function on the objects. Lastly, the UndoAllCommands function removes every command that has been called. There are also four commands that we are using in this project: ClickDetectionCommand, ExtrudeObjectCommand, PredictGeneStateCommand, and ChangeColorBasedOnStateCommand.

Every command includes the three functions and variable that was defined in the ICommand interface (Execute, Undo, Set). This is because each of them is defined using the ICommand interface. The ClickDetectionCommand gives us information and changes the state of the node that was recently pressed. The ChangeColorBasedOnStateCommand changes the color of each node based on its current state. The PredictGeneStateCommand uses statistical methods to predict the new states of genes based on current changes. Finally, the ExtrudeObjectCommand extrudes nodes based on how expressed they are. Using this invoker and ICommand interface, and interjoining it with my Simulator classes, I can create a command system that is easy to use and modify. If someone wanted to add new commands, they would need to create a new class (such as PrintHelloCommand) and allow it to derive from the ICommand interface. After filling in the functions with information relevant to how they want the command to function, they would then pass the command in a list to the invoker. This would then execute the command. Therefore, this code is easily scalable.

Other Supporting Classes

There are a few other classes with continued responsibilities. They are present in other parts of the code to maintain code integrity. They are as follows: SimulationObjects, SimulationObjectResponder, CameraController, CameraMovement, and CameraRotation. The SimulationObject is a representation of the node. It contains four functions: Init, OnPointerClick, OnPointerEnter, and OnPointerExit. The Init (Initialize) function initializes the function using the ISimulator parameter passed to it. The OnPointerClick function is used to check for mouse press interactions. The OnPointerEnter function checks for mouse hover events while the OnPointerExit event checks to detect when the mouse is not hovering over the object. The OnPointerClick, OnPointerEnter, and OnPointerExit functions are derived from the interfaces IPointerClickHandler, IPointerEnterHandler, and IPointerExitHandler. To further separate logic from the SimulatonObject class we had to create two other classes, GeneDataDisplay and SimulationObjectResponder.

The GeneDataDisplay contains 2 Functions and a constructor. The Constructor wants you to pass a datastructure of GeneDisplayInformation which contains information about the gene. The other two functions are the OnPointerEnter and OnPointerExit. These are used to represent the pointer interaction for SimulationObjects. The SimulationObjectResponder has a constructor and an OnClick function. The SimulationObjectResponder constructor requires an ISimulator and the simulation object to initialize. The OnClick function notifies the simulation manager about interaction (on click presses). These two classes are connected to the SimulationObject class so we can maintain our core code principles (SOLID).

The CameraController, CameraMovement, and CameraRoation classes are all responsible for the movement and rotation of the camera. The CameraMovement class controls the

movement while the CameraRotation controls the rotation. Both objects are controlled by the CameraController class which initializes them.

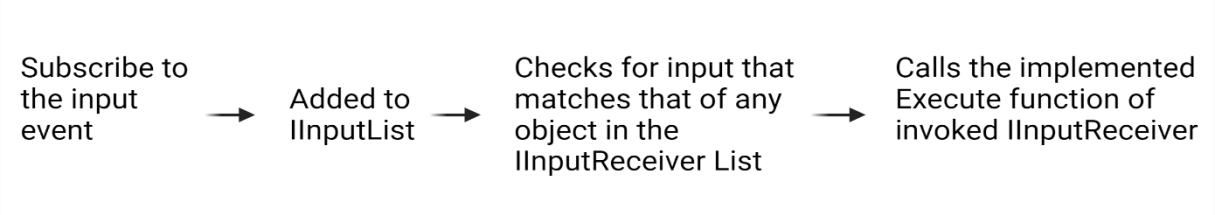

Figure 20: Shows the basic flow for how we handle user input. Allow users to subscribe to the action, and then we invoke the object once the condition is met.

Input Events

We needed to construct another event system for inputs that worked with our current event architecture. We did not want to use commands for this event system due to the complexity, and the amount of data that would be saved every time we pressed the button. Therefore, the creation of an input event class was warranted (Figure 20).

```
namespace _Scripts.Event
{
    public class InputEventListener : MonoBehaviour, IEventReactor
    {
        public BaseEventScriptableObject inputEventScriptableObject;
        private readonly List<IInputReceivers> _receiversList = new
List<IInputReceivers>();

        private void Awake()
        {
            inputEventScriptableObject.Subscribe(this);
        }
        private void Update()
        {
            foreach (KeyCode keyCode in
System.Enum.GetValues(typeof(KeyCode)))
            {
                if (Input.GetKey(keyCode))
                {
                    CheckForInputEvent(keyCode);
                }
            }
        }
        private void CheckForInputEvent(KeyCode key)
        {
            foreach (var receiver in _receiversList)
            {
                if (receiver.Keys.Contains(key))
                {
                    receiver.ExecuteKey(key);
                }
            }
        }
        public void Execute(object obj)
        {
            var inputReceivers = (IInputReceivers)obj;
            _receiversList.Add(inputReceivers);
        }
    }
}
```

Figure 21: This is the class we use to create the input system. Objects can subscribe to it through the IEventReactor Structure, and the objects that are subscribed to the object are IInputReceivers. We then check for certain inputs and invoke those input events.

The class we created was called InputEventListener (Figure 21). It inherited the IEventReactor interface and contained four functions: Awake, Update, CheckForInputEvent, and Execute. The awake command was used to subscribe to our BaseEventScriptableObject. The CheckForInputEvent function takes advantage of the newly created interface IInputReceiver.

IInputReceivers is an interface that allows objects to contain input events (Figure 22). Therefore, creating a list of IInputReceivers and forcing the objects to subscribe to it from the IEventReactor worked flawlessly. This allowed the CheckForInputEvent Function to only need to receive the key that was clicked last frame and check if any of the IInputReceiver object input keys are equal to it. The Update function would check every frame and call the CheckForInputEvent function to ensure the flow. The creation of this not only allows us to move the camera based on input interactions but if we wanted to add more events based on key inputs, we could easily create a new class with the IInputReceiver and have it notify the InputEventListener class to be added to the list.

```
namespace _Scripts.Interface
{
    public interface IInputReceivers
    {
        KeyCode[] Keys { get; }
        void ExecuteKey(KeyCode code);
    }
}
```

Figure 22: The interface used to define IInputReceivers. Using this we can confirm that the IInputReceivers have the functionality needed to work within the system.

Network Extraction and State Transition Modeling

We had to employ several statistical methods to properly visualize a biological network. For the prediction of gene states, we used a Partial Least Squares (PLS) approach. PLS is a method based on principle component analysis (PCA) and is used to reduce the number of variables you need to make predictions (Esposito Vinzi & Russolillo, 2013). This was implemented by creating a class named PartialLeastSquresPredictionModel (Figure 23). This class contains two functions and a constructor. The two functions are named Predict and Initialize and the constructor takes in the network data. The Initialize function maps the data to

allow for predictions, and the Predict function makes predictions based on the data given. For our Partial Least Squares method, we are using the Accord.net package.

```
public class PartialLeastSquaresPredictionModel
{
    private CsvNode _nodeToPredict;
    private PartialLeastSquaresAnalysis _partialLeastSquaresAnalysis;
    private MultivariateLinearRegression _linearRegressionModel;
    private int _min = 0;
    private int _max = 2;
    public double UnRoundedPredictionValue = 0;

    public PartialLeastSquaresPredictionModel(CsvNode node, Csv csv)
    {
        _nodeToPredict = node;
        Initialize(csv, node.Name);
    }

    public double Predict(double[] newStateChanges)
    {
        // Samples have to be == to features used (columns used)
        // Normalize the feature vector and make predictions
        double[][] newSampleInput = new double[][] {newStateChanges};
        double[][] predictions =
                _linearRegressionModel.Transform(newSampleInput); // The
transformation vector, needs a feature vector (**see above)
        double val = Math.Clamp(predictions[0][0], _min, _max);
        UnRoundedPredictionValue = double.IsNaN(val) ? 0d : val;
        val = Math.Round(val);
        if (double.IsNaN(val))
            val = _nodeToPredict.CurrentState; // if it is NAN that means the
data(in this case) has no baring
        return val;                             // Or effect on the current
data (keep it the same)
    }
```

Figure 23: This figure shows how we make predictions using the partial least square method. We are using accord.net to make predictions.

The MapSimulationObject class we discussed described mapping objects together based on relationship. To accomplish this, we had to use a mixture of statistical tests with weights. In our case, we took the correlation and covariance of the input network data. This allowed us to space out objects in a three-dimensional space based on how closely related they are. This was accomplished by creating a RelationshipStatisticalAnalysisModel Class. This class has three functions, CovarianceTest, CalculationCorrelation, and AnalysisRelationship (Figure 24). The

CovarianceTest function calculates the covariance while the CalculateCorrelation function calculates the correlation between the two nodes. The AnalysisRelationship combines both of those values using a weighted approach. We decided to take twenty-five percent of the covariance, and seventy-five percent of the correlation, and then add them up. This would give us an averaged-out relationship between the two variables.

```
namespace _Scripts.Statistics
{
    public class RelationshipStatisticalAnalysisModel
    {
        private double CalculateCorrelation(IEnumerable<double> dataOne,
IEnumerable<double> dataTwo)
        {
            return Correlation.Pearson(dataOne, dataTwo);
        }

        private double CovarianceTest(IEnumerable<double> dataOne,
IEnumerable<double> dataTwo)
        {
            return dataOne.Covariance(dataTwo);
        }

        private double CrossCorrelation(List<double> dataOne, List<double>
dataTwo)
        {
            return
CrossCorrelationAnalysisModel.CalculateCrossCorrelation(dataOne.ToArray(),
dataTwo.ToArray());
        }

        public double AnalysisRelationship(List<double> initialData,
List<double> otherData)
        {
            //Lets apply weights
            List<double> variances = new List<double>
            {
                CovarianceTest(initialData, otherData),
                CalculateCorrelation(initialData, otherData)
            };
            //Weighted measure
            const double weightOne = 0.25d;
            const double weightTwo = 0.75d;

            double weightedRes = (weightOne * variances[0]) + (weightTwo *
variances[1]);
            //Debug.Log($"{variances[0]}, and {variances[1]}:
{weightedRes}");
            double result = weightedRes;
            return double.IsNaN(result) ? 0 : result;
        }

    }
}
```

Figure 24: This is the function we are using to conduct covariance and correlation. Also uses accord.net.

Inside of the MapSimulationObject, we also had to calculate the relationship between the two nodes. Therefore, we recruited the use of Granger causality testing. Granger causality testing (GCT) determines a "Granger" relationship between variables depending on how well one time series predicts another. Using this together with a t-test, chi-squared test or f-test will give us the data needed to make the predictions. Heerah et al. also used GCT to identify causal relationships, however, we had to take a different approach. To incorporate this inside of unity we created a GrangerCausalityTestingModel class (Figure 25). This class has two functions, IsGrangerCausal and FitAutoRegressiveModel. The IsGrangerCasual Function is created with a default maxLag variable, which is set to ten. This then uses the lagged variables to create the forecaster models using an Ordinary Least Squares (OLS) model. We used Accord.net to create the OLS model. We then finish by using a chi-squared test to compare the two models. This gives us the information necessary to perform the analysis and draw conclusions. This was the code required to accomplish our task but mathematically it looks a little different.

Network Extraction

$$Model\ A = \gamma(t)_a = \alpha_a + \beta_a * \gamma(t-1)_a \quad (1)$$

$$Model\ B = \gamma(t)_b = \alpha_b + \beta_b * \gamma(t-1)_b \quad (2)$$

To extract the data needed to create the network, we had to use a combination of GCT, correlation, and covariance. GCT provides the relationships, and correlation and covariance spaces nodes based on their relationship. We had to create two different equations to calculate the GCT for the data. Look at equations 1 and 2, these are the equations necessary to create the standard regression models. Alpha(a) is represented by the intercept of the model, Beta(a) is represented by the weight and Gamma (t – 1) is the lagged values. This is also depicted in

Figure 25. Models A and B represent the created predicted values based on the original data and the lagged data.

$$Relationship\ Significance\ of\ Model\ A = \sum \frac{(\gamma(t)_{Bi} - l(s)_{ai})^2}{l(s)_{ai}} \quad (3)$$

$$Relationship\ Significance\ of\ Model\ B = \sum \frac{(\gamma(t)_{ai} - l(s)_{bi})^2}{l(s)_{bi}} \quad (4)$$

Once we created our models and had a list of predicted values, we used a chi-squared test of goodness to test the significance (3 and 4). In equations 3 and 4, the observed value is Gamma(t) (The predicted values), and l(s) is the lagged series of values (or the expected values). This allowed us to determine the bi-directional relationship between nodes. This would allow users to predict the downstream effect of the PLS model before making the prediction. And although correlation and covariance are very similar tests, we used them together with these models, to determine the positions of the nodes in the network. The reason we used them together was to overcome any additional errors or incorrect results that come about from using only one statistic. There were times when I believed we needed to use another test of correlation or a meta-analysis to predict it more accurately. However, for the case of this proof of concept, this was more than sufficient to give us an accurate representation of how well this tool can work.

State Transition Model

For our state transition model, we used PLS. I trained a PLS model using the seventeen features (columns) from our dataset as input. I then created an output list that contains all the states of the node (feature) that we are trying to predict. To make the prediction, I input a new list with changed values inside of the model. The linear model then computes it and makes a prediction based on the changes in the features. To represent the changes downstream, we apply

the change in states to every node. This works because every node has a different model based on their predictors and input data. So, we apply this method of changes to the list of data and predict to determine what state each node will transition to.

```
public static GrangerRelationship IsGrangerCausal(double[] seriesA, double[] seriesB,
int maxLag = 10)
{
    int aBSignificantCounter = 0; //If A and B are significant, Bi-direction
relationship
    int aSignificantCounter = 0; // If just A is significant, A Granger B
    // Iterate over lags
    for (int lag = 1; lag <= maxLag; lag++)
    {
        // Extract lagged values
        double[] laggedA = new double[seriesA.Length - lag];
        double[] laggedB = new double[seriesB.Length - lag];
        Array.Copy(seriesA, lag, laggedA, 0, laggedA.Length);
        Array.Copy(seriesB, lag, laggedB, 0, laggedB.Length);

        double[] outputA = seriesA[lag..];
        double[] outputB = seriesB[lag..];
        // Fit autoregressive models
        double alphaA, betaA;
        int degreesOfFreedomA, degreesOfFreedomB;
        FitAutoregressiveModel(laggedA, outputA, out alphaA, out betaA, out
degreesOfFreedomA);

        double alphaB, betaB;
        FitAutoregressiveModel(laggedB, outputB, out alphaB, out betaB, out
degreesOfFreedomB);
        // Perform comparison (you might use statistical tests here)
        //To calculate residuals (for predictions)
        // Y(t) = alpha + Beta(1) * Y(t-1)
        double[] predictedListA = new double[outputA.Length];
        for (int i = 1; i < outputA.Length; i++)
        {
            predictedListA[i] = alphaA + betaA * outputA[i - 1]; //output A is already
lagged at time
        }

        double[] predictedListB = new double[outputB.Length];
        for (int i = 1; i < outputB.Length; i++)
        {
            predictedListB[i] = alphaB + betaB * outputB[i - 1]; //output A is already
lagged at time
        }

        ChiSquareTest chiSquareModelA = new ChiSquareTest(outputA, predictedListB,
degreesOfFreedomA);
        ChiSquareTest chiSquareModelB = new ChiSquareTest(outputB, predictedListA,
degreesOfFreedomB);
        switch (chiSquareModelA.Significant)
        {
            case true when chiSquareModelB.Significant:
                aBSignificantCounter++;
                break;
            case true:
                aSignificantCounter++;
                break;
        }
    }
    Debug.Log($"Evidence: AB significance: {aBSignificantCounter}/{maxLag}, " +
              $"A Significance: {aSignificantCounter}/{maxLag}");
    // No evidence of Granger causality
    return aBSignificantCounter > aSignificantCounter?
GrangerRelationship.Bidirectional : GrangerRelationship.Directional;
}
```

Figure 25: This is the function we used to define Granger relationships. As described, we used to take the average of two lagged models and compared the models using Chi-Squared tests.

Monobehaviors, Classes, and Test-Driven Development (TDD)

One of the biggest reasons for the large number of classes was due to the need for test-driven development (TDD). Test Driven developed is used to test the functionality of classes before you run your software. This allows you to quickly find issues with your code and fix them. It also helps to eliminate the fear of issues that may arise (Beck, 2022). However, there is a problem with TDD in the unity engine. Inherently all unity classes that are seen inside the game inherit the MonoBehvaiour class. This class controls the event functionality, creation, and manipulation of game objects. Without it, we are unable to directly control how GameObjects act. Furthermore, a game object must inherit this class to be added to objects inside the scene. The issue for TDD comes from this behavior. To test objects, you must recreate the actual in-game behavior. Normally this would be impossible to do, but since we have separated all our game logic into classes, we can easily test the output of the data (Figure 26).

```
namespace Tests
{
    [TestFixture]
    public class CsvDataTest
    {
        public DataInserter Inserter;
        [Test]
        [TestCase("SimulationFiles/DataExtra.csv")]
        public void TestInserter(string path)
        {
            path = Path.Combine(Application.dataPath, path);
            FileInserter fileInserter = new FileInserter(path);
            Csv csv = fileInserter.ReadData();
            Assert.True(csv.Data.Count > 0);
        }

        [SetUp]
        public void Setup()
        {
            GameObject gameObject = new GameObject();
            Inserter = gameObject.AddComponent<DataInserter>();
        }

        [Test]
        [TestCase("SimulationFiles/DataExtra.csv")]
        public void TestDataInserter(string path)
        {
            path = Path.Combine(Application.dataPath, path);
            bool isTrue = Inserter.OnInsertEvent(path);
            Assert.True(isTrue);
        }
    }
}
```

Figure 26: This is how we constructed our test for test-driven development. Unity has a built-in unit test package that we used. The [Setup] tag is used to set up the objects we are going to test. The functions that have the [Test] tag are the functions that run the test for the given functionality. Finally, the [TestCase] tag is used when you want to test multiple tests. It is passed as a path in the TestDataInserter function.

Cytokine Data

The dataset we are using is based on predicting immune response to drugs in a coronavirus-induced cytokine storm (Morris et al., 2020). We are going to transform the 2D visualization that was created from this paper (Figure 4) and add another dimension to it. We are also going to change the statistical analysis methods on data. This is to ensure we have enough data to represent the model in 3D.

The data inside of the dataset is a collection of discrete values: zero, one, two. Each value represents a state. Zero represents a “No expression state”, one represents “Low expression”, and two represents “High expression.” With the use of the statistical package Accord.net, we were able to create regressions and make predictions. The Accord.net packages required us to store each state in the column in a list. Following this, we created a target value list for a row of values as our “target predictors.” Using our target predictors with the list of data we acquired from the dataset we created a PartialLeastSquaresAnalysis model in Accord.net (the process was similar for all other types of models in the package). Considering the dataset refers to nineteen immune mediator genes, there were initial assumptions I had made about the data. Such as expecting the genes to be strongly affected by their related genes. However, that was not always the case. Many other gene-to-gene interactions are much easier to see in a 3D visualization than a 2D one.

How Does It All Work Together

To accomplish the creation of our system, there were a myriad of systems to be put in place. We had to construct methods for the simulation to work properly. There was a need to insert and analyze data properly. We needed to ensure that all the nodes were respected properly using the data. To develop relationships and make predictions in data we needed to create statistical methodologies and classes. Furthermore, there was the need to create a scalable system, so we had to explore the use of a command programming pattern. Let’s look at how this all works together on a deeper level (Figure 27).

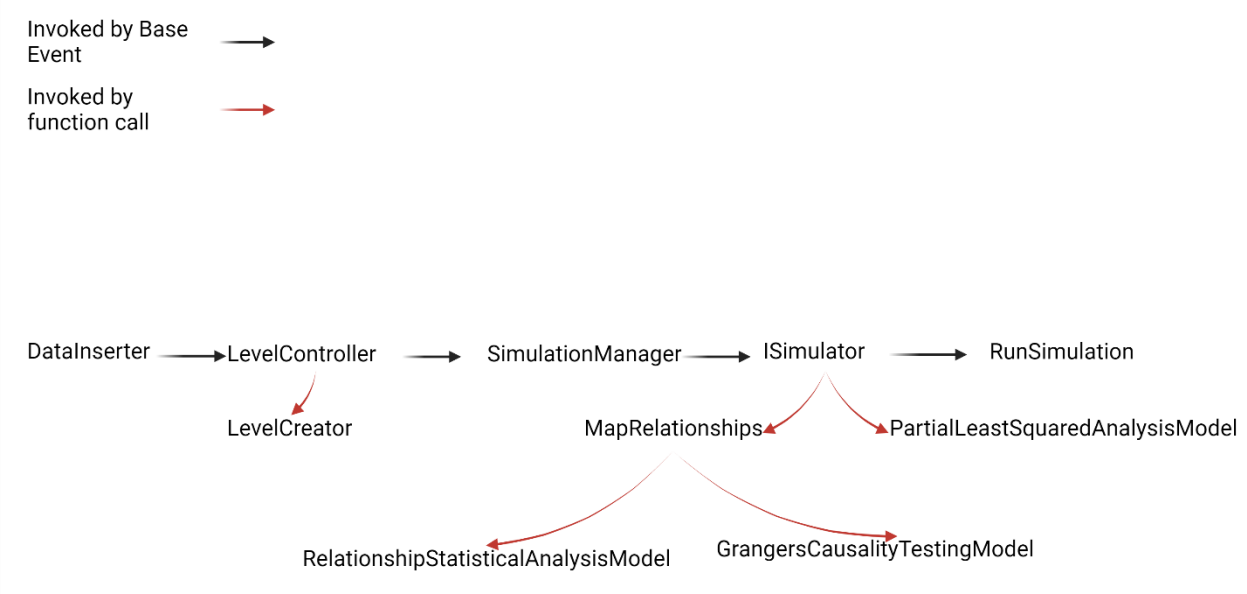

Figure 27: The flow of the simulation. We start by inserting the data. Once the data is inserted, the level is created using the LevelCreator. Then, the Simulation manager prepares the ISimulator and the ISimulator begins to prepare the simulation. It must prepare all the statistical methods (Partial Least Squares, Grangers, and all the relationship statistics). Once done it runs the simulation.

Firstly, we start by inserting data inside the program (for this proof of concept we have data on the client). This is done using the DataInserter class. Once the data is inserted and returned as a CSV data structure, we use the BaseEvent SO to call the LevelController Class. Inside this class, after receiving the CSV data, we create the base level using the LevelData SO. After creating the level, we use the BaseEvent SO to call the SimulationManager. In the simulation manager, we set up the DefaultSimulation and prepare the SimulationConfig and other background tasks such as the ThresholdMeter. After this is created, an event is invoked and that calls the current ISimulator and prepares the simulation. The ISimulator then calls its Init function, maps the nodes to their proper states, and calls the MapSimulationObjects class to position the objects while creating relationships (using the line renderer). Once the initial setup is finalized the user can click on the nodes (SimulationObjects) and interact with them by changing the states and making predictions based on the state changes. They can also use the

AutoDocumetation button to print out documentation based on every move they have made. This is the basis for how the program allows scalability and modifications if needed.

Results

Through preliminary research, we tested the feasibility of this project, and through the actual development, there were a few questions we wanted to answer to prove that we were successful. The questions are as follows: Do individuals have fun when playing our game? Does giving users ownership of the solution help with the black box problem? Can non-scientists understand what is going on by playing the game? Does the game design and ideology make sense? Are there better ways to visualize big data? And finally, is there a way to bridge the connections between non-scientists and scientists alike? Let's look at our final product to understand how we accomplished this goal.

The simulation itself was efficient and worked as expected. We were able to implement a dataset related to the immune response drugs in corona-virus-induced cytokine storm. With this dataset, we analyzed it and created a network that displayed information based on how closely related the genes are. Therefore, we can assume that the tool to create the network itself was successful. However, looking at the original relationships that were inferred in a two-dimensional graph, there were many differences.

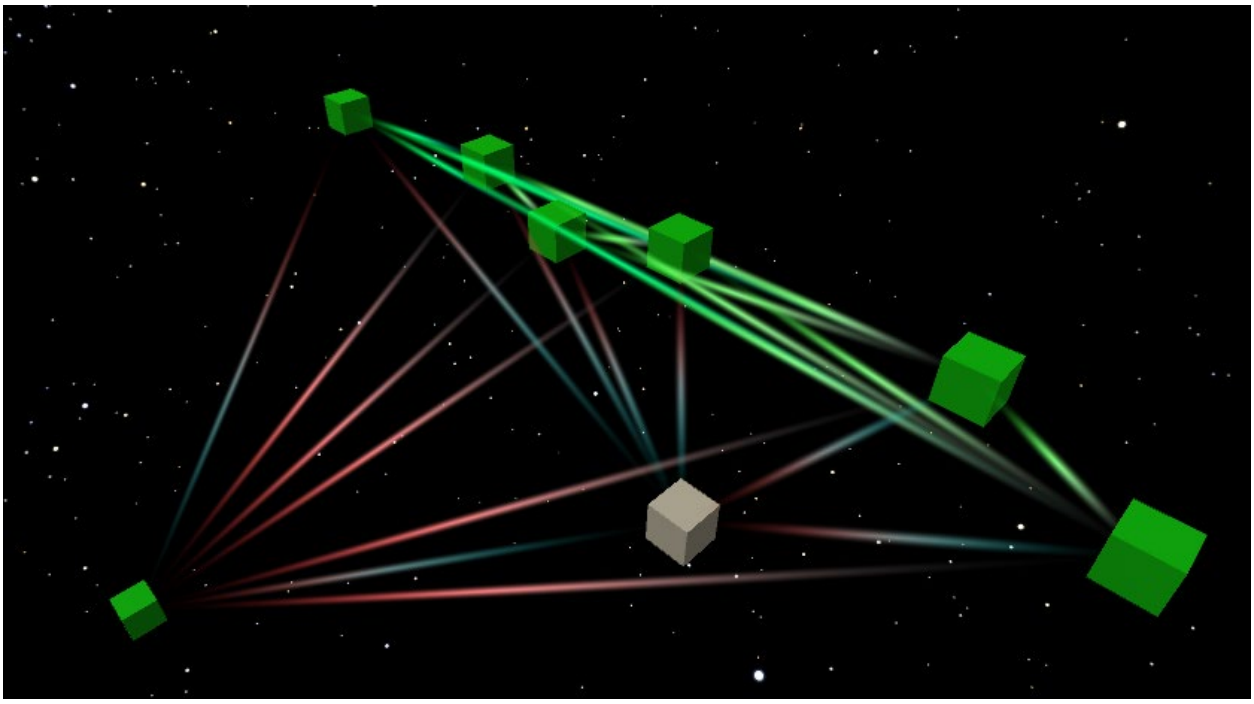

Figure 28: The simulation that we successfully created. By reading the data and conducting many statistical methods, we were able to transform the two-dimensional figure into a three-dimensional simulation.

2D vs 3D data

Looking at the data in two-dimensional space vs three-dimensional space reveals a lot of differences (Figure 28). The data is linear, in 2D, but in 3D you can see the relationships of every node. There are certain nodes in the 2D data that are hard to infer about their relationship with other nodes. If you look at figure 28 and 4, you can see both the 2D and 3D data. In the 2D data image, because there are only two dimensions (x and y) there are only two ways for the relationship to be depicted. Therefore, it looks like “spaghetti”, and it is hard to understand everything that is going on. You must look very closely to differentiate how everything is related to one another. In BioDOV (the 3D version), you can very clearly see how the data is related. It is also dynamic, so it allows users to make new interpretations by making predictions.

If you look deeper into the differences between the two. A lot of the relationships were different, although the data was the same. This was related to the differences in statistics.

However, we were able to see more than just a relationship. We were able to see how they interacted with each other. Through predictions and how highly expressed the nodes were. Since we looked at expression as a continuous value instead of a discrete one, we were able to show the expression level of nodes in the pathway. If one changed, there would be a downstream effect on other nodes. This would allow scientists to attempt to make theoretical predictions about the downstream effect of drugs. In our case, we were able to see just how much a certain node (gene) affected others. This is also related to the reason we allowed an auto-documentation feature. Scientists can see exactly what changes they made to the network and look at the information inside of the text file. This way if they wanted to come back later and repeat the same results, they were able to do so. Thus, we were able to create a system that encompassed feedback and replayability (McGonigal, 2011). Two important aspects are needed in the creation of a game, or the process of gamification.

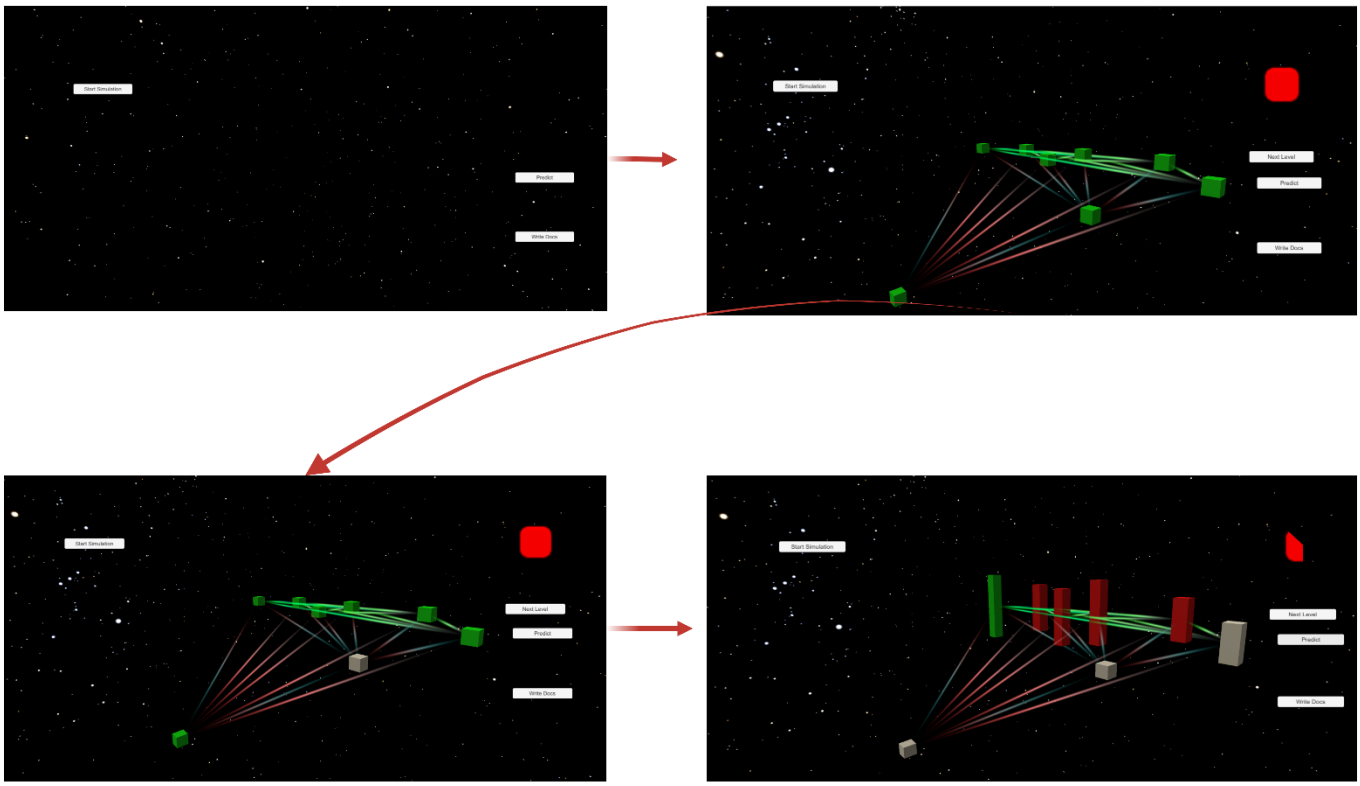

Figure 29: This figure represents the basic flow of the completed simulation. You start by pressing the start simulation button. You then are greeted with a screen that shows information (the amount of information is based on the level). After that, you can press (interact) with a node to change its state. I changed a node from being highly expressed to no expression. Finally, I pressed the predict button and was able to see how that affected the network.

Understanding the Game

Figure 29 is a perfect representation of how the game works. You start with an empty screen, you then start the simulation, interact with a node, and make a prediction. The main goal of the game is to get back to a stable state, which only occurs when all the nodes are white (no expression). This is why the game shows relationships, it is to help individuals make predictions when attempting to acquire this stable state. Each time you make a prediction, a few nodes that are statistically affected by it change. You are allowed to continue doing that until you acquire the threshold, and get the network in as stable of a state it can be in. In Figure 29, the red box in the corner also is a threshold that represents how stable the network is. The fuller the cube, the more stable the state.

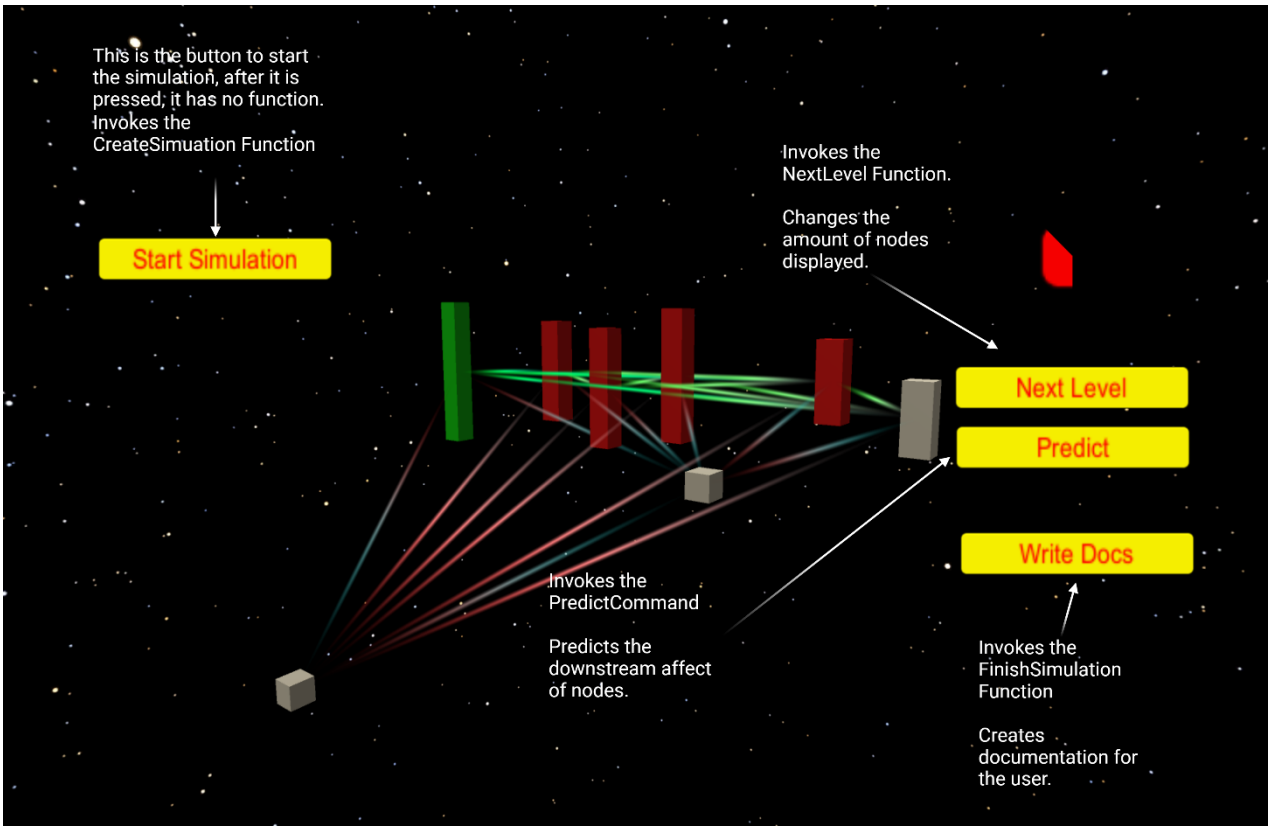

Figure 30: Every button has a different purpose. The StartSimulation button will begin the simulation. The NextLevel button allows you to progress to the next level. As you progress to newer levels, the number of nodes in the network increases. The predict button is called the PredictionCommand. This command makes predictions of all the other nodes in the network based on the changes. Finally, the WriteDocs button writes out the documentation for every change you make in the network (Refer to Figure 14 to see how Auto Documentation outputs).

Why Gamification Worked

For the creation of our tool, we were able to successfully employ proper gamification techniques. How did gamification affect the result we acquired? It impacted the way we perceive the tool and how we the tool will engage and reward other scientists that use it. Let's look at the four features that are required to make a game: A goal, rules, feedback system, and voluntary participation. The goal is the outcome we want players to achieve. The rules are the constraints and limitations required on how the player can accomplish their goal. The feedback system is a system that lets the player know how class they are in achieving the goal. Finally, voluntary

participation is when the player accepts the rules, the goals, the feedback, and voluntary participation to play the game (McGonigal, 2011).

We were able to accomplish these goals as we created the features of our game. Firstly, the threshold meter is how we were able to accomplish our goal. The closer the player is to a baseline state, the more filled it will be. The rules are defined by how the user can make predictions. You must interact with the network and make predictions to change the others. The feedback system is a combination of interactivity, auto-documentation, and how the nodes are characterized after predictions are made. This is how we handle the feedback loop. Finally, voluntary participation will be accomplished when the user inputs their dataset. This means that they want to understand more about the data in a three-dimensional space. By doing this they accept the rules, feedback, and goal of the simulation. This is how we were able to successfully gamify and create a proper proof of concept by following our initial rules. This made our tool very effective for its purpose and allowed us to prove that there are more ways to create engaging visualizations.

Discussion

After the creation of this tool, we reflected on what is next for the future and expansion of this tool. The most important thing to note is that BioDOV is made to be extended. The software was created to be open-ended enough that anyone could add their code to it. This will allow researchers to modify this tool for their data and make their interpretations. If they want to change the type of statistics that affect the dataset, they can do it in the code base. If they want to modify the simulation settings, there is another class called CustomSimulation which only includes empty functions. There are many ways to build upon this tool.

We also are hopeful that scientists will look at this tool and begin to realize that there are millions of avenues to explore regarding visualization. There are so many ways to visualize data instead of using the same 2D graphics every time. Sure, there are visualization platforms like Simularium, but we believe the creation of a tool that can be extended for your use is many times more useful (Lyons et al., 2022) (You can view Simularium at https://simularium.allencell.org/).

To further expand upon this proof of concept, look at Figure 29. The third and fourth pictures show the before and after a prediction was made. This was the prediction made after changing the state of the gene CSF3 from two to zero. CD40, CD80, CD86, and CTSB went from state two to state one (high expression to low expression). CD40, CD86, and CD80 are costimulatory molecules that aid in the initiation of CD4+T cells (Rogers et al., 2003). CTSB mediates cell death (Wang et al., 2023). CTSL and CD200R1 went from state two to state zero (high expression to no expression). CTSL aids in intracellular protein catabolism (Zhang et al., 2022). CD200R1 is involved in the inhibitory pro-inflammatory response (Dentesano et al., 2012). Finally, CCL5 remained the same. CCL5 plays an important role in recruiting leukocytes into inflammatory sites (Aldinucci & Colombatti, 2014). Finally, CSF3 is an important neutrophil-promoting cytokine that impacts neutrophil survival (Ouyang et al., 2020). CD40, CD80, and CD86 all have a negative relationship towards CSF3, but they were downregulated. Although the downregulation was not much (You can tell by the extrusion that they state one, highly expressed), it is still shown.

With this information, we can make additional assumptions, such as how a neutrophil-promoting cytokine affects costimulatory molecules. Or what is the relationship between neutrophils and inhibitors of inflammatory response? The results of this prediction could also be the results of the way we conducted our statistical analysis, but the results are within the realm of

expectations. There are times when we make imperfect predictions using this tool, but biology itself is not standard. There may lie a relationship within this downstream effect that we are unaware of. This would allow scientists to try this in a wet lab by conducting their experiments. That would be the perfect type of result that is expected for this proof of concept.

We want to engage scientists and continue to question the science and results. By giving them information that may lead to more questions, it proves to us that our proof of concept was successful. A tool like this can keep science more dynamic and allow scientists to continue to conduct exciting research. Furthermore, there are more ways that we want to accomplish this.

There were a few extra features we wanted to add to the tool, but the scope would have increased too large. The features we wanted to add were as follows: Customization of Excel files in Unity, importation of any dataset, importation of simulation settings, more eye-popping art, and the choice of statistical test. The customization of Excel files would have allowed us to create the tool with a more dynamic front. The user would not have to clean up their data, they could just choose the columns they wanted to include in the analysis, and it would create the network using the appended data. The importation of datasets is the most important feature, but because we wanted to test proof of concept, we were unable to add this feature.

We wanted users to have the ability to add any dataset, from any collection of data inside of the simulation. This dataset can work with any kind of data, it does not have to be all biological. It can still make inferences based on any data (bearing a few adjustments). The importation of simulation settings (or the customization) would allow users to have more freedom in how they see the data. Maybe they wanted stars instead of cubes. Maybe they wanted the lines or the cubes to be a different color to match their theme. Whatever it may be this would

have given users more options when it comes to the biological network. The importation of simulation settings goes hand and hand with more eye-popping art.

Science and statistical analysis may intrigue the scientific community, but the non-science community won't have the same reaction. So having something that looks pretty would make them more likely to because interested. Finally, the choice of statistical test. I already said if we could modify simulation settings we could add it, but if we allowed users to add their statistical test without having to modify the code base, this tool could be used by people who are not programmers. The addition of all these features would make this tool much more feasible and complete. However, due to the limitation of time, these were unable to be added.

Someone may take this in another direction and make a full game out of this project as well. The simulation is made to be highly dynamic and extendable. It is important to keep in mind that this project can be used for all purposes, but we hope that it accelerates data visualization in the scientific community. We have already explored the differences between this tool and the model in 2D. Many differences could be seen briefly. Can this be improved even more? Can we add more dimensions to this data? It poses questions as to what is currently possible and how we accomplish them.

Maybe data visualization is adopted in a virtual reality simulation. This would make the simulation more interesting to the non-scientific community. However, this would also increase the cost of tools needed to run the simulation. It is possible that this could reveal data that we could not see before. In our case, we truly believe that perception is the most important aspect of data visualization. How people see and process the data is the most important aspect of visualization. You can see this by looking at 2D visualizations. Some people pick up an understanding quickly, while others need time to absorb the information fully. So, by allowing

this tool to be extended, it gives scientists a guideline for how to create many different visualizations in a way only they can.

It's also important to note that for bigger datasets (greater than 250000 rows) we may run into performance issues. We wanted to create BioDOV using a Data Oriented Design (DOD) approach, but it was not feasible. In a nutshell, DOD is a design pattern that takes advantage of cache misses. It structures the data in a way that helps processes avoid cache misses, thereby making the code infinitely more efficient. The DOD package for unity was still in alpha and was not completely ready. This causes us to run into many issues when trying to implement it.

It's very important to understand that this project was used to help combat the black box problem. The black box problem is a neural net issue, it represents the returned data from the neural net. The data is different every time, and no one knows what happens inside the processing. This makes it extremely difficult to use for medical interpretations. Especially, since you want to know exactly why the neural net predicted this value instead of another. In a field where it is crucial to understand everything that is going on, some predictability is hard to rely on. This is why there are confidence intervals before companies can properly prescribe new medicine. This is why we want users who use this tool with medical data to understand everything that is happening. So, we are attempting to give them ownership of the solution. This will help the medical community trust the output data more readily than other methods. If we can successfully aid in finding a solution for this problem, I would instantly consider BioDOV a success.

In the future, I can see BioDOV being used to view intensive biological networks in 3D, specifically for medical data. This would aid in the diagnosis of diseases. It would give doctors and clinicians better perspectives of the pathways that may be involved in the pathogenesis of the

disease. They could then use that to either create new treatments or for prognosis. But in current times, using this tool as a base to further visualization technology to pave the way for the future would be ideal.

This does not mean that gamification must be the only method for new visualizations. It is possible in the future that scientists will find a way to properly display 4D images. Or even properly unveil the use of quantum computing to work with big data. Many pathways could accomplish the future of big data visualization that is needed. However, once visualization is improved upon, many will be able to actively participate in medical research. This will help the community and researchers alike and prevent the issues that happened during the COVID-19 pandemic (the pandemic of misinformation). And anything that can help us avoid that is a wonderful invention. Furthermore, if scientists are more active in scientific communication, and can create things that are hard for people to understand. They could help reduce the spread of diseases and help the field of epidemiology. Anything that uses easy-to-understand images will help teach the non-scientific community about topics they do not know about. But, giving them the power to see visualizations that are easy to understand, and can be interacted with, helps give people the confidence to believe in science. Through this research, we can improve the lives of millions around the world through better visualizations and communications.

Code Availability

All source code for this project is available at https://github.com/YoungKrug/BioDOV. We used the Unity engine (version 2022.3.4f1 LTS), as well as Accord.Net (version 3.8.2) and Math.Net (version 5).

References

Aldinucci, D., & Colombatti, A. (2014). The Inflammatory Chemokine CCL5 and Cancer Progression. *Mediators of Inflammation*, *2014*, e292376. https://doi.org/10.1155/2014/292376

Beck, K. (2022). *Test Driven Development: By Example*. Addison-Wesley Professional.

Conick, H. (2019, July 30). Gamification is Manipulative. Is It Ethical? *American Marketing Association*. https://www.ama.org/topics/ethics/

Dentesano, G., Straccia, M., Ejarque-Ortiz, A., Tusell, J. M., Serratosa, J., Saura, J., & Solà, C. (2012). Inhibition of CD200R1 expression by C/EBP beta in reactive microglial cells. *Journal of Neuroinflammation*, *9*(1), 165. https://doi.org/10.1186/1742-2094-9-165

Doutreligne, S., Gageat, C., Cragnolini, T., Taly, A., Pasquali, S., Derreumaux, P., & Baaden, M. (2015). UnityMol: Interactive and ludic visual manipulation of coarse-grained RNA and other biomolecules. *2015 IEEE 1st International Workshop on Virtual and Augmented Reality for Molecular Science (VARMS@IEEEVR)*, 1–6. https://doi.org/10.1109/VARMS.2015.7151718

Esposito Vinzi, V., & Russolillo, G. (2013). Partial least squares algorithms and methods. *WIREs Computational Statistics*, *5*(1), 1–19. https://doi.org/10.1002/wics.1239

Fan, X., Wang, Y., & Tang, X.-Q. (2019). Extracting predictors for lung adenocarcinoma based on Granger causality test and stepwise character selection. *BMC Bioinformatics*, *20*(Suppl 7), 197. https://doi.org/10.1186/s12859-019-2739-z

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Heberle, H., Carazzolle, M. F., Telles, G. P., Meirelles, G. V., & Minghim, R. (2017). CellNetVis: A web tool for visualization of biological networks using force-directed layout constrained by

cellular components. *BMC Bioinformatics*, *18*(10), 395. https://doi.org/10.1186/s12859-017-1787-5

Heerah, S., Molinari, R., Guerrier, S., & Marshall-Colon, A. (2021). Granger-causal testing for irregularly sampled time series with application to nitrogen signalling in Arabidopsis. *Bioinformatics*, *37*(16), 2450–2460. https://doi.org/10.1093/bioinformatics/btab126

Hussain, A., Shakeel, H., Hussain, F., Uddin, N., & Ghouri, T. (2020). Unity Game Development Engine: A Technical Survey. *University of Sindh Journal of Information and Communication Technology*, *4*.

Ingeno, J. (2018). *Software Architect's Handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing Ltd.

Kadir, S. R., Lilja, A., Gunn, N., Strong, C., Hughes, R. T., Bailey, B. J., Rae, J., Parton, R. G., & McGhee, J. (2021). Science Forum: Nanoscape, a data-driven 3D real-time interactive virtual cell environment. *eLife*, *10*, e64047. https://doi.org/10.7554/eLife.64047

Kolchanov, N. A., Anan'ko, E. A., Kolpakov, F. A., Podkolodnaya, O. A., Ignat'eva, E. V., Goryachkovskaya, T. N., & Stepanenko, I. L. (2000). Gene networks. *Molecular Biology*, *34*(4), 449–460. https://doi.org/10.1007/BF02759554

Lv, Z., Tek, A., Da Silva, F., Empereur-mot, C., Chavent, M., & Baaden, M. (2013). Game On, Science—How Video Game Technology May Help Biologists Tackle Visualization Challenges. *PLOS ONE*, *8*(3), 1–13. https://doi.org/10.1371/journal.pone.0057990

Lyman, C. A., Richman, S., Morris, M. C., Cao, H., Scerri, A., Cheadle, C., & Broderick, G. (2021). Attractor Landscapes as a Model Selection Criterion in Data Poor Environments. *bioRxiv*. https://doi.org/10.1101/2021.11.09.466986

Lyons, B., Isaac, E., Choi, N. H., Do, T. P., Domingus, J., Iwasa, J., Leonard, A., Riel-Mehan, M., Rodgers, E., Schaefbauer, L., Toloudis, D., Waltner, O., Wilhelm, L., & Johnson, G. T. (2022). The Simularium Viewer: An interactive online tool for sharing spatiotemporal biological models. *Nature Methods*, *19*(5), Article 5. https://doi.org/10.1038/s41592-022-01442-1

McGonigal, J. (2011). *Reality Is Broken: Why Games Make Us Better and How They Can Change the World*. Penguin.

Morris, M. C., Lyman, C. A., Richman, S., Cao, H. B., Cheadle, C., & Broderick, G. (2020). Predicting the Immune Response to Repurposed Drugs in Coronavirus-induced Cytokine Storm. *2020 IEEE 20th International Conference on Bioinformatics and Bioengineering (BIBE)*, 458–465. https://doi.org/10.1109/BIBE50027.2020.00080

Ouyang, S., Liu, C., Xiao, J., Chen, X., Lui, A. C., & Li, X. (2020). Targeting IL-17A/glucocorticoid synergy to CSF3 expression in neutrophilic airway diseases. *JCI Insight*, *5*(3), e132836. https://doi.org/10.1172/jci.insight.132836

Pirch, S., Müller, F., Iofinova, E., Pazmandi, J., Hütter, C. V. R., Chiettini, M., Sin, C., Boztug, K., Podkosova, I., Kaufmann, H., & Menche, J. (2021). The VRNetzer platform enables interactive network analysis in Virtual Reality. *Nature Communications*, *12*(1), 2432. https://doi.org/10.1038/s41467-021-22570-w

Rogers, N. J., Jackson, I. M., Jordan, W. J., Hawadle, M. A., Dorling, A., & Lechler, R. I. (2003). Cross-species costimulation: Relative contributions of CD80, CD86, and CD40. *Transplantation*, *75*(12), 2068. https://doi.org/10.1097/01.TP.0000069100.67646.08

Steenbeek, J., Felinto, D., Pan, M., Buszowski, J., & Christensen, V. (2021). Using Gaming Technology to Explore and Visualize Management Impacts on Marine Ecosystems. *Frontiers in Marine Science*, *8*. https://doi.org/10.3389/fmars.2021.619541

Tisoncik, J. R., Korth, M. J., Simmons, C. P., Farrar, J., Martin, T. R., & Katze, M. G. (2012). Into the Eye of the Cytokine Storm. *Microbiology and Molecular Biology Reviews*, *76*(1), 16–32. https://doi.org/10.1128/mmbr.05015-11

Wang, J., Zheng, M., Yang, X., Zhou, X., & Zhang, S. (2023). The Role of Cathepsin B in Pathophysiologies of Non-tumor and Tumor tissues: A Systematic Review. *Journal of Cancer*, *14*(12), 2344–2358. https://doi.org/10.7150/jca.86531

Zhang, L., Wei, C., Li, D., He, J., Liu, S., Deng, H., Cheng, J., Du, J., Liu, X., Chen, H., Sun, S., Yu, H., & Fu, J. (2022). COVID-19 receptor and malignant cancers: Association of CTSL expression with susceptibility to SARS-CoV-2. *International Journal of Biological Sciences*, *18*(6), 2362–2371. https://doi.org/10.7150/ijbs.70172