Complex System Simulation Framework for Shared Augmented Reality Applications

by

Praneet Mekala

B.S. Computer Science and Engineering, Massachusetts Institute of Technology, 2023

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by:	Praneet Mekala Department of Electrical Engineering and Computer Science May 19, 2023
Certified by:	Eric Klopfer Professor and Director of the Scheller Teacher Education Program Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Complex System Simulation Framework for Shared Augmented Reality Applications

by

Praneet Mekala

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Complex systems perspectives are a tool that give an individual the ability to better understand the outcomes of actions on both the environment and other actors within the system. These perspectives are rarely given an opportunity to develop within educational environments, and many students lack this perspective. System simulations to aid in the development of complex systems perspectives have been created in the past. These past simulations, however, fail to provide either an immersive of expansive experience for the students. The We're In This Together (WIT) team is addressing these issues by developing complex system simulations for mixed reality headsets to be used in a classroom setting. This project specifically focuses on the subtask of creating a programmatic interface for simulation data to be updated and shared uniformly across devices. This interface will be usable by developers to create their own customized simulations.

Thesis Supervisor: Eric Klopfer
Title: Professor and Director of the Scheller Teacher Education Program

Contents

List of Figures

Chapter 1

Introduction

At first glance, it may be difficult to find similarities between the location of fishing boats in the Caribbean, the spread of viruses in developing countries, and planning critical infrastructure for building cities. Each of these examples has a limitless number of factors that affect their respective outcomes. Fishing boat locations can be affected by the temperature or the amount of fish present in the area. Epidemics can be affected by city population or healthcare availability. City planning can be affected by environmental conditions or citizen happiness. All these complexities and factors that make them unique unite these systems under one categorical term - complex systems.

1.1 Complex Systems

Studies of complex systems can result in world-changing perspectives on problem-solving [32]. It allows researchers to approach problems with a broader mindset and focus on aspects of systems that are traditionally over-simplified. This perspective enhances an individual's ability to understand how an agent's action within a system affects the system itself. Unfortunately, these perspectives are rarely taught in schools, and students have difficulties understanding the intricacies of complex systems [14]

A **complex system** is more concretely defined as a system with many elements or agents that exchange stimuli with each other and their environment. Complex systems generally display organization without any organizing elements being in place. Any alteration of the system by an internal agent or an external force results in a modified state of the functioning system. In elaborate cases and ones being studied by this project, agents act on a limited amount of information about the system to progress their individual goals.[25]

1.1.1 Example

To help understand what a complex system entails, I will elaborate on the fishing boats example provided earlier. In this example, the agents in question are the individual fishing boats, and the environment is a shared fishing space for the agents to catch fish for profit. Several forces drive this system. Agents can modify the environment directly by catching fish, depleting the environment of its resources. Agents also can communicate with each other to strike deals about where and when to fish, affecting their actions in the future. Additionally, external forces such as fishing policies can affect when and how much agents can fish. An example of this is a policy that only allows fishing boats to be out between 9 am and 5 pm. By limiting the time that fishing boats are allowed to act, the fish population has a greater window of recovery than without the policy. To add to the complexity, perhaps environmental conditions change in the fishing area, such as the water freezing over during the winter and such. All actors in this system have their own goals and can perform actions to modify the environment to achieve those goals [13]. Although this is a small example with a few possible actions outlined, the space of state action pairs for any agent is enormous. For a real-world system with thousands of compounding factors, the number of state action pairs is magnitudes larger. This enormous space makes it unfeasible to mathematically model the system, which is why gaining the complex system perspective is vital to understanding how the system works as a whole [31].

1.2 Mixed Reality

Virtual reality (VR), augmented reality (AR) and mixed reality (MR) are emerging technologies that have the potential to change education in classrooms. Virtual reality is defined as a fully virtual environment in that a user experiences full presence. This means there are no visual stimuli from the real world, and the entire experience is conducted on a headset device. Augmented reality on the other hand uses the physical environment and augments it with virtual elements. Mixed reality is not as well defined as VR and AR, but sits somewhere in the middle of the two, combining both physical and virtual worlds into one [17].

In the recent few years, virtual reality (VR) and augmented reality (AR) has seen increasing popularity in both entertainment and research. In the past 4 years alone, virtual reality headset sales have gone up by over 400% [33]. The rise in popularity is due to three main characteristics of this technology: immersion, perception to be present in an environment, and interaction with the environment. These three factors feed into a sense of realism for the user, aligning the user's expectations with their expectations of reality [16].

Past research on MR used in education settings has shown that the introduction of MR technology in classrooms makes the learning process "more interesting and pleasant for the students" [26]. Through the use of a virtual environment, students can make decisions affecting the environment without needing to worry about facing real-world repercussions. This allows students to get hands-on experiences on topics they struggle with, which has been shown to engage students more in what they are learning and develop their critical thinking skills [29]. Mixed reality technology is an effective tool that can improve students' learning experiences in a classroom.

1.3 We're in this Together

We're in this Together (WIT) is one of the projects within the MIT Scheller Teacher Education Program (STEP). The goal of this project is to create a framework that external developers can use to easily develop personalized simulations. This framework is being built in Unity, as Unity provides an interface with thousands of functions that allow for visual virtual entities to be modifiable by script. WIT has been split into three separate components, that each interact with each other in unique ways.

1. The first is known as the view layer. The view layer has two main goals. The first goal is to ensure that all members in the simulation have synchronized perspectives. One of the biggest challenges of creating a shared augmented-reality simulation is that coordinate systems on individual augmented-reality enabled do not automatically align. If an object within the shared simulation is supposed to appear in a certain location on one device, it will likely not appear in the same location on another device. The second goal of the view layer is to access real-world objects in the headset's surroundings and represent them locally. This is to allow the simulation to interact with the real-world environment, creating a more immersive experience for the users.

2. The second layer is the simulation layer. This layer is a lower-level layer that runs the simulation behind the scenes. The goal of this layer is to be able to run a complex simulation efficiently. This includes rules to update the state of the environment and Unity's GameObjects. While users experiencing the shared simulation will not be directly affected by the consequences of this layer, this layer is the main interface that simulation developers can use to create their own simulations.

3. The third layer is the interactions layer. The goal of this layer is to create mechanisms for users within the simulation to modify virtual objects in the simulation in similar ways to real-world interactions. This layer will define various interaction types and create objects that can link actions from the user with modifications of the internal simulation.

A full diagram of the three layers and their interactions is shown in Figure 1-1

WIT is a three-year-long project that will be completed in 2025. The goal for the developers within the first year is to create a simulation framework from end to

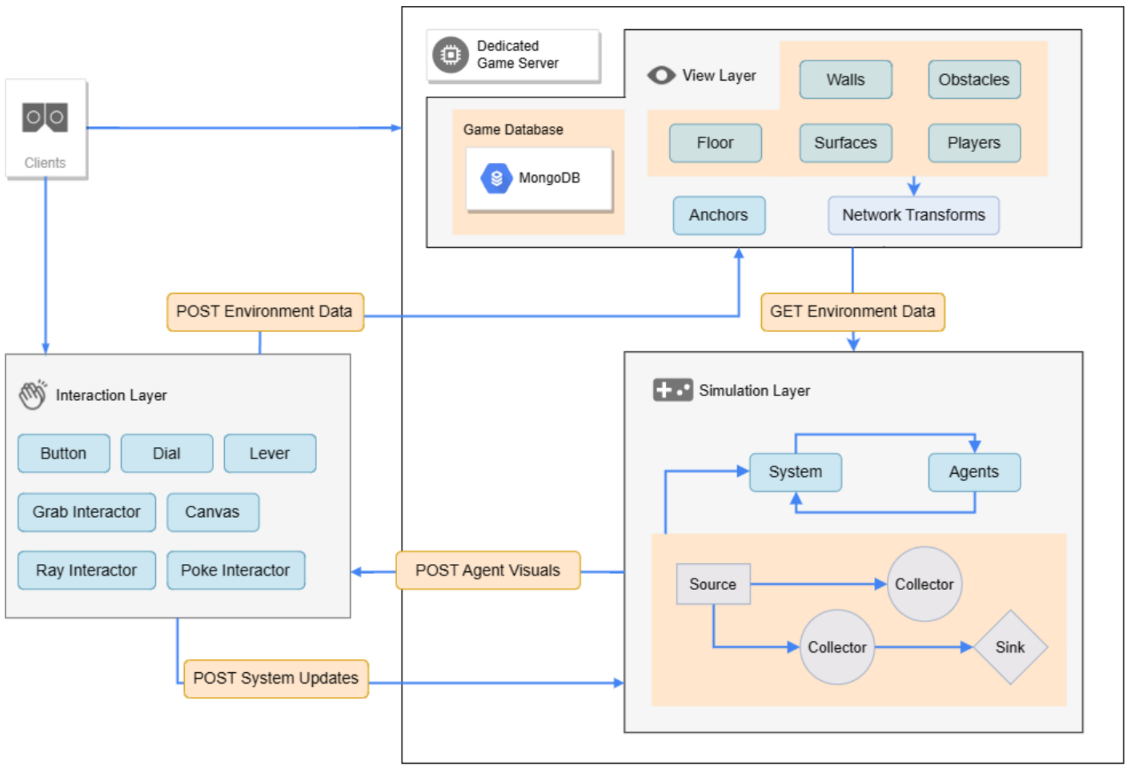

Figure 1-1: Diagram showing all three layers and their interactions within the system [37]

end that allows for these complex system simulations to be created. The current developers will only be working on this project for a year, at which point it will be handed off to another group of developers. It is important that within this first year, the design of the created layers is well thought out, in order to ensure a seamless transition between developers. As a result, the three layers defined above

There are two main types of hardware that the WIT team is using for this project: a virtual reality headset with pass-through AR features and a server to store all system data. The headsets have inbuilt cameras that allow for stereo vision of the surroundings on the internal display, allowing for virtual objects to appear on screen with the camera feed [36]. Each student in the classroom will be equipped with a separate headset, each of which is connected to the server, a computer running the system code. We choose to use a local server rather than a hosted server to minimize any latency issues that could be present. Students will be able to view the environment determined by the server code and interact with it by sending requests directly to the server over the network that will be set up.

Each headset has a unique set of features and unique SDKs to use for development. The current implementation of the WIT framework uses Meta Quest Pro headsets, which have features vital to the development of the end goal, namely cloud anchors [9] and pass-through AR [6]. Purchasing Meta Quest Pros for classrooms full of students will be infeasible for the team budget, so future work in choosing a cheaper headset and adapting the framework to use the new headsets' SDK must be done. Currently, the only requirement for server hardware is a Windows 11 operating system. Beyond this, further research must be done to determine hardware limitations with respect to the created framework.

For more information on the view layer, read Ellen's thesis "Capturing Worlds of Play: A Framework for Educational Multiplayer Mixed Reality Simulations" [37]. For more information on the interactions layer, read Anna's thesis "Designing Student Interactions To Explore Systems Thinking in Augmented Reality" [38].

1.4 Problem

As the use of Augmented and Virtual reality grows in educational contexts, the need for frameworks for application development on the headset will follow. The goal of the WIT project is to provide one of these frameworks specialized in creating complex system simulations. As the space is still growing, currently there exist no frameworks that can create these shared augmented reality complex system simulations. As a result, the WIT project, composed of its three layers, must be built from the ground up. The focus of this thesis is on the simulation layer that will be created. The simulation layer must display the following properties:

1. The ability to run simulations to modify GameObjects in Unity by following common complex system simulation paradigms
2. The ability for an external developer to be able to define customized simulations using the framework provided in an intuitive manner
3. The ability for the simulation layer to interact with the view layer and the interaction layer to have simulation components be affected by the real world
4. The ability to have its feature set expanded easily as new abstractions for complex system simulations are created

Creating a simulation layer that follows these properties will be one of the main stepping stones that will allow for complex system simulations to be created in an efficient and effective way by external simulation developers.

Chapter 2

Related Works

Complex system simulations for classroom use have been experimented with in the past to explore the impacts of these systems perspectives on student thinking. The goal of these is to immerse students in agent-based roles of the simulation to make decisions and understand the resulting outcomes[34]. In the past, there have been two main types of systems simulations studied. The first is physical system simulations. These involve creating a system in the physical world that allows students to move around freely while interacting with the real world to understand how their decisions affect other members of the system[18]. This method allows for sensory immersion due to interactions with the environment being entirely physical. Physical simulations, however, are limited to a small space of possibilities due to the resources required to set them up and are also limited in their ability to convey global changes to the environment due to an individual's actions. The second type of simulation studied in the past is computer simulations. These involve students taking the role of an agent on a computer-based simulation and running the simulation entirely virtually, interacting with other agents through a network [35]. While these have a greater degree of freedom in terms of simulation capabilities, they lack sensory immersion due to all interactions between students and the environment being done through a computer screen.

We plan to build upon these approaches by combining aspects of the two that make the simulation more enticing to the target audience. By running the simulation in a virtual mixed-reality world, we enable sensory immersion through the use of spatial anchoring of objects and headset audio, while enabling a full range of freedom for simulation development. Combining these traits from the approaches taken in the past will allow us to create a simulated system that negates the downsides of each of the approaches individually.

Chapter 3

Networking

One of the key decisions that needed to be made before creating the simulation layer of WIT was the networking solution to be used. This would dictate what types of data needed to be stored and how data would be transferred between the server and clients, as well as between two different clients.

3.1 Requirements

There were 3 main factors taken into consideration when choosing which networking solution would be used:

1. Robustness. We were seeking a networking solution that had near limitless capability in terms of network architecture, as we did not have a finalized use case or network model in mind yet. As design decisions in other areas would be made, such as data size and hardware capabilities, the networking solution would need to be able to adapt to our network needs.

2. Complexity. Although networking was an important part of the WIT project, it was only one piece of the component and all members of the WIT team would have other tasks to accomplish to make the project come together. As a result, the time it would take to understand the intricacies of a complex network solution could instead be put in places to make the overall simulation

achieve WIT's main goals. Additionally, since members of WIT will constantly be changing over the multi-year lifespan of the project, we sought a simpler networking solution such that new members would not need to dedicate as much time relearning the technology.

3. Support. This encompassed two main ideas. Firstly is the support of the networking solution by its own developers. Since WIT is a multi-year-long project, we wanted to choose a solution that would be kept up-to-date by its developers and could potentially be expanded in terms of technical ability and ease of use. Also, in case of bugs with any libraries or methods, we wanted the ability to contact the solution's developers to ensure a speedy resolution. Secondly was the support for new developers. Having lots of documentation and other projects to reference online would mean that issues that would come up during development could be solved relatively easily.

3.2 Architectures

Before searching for networking solutions to use for the project, we needed to understand and analyze different network architectures and how they would fit into WIT. There are three standard architectures commonly used in networked games and simulations that were considered.

3.2.1 Peer-to-peer

A peer-to-peer network architecture takes a distributed approach to client communication. In this model, clients are able to communicate with each other directly in order to exchange data (See Figure 3-1). This can be done manually, through cabled connections between devices, or over a wireless network. Additionally, each client has the ability to affect a subsection of the game/simulation, meaning clients will be constantly communicating with each other to ensure their local copies of the game/simulation are up-to-date. [28]

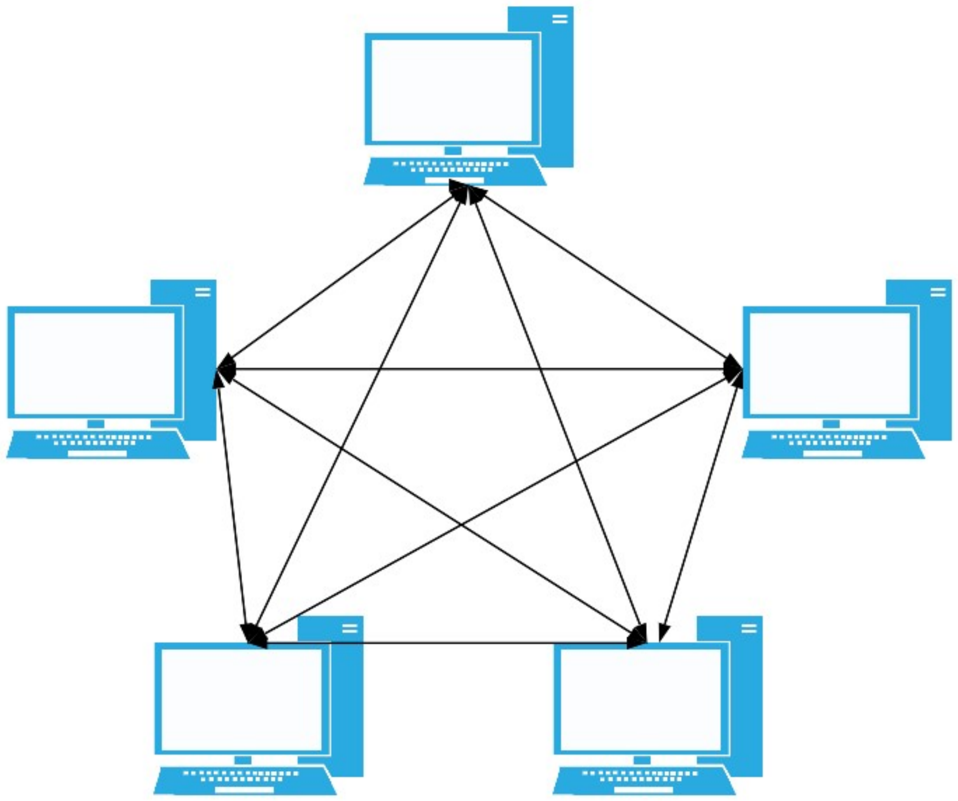

Figure 3-1: Peer-to-Peer connections. All devices on the network are able to send and receive data from any other device.

This network architecture was quickly ruled out for the WIT project for one main reason: using a distributed computation system would mean that there would be no single source of truth. In an environment designed to ensure that participants would be able to see the same simulation state as each other, a single source of truth would be a necessary feature. Due to this reason, a peer-to-peer architecture was not taken into consideration for the final implementation.

3.2.2 Client Hosted

In a client-hosted model, one client acts as an authority and communicates with other clients to alter the game state. This way, each client aside from the host only needs to communicate with one other client (the host), reducing the overall network

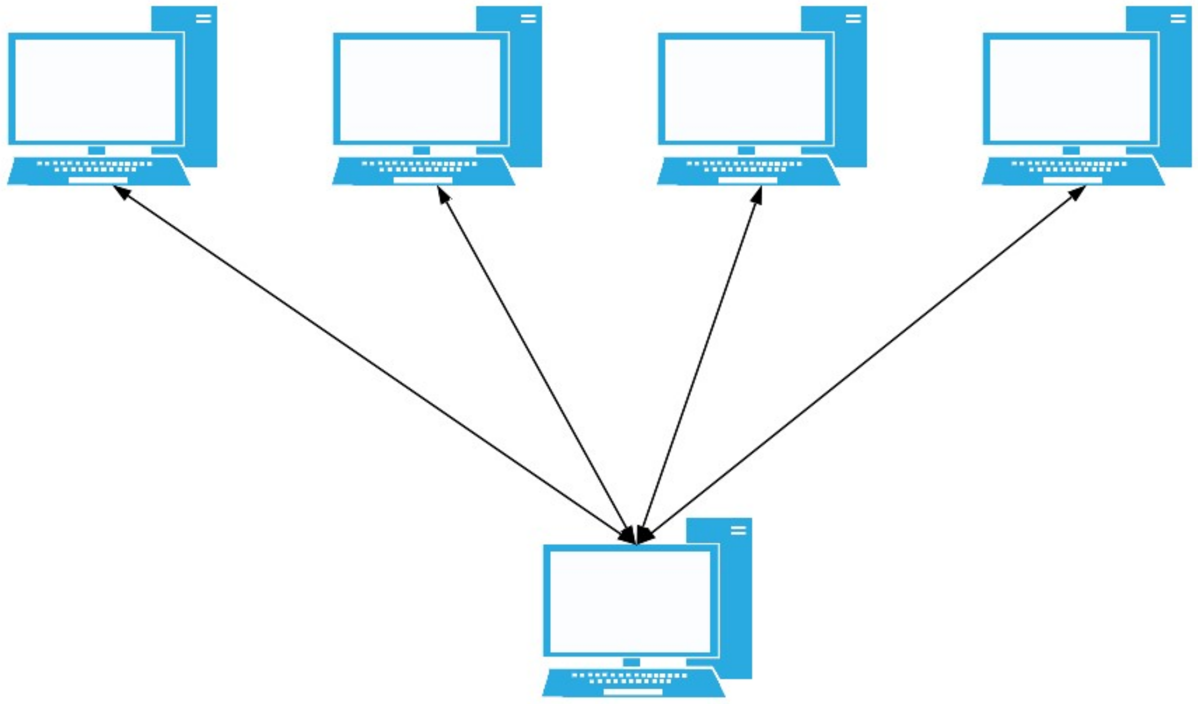

Figure 3-2: Client-hosted connections. All devices on the network must communicate by going through the client.

traffic compared to a peer-to-peer model (See Figure 3-2). This, however, means that the hosting device will perform a majority of the calculations required for a game/simulation to run.

While a client-hosted model solves the issue of not having a single source of truth that the peer-to-peer architecture faced, it introduces a new issue of hardware capabilities. The headsets that were used for testing (which were higher-end headsets that would likely not be used in a classroom setting) had a limited 6GB RAM. To test the limitations of the headsets, objects with positional updating logic were spawned until the lag was noticeable. The smooth performance capped at approximately 30 virtual objects, which we predict would be less than most complex simulations would need. As a result, due to the hardware limitations of the headsets, a client-hosted model was also not taken into consideration for the final implementation.

3.2.3 Dedicated Server

The final network architecture that we looked at was a dedicated server model. In this type of model, a device is set aside to perform all computations and send resulting data to the clients (See Figure 3-3)[22]. This is very similar to a client-hosted model, with the exception that the center of computations no longer needs to be a client itself. Many modern game companies with access to powerful computers to use as servers use this architecture for their games for two main reasons. Firstly, it prevents malicious inputs from clients since the server can only operate on a developer-defined set of operations, but more importantly, it allows all users to share a consistent state [11].

A dedicated server model solves the problem introduced in the previous section with hardware limitations being an issue. The server architecture means we are no longer limited tp having a headset perform simulation calculations. Instead, we can set up a computationally powerful computer with more memory than the headsets as the server and have all headsets communicate with this computationally powerful computer. A dedicated server, however, is not a perfect solution. Since the server code is being run on a desktop computer instead of a headset, the same build that was used for the headset cannot be used on the server. In order to optimize computation speed and not waste resources on the server, all visual elements of the headset build must be stripped away, leaving an un-interactable build on the server [12]. This means that any alterations to the server-side simulation (such as pausing the simulation) must be done by sending a message from a headset to the server, altering the server's simulation state based on the contents of the message, and sending the altered simulation state back to the headset. This effectively doubles the necessary amount of network traffic on server-sided operations. We predict, however, that in our simulation, server-sided operations will be rare and this extra network traffic will not have a significant impact on the network.

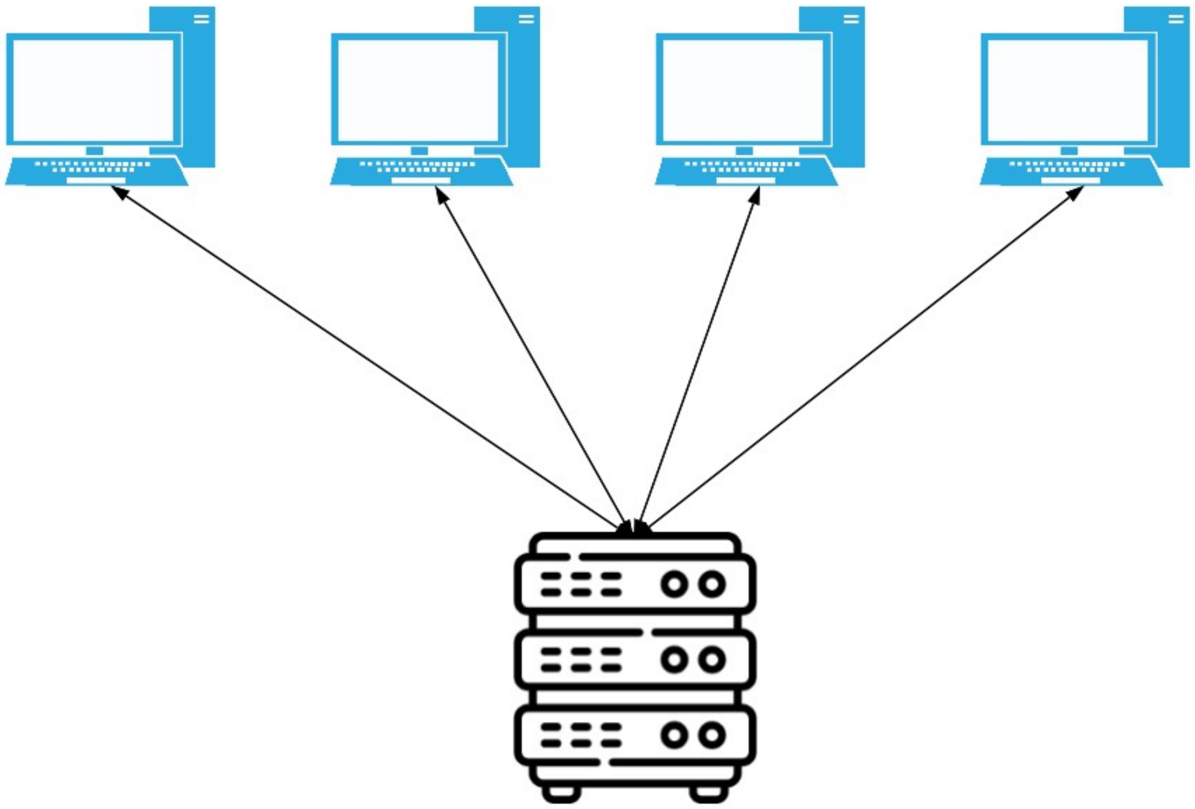

Figure 3-3: Dedicated server connections. All devices on the network must communicate by going through the server.

3.3 Products

In the search for a networking solution that could satisfy all three of the goals stated above, we came across several products that satisfied the requirements we set to varying degrees. For each product, some research and minor development were done to get an understanding of the benefits and drawbacks they could provide when completing our projects.

3.3.1 Netcode For GameObjects

The first networking solution that we looked into was Netcode for GameObjects. Netcode for GameObjects is a networking solution created in late 2022 by Unity Technologies. It provides users with an abstract library for sending GameObject data as well as global scene data across the network [20]. Netcode for Gameobjects additionally supports multiple network architectures, including peer-to-peer models, dedicated server models, and client-hosting models [27]. As part of understanding the intricacies of Netcode for GameObject, we created a demo centered around a client-host model. During this process of creating a demo, we came across multiple pitfalls that made Netcode for GameObjects a difficult networking solution to work with.

Since Netcode for GameObjects is meant to be a standalone solution that gives developers a high degree of freedom for designing their network, there were many features that needed to be implemented which other solutions abstracted away. This included technical details such as having to programmatically determine the server's IP address, or choosing packet sizes/sending rate. These are important features when developing an entirely independent application using customized server settings, however, were not necessary for us as other solutions (discussed below) abstracted away these specifics. Furthermore, because Netcode for GameObjects does not use any intermittent servers for communication (only devices the user specifies), there were multiple issues with respect to sending data across public networks. Many public networks limit what devices can send and receive certain data. On the MIT public network, for example, we needed to request special permission in order to use certain devices as a server in the demo. We predicted that this would be an issue for WIT's goal of making a framework for educators to create simulations on, as educators generally operate on public networks, and would likely run into similar issues.

Additionally, Netcode for GameObjects' release in October 2022 meant there was very little documentation and prior projects to learn from. During the development of the demo, whenever there were issues that had no immediate solution, finding solutions was a very time-consuming process. Perhaps in the future, as documentation improves and more projects utilize Netcode for GameObjects as their networking solution, WIT can slowly transition to using Netcode for GameObjects as its own networking solution. For now, however, due to the aforementioned issues with this networking framework, WIT has decided to seek other options for our networking solution.

3.3.2 Photon Unity Networking

The second networking solution that we explored was called Photon Unity Networking (abbr. PUN). It was released in 2018 and has been one of the main networking solutions for Unity ever since. PUN can be thought of as 2 separate layers – a communications layer that handles the low-level details about how the clients communicate such as transport protocols and custom server settings, and a data layer that sits on top of the communication layer and handles which data is sent over the network as well as how data interacts with clients. PUN allows for development in both layers to provide its users with a high degree of flexibility when designing their network. However, PUN also provides a default implementation for the communications layer which can be minimally modified based on the developer's needs [8].

Due to its vast presence in Unity multiplayer projects, PUN is very well documented and has lots of resources to help developers deal with issues that come up while developing multiplayer games. PUN does have its downsides, however. It is built upon using a peer-to-peer network architecture rather than a dedicated server architecture as needed by the WIT project [2]. Although a peer-to-peer network architecture can be cleverly modified to be turned into a dedicated server architecture, the tipping point for not choosing this solution was that PUN is being discontinued by Photon in favor of a newer networking solution. Because of the lack of future support for PUN, we decided to step away from it and seek other networking solutions.

3.3.3 Photon Bolt

Photon Bolt is another networking solution developed by Photon. Bolt was released in late 2016 and also grew to become one of the most popular networking solutions for building multiplayer games. Unlike PUN, however, Bolt uses a server-based model, supporting both client-hosted and dedicated server type architectures [1].

Bolt additionally abstracts away much of the communication side of development, meaning details on how devices are connected over the network are not necessary to implement. There were two major downsides to Photon Bolt. Firstly, similar to PUN, it is also no longer being supported by Photon in favor of more modern networking solutions. Additionally, being an older networking solution, we feared that Bolt would not be optimized for modern network capabilities. As a result, Bolt was quickly taken out of consideration from the choice of networking solutions for the WIT project without further exploration or the creation of demos.

3.3.4 Photon Fusion

Photon Fusion was the final networking solution that the WIT team considered. Fusion is the most recent networking solution developed by Photon and is meant to be an all-encompassing network solution for unity development. Fusion supports all types of network architectures outlined in section 3.2. According to the Photon website, Fusion was developed with the intent of evolving and replacing both PUN and Bolt, which is why both PUN and Bolt are no longer being supported by Photon [3]. Similar to both PUN and Bolt, Fusion abstracts away much of the lower-level detail necessary for creating multiplayer games. For example, in a dedicated server setup, a developer can specify which device is the server and which devices are the clients. If the developer wants to send data from the server to a client or vice versa, they only need to specify the data itself (ensuring it is in a network admissible format), and the client ID (which is assigned by Fusion when the client joins the network). All other communication information, such as data compression and client-side prediction, is abstracted away from the user and is not necessary for development.

Due to Fusion being one of the more recent networking solutions to be released on the market, it shares some of the same issues that Netcode for GameObjects had. Specifically, documentation for Fusion is still improving and very few public projects that use Fusion exist that can be used as reference. On the other hand, however, since Fusion was built by Photon as a replacement for PUN and Bolt, any of the features and concepts in Fusion were shared by PUN and Bolt as well [3]. Although the order solutions have many differences from Fusion, the documentation for PUN and Bolt could be generally used to help understand concepts in Fusion that we were confused about. Additionally, Fusion provides its users with helpful tech demos showcasing the ability of Fusion in different contexts. One of the demos that was provided was a VR demo with a shared server architecture, which we were able to utilize to assist us in the initial architecture setup phase [5].

The feature set that Fusion provides developers with was extremely useful for achieving WIT's goal of making a framework for simulation development. When making a simulation framework, the simplicity of developing using that framework must be kept in mind. Fusion provides developers with some basic functionality that are universal to developing multiplayer games/simulations. This includes features such as auto-syncing object locations in the environment, as well as sending data between devices [3]. On top of this universal feature set, however, Fusion contains other features that made the creation of a simulation framework significantly easier. One of these features was a tick-based model for syncing clients to the server. Unity natively uses a tick-based model to render and update the scene locally [4]. When creating the simulation framework for an external developer to make simulations on, we operated under the assumption that they would have some basic level of Unity experience. Because local Unity development and networked development on Fusion both share a tick-based model, we could have the developer leverage the use of Fusion's tick based model by abstracting away all the network-specific details in the simulation framework. This would allow the developer to create a simulation in a near-identical way as they would in a local Unity setting, making it easier for them to develop simulations on the framework.

Another unique feature that Fusion provides is inbuilt client-side prediction [4]. One of the goals of the WIT team is to ensure a comfortable experience for users that wear headsets. It has been shown that lag in VR headsets can cause some users to feel nauseated. A virtual environment lag as short as 48ms can cause users to feel dizzy and induce motion sickness [21]. In a situation where a user is grabbing and moving a virtual object, if the object moves 48ms after their hand does, they are likely to experience dizziness. In a networked setting, a delay of 48ms between a client sending input to the server and subsequently receiving an update is almost inevitable. Because of this, we need to ensure that visual lag is minimized when creating a simulation on the network. Fusion's client-side prediction feature helps us with this task. Figure 3-4 shows the difference in a dedicated server architecture with and without client-side prediction. Although in both scenarios, the server sends an updated position to the client at the same delay after the client's input, when client-side prediction is used, the client does not observe this delay at all.

Fusion follows many of the same abstractions as Unity does. For example, in order to make a GameObject have a synced position across devices, a NetworkTransform component can be added which defines the position and rotation on the network, similar to how a Transform component defines the position and rotation locally. Similarly, a NetworkRigidBody component allows for physics computations to occur on the network, similar to how a RigidBody component allows for physics computations to occur locally. Because Fusion and Unity share many of the same abstractions, by including networking details as part of the simulation framework, developers that use the framework can follow the same experience developing networked simulations as they would developing local simulations in Unity. This would make a developer's experience in creating networked simulations just as easy as creating local simulations

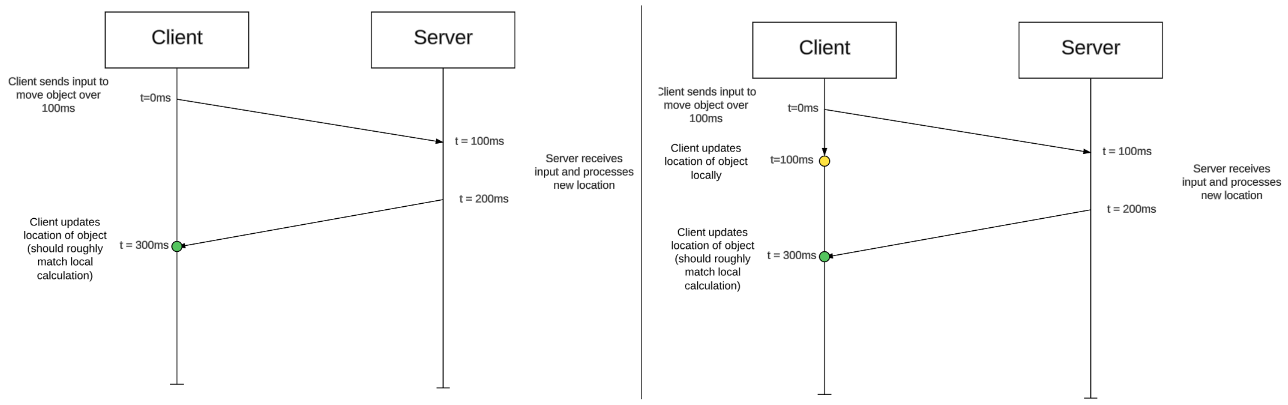
Client
Server
Client sends input to move object over 100ms
t=0ms
t = 100ms
Server receives input and processes new location
t = 200ms
Client updates location of object (should roughly match local calculation)
t = 300ms
Client
Server
Client sends input to move object over 100ms
t=0ms
t = 100ms
Client updates location of object locally
t=100ms
Server receives input and processes new location
t = 200ms
Client updates location of object (should roughly match local calculation)
t = 300ms

Figure 3-4: Client-side prediction. Left: Network without client-side prediction. Right: Network with client-side prediction. Green represents time when client gets the network result of the input, yellow represents time when client computes local result of input

in Unity. Along with other abstractions provided by the simulation framework, using Fusion will allow developers to create shared augmented reality simulations with relative ease.

Chapter 4

Simulation Engine

The main goal of this project was to make a simulation engine that would let developers easily create interactable augmented reality simulations for classroom use. In this section, I detail the goals, interface, and design decisions I faced when creating the simulation engine.

4.1 Goals

Before even thinking about how to implement the simulation engine, the first step in the process was defining what the end goals for the project were. These goals would be a combination of both WIT's goals in creating an end-to-end platform for a shared augmented reality experience in classrooms, and goals specific to just the simulation aspect of the project itself. After deliberation with the WIT team, we came up with the following list of goals that the simulation engine needed to satisfy:

1. Ability to create simulations. The core of any simulation engine is to provide developers with a set of tools to set up the simulation itself. Within the simulation engine framework, we needed to provide developers with a way to specify what components they could include in their simulation and how those components interact with each other. This had to be done in a generalized way such that developers were not constrained in the types of simulations they could produce, while still providing developers with specialized tools and functions to ensure

they would not need to start from scratch. Finding this balance of abstraction vs specificity would be the basis of creating the simulation engine and would define the interface that would be provided to developers.

2. Ability to run simulations in a predictable manner. Once developers set up a customized simulation, the next step is to run the simulation. Our simulation engine needed to run in a way that would be expected by the developer. This would mean handling interactions between developer-defined components properly, as well as modifying developer-defined values in ways that the simulation developer could easily understand. Fulfilling this goal would entail processing all parts of the simulation in line with the full interface we provide developers with

3. Ability to interact with simulations. One of the main focuses of the WIT project is for the simulation to be interactive. Interactable simulations are predicted to cause a heightened sense of immersion within individuals participating in the simulation. Interactability with simulations can take on countless forms, such as modifying flow rates between components, or moving objects from one location to another. Similar to the simulation creation goal, an interface for modifying the simulation would need to be specified that is abstract enough for supporting several different types of interaction processes, while still being specific enough to provide developers with a set of functions to define interactions so they do not have to start from scratch.

4. Ease of simulation development. This goal centers around the developers' experience when creating a simulation. Unity provides developers with an unbounded amount of functionality and freedom to create whatever the developer desires. This, however, comes with the cost of simplicity. In order to create a complex system simulation in Unity from scratch, developers need to deal with the inner workings of Unity and potentially learn hundreds of functions. This is a time-consuming process and not all educators who want to create complex system simulations have the time or patience to learn these Unity concepts. As

a result, the simulation framework we provide to developers needs to be easily understandable while still providing developers with a full set of functionality to create simulations from.

5. Ease of expansion. As stated previously, WIT is a long-term project with team members constantly changing throughout its multi-year lifespan. Within this first year of development, small-scale demos are being created to test the functionality of different components. It is likely that within the next few years of development, the feature set of WIT will be greatly expanded. As a result, we need to design a simulation engine in a way that can be easily understood and further expanded upon by future members of WIT.

This list of goals is not an exhaustive list of factors that affected design decisions. Other factors, such as designing the simulation engine for integration with other members' work, also affected some of the choices made, however, these 5 factors impacted the design of the simulation engine the most.

4.2 Complex System Simulation Models

There are two main mathematical modeling paradigms when it comes to complex system simulations: system dynamics models and agent-based models. In this section, we provide details about both types of models, as well as providing an analysis of which type of model fits the learning goals of WIT the best.

4.2.1 System Dynamics Models

The first paradigm of complex system modeling is system dynamics modeling. This type of modeling excels at understanding the structures and dynamics of complex systems. It uses the concepts of state variables and information feedback loops to determine how a system changes. At a high level, it can be thought of as a series of "stocks and flows". Stocks are containers that hold a given amount of some basic unit. Flows are the rate at which basic units move in and out of stocks [30]. For

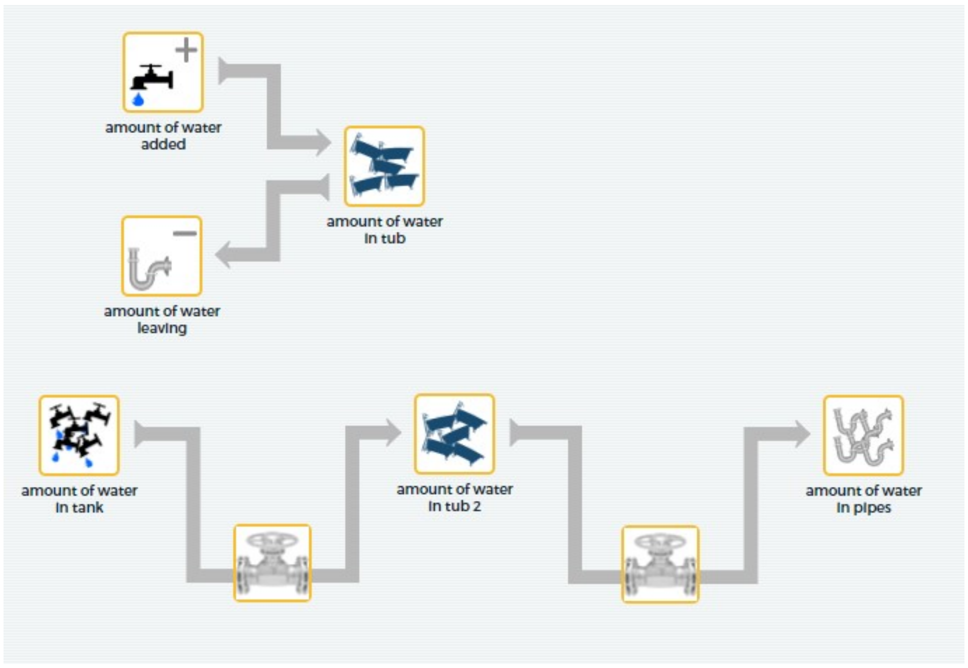

Figure 4-1: An example of a system dynamics model, created in the sagemodeler tool.

example, a water system can be modeled with the systems dynamics paradigm. In this case, we would have one basic unit: water. Stocks in this scenario could be a well, a lake, or even clouds in the sky. Flows, on the other hand, represent the flow rate of water between two stocks. An example of a flow would be rainfall, transferring water from clouds into a lake.

System dynamics models are powerful tools to help understand how a system works on a macro-scale. In the example above, we can imagine expanding the water system to contain hundreds of stocks and flows. By varying the amounts of the basic unit within a stock, or changing the rate of a flow, we can study how modifying one component in the system dynamics model affects all other components in the system. In theory, with a detailed enough water systems dynamics model, we could answer questions such as "How would higher evaporation rates due to global warming affect how much water reaches my home?"

This system of stocks and flows that make up a systems dynamics model can be represented mathematically as a differential equation [19]. Suppose we have a vector S which is a stacked vector of state variables (amounts contained in stocks, environment variables, etc). Then, we can represent the system dynamics model as

$$\frac{dS}{dt} = F(S)$$

where F is a non-linear vector function that represents the flows between the different stocks. Since F is non-linear, however, the differential equation becomes analytically intractable [19]. As a result, the system dynamics are best approximated using a forward Euler "step" function:

$$S_{t+1} = S_t + \Delta t F(S_t)$$

The fact that the function F is a multivariable non-linear function is what makes system dynamics models so robust. There are two important consequences of this fact. The first consequence is that flows are not necessarily symmetric. The amount of inflow within a system is not necessarily equal to the amount of outflow. This is a necessary feature if multiple basic units are involved in the system. For example, suppose we wanted to expand our water example to include crop amounts in farms. Suppose that 100 water units from a well are converted to 1 crop within the farm. The flow in this case would remove 100 units from the well but only add 1 unit to the farm. Due to the asymmetry of flows, we can effectively model a system containing an arbitrary number of units. The second consequence is that a flow can be impacted by any number of variables. The amount of a basic unit that moves between two stocks can be a function of any of the state variables of the system. Suppose, in our example, a farmer owns two farms in different locations, Farm A and Farm B. If Farm A contains 10 units of crops, the farmer wants to stop all production on Farm B. This can be modeled by setting the inflow of Farm B as a function of the state of Farm A. These two consequences make system dynamics a robust option for modelling how large-scale systems change over time.

One of the shortcomings of system dynamics models is the difficulty in understanding and implementing decision-making. While systems dynamics models are useful for studying how changing one component in a system affects another component, it falls shorts of identifying why those things happen [10]. Expanding upon the example we have been building up, suppose the farmer has hundreds of farms in different locations, each with different relationships with each other. The farmer makes decisions on what to do with one farm based on the states of all other farms. While these decisions could be encoded using functions similar to the case with two farms, this would become a cumbersome task and difficult to get correct. On top of this, extracting information such as what decision is being made by the farmer at any given moment is not possible with the current model described. As a result, systems dynamics models do a poor job of studying when actors in a system perform certain decisions and why.

4.2.2 Agent Based Modeling

Agent-based modeling is another complex system modeling paradigm that splits up the system in terms of its constituent parts called agents. Agents are equipped with various properties and can interact with each other and the environment in predefined ways [24]. In order to understand agent-based modeling at a greater depth, we must first define what exactly an agent is.

There are many ways agents have been defined in past literature, each of which contains some universal properties. At a fundamental level, agents display the following properties[15]:

1. Agents are autonomous. Within agent-based models, agents must be able to operate without external stimuli. More specifically, agents contain a set of behaviors by which they operate on. They may display functional logic based on other elements they "sense" in their environment, but actions must be taken on their own.

2. Agents are self-contained. An agent must be an identifiable entity with clear boundaries. For each element in the agent-based model's state, it must either be a part of an agent, or not part of an agent.

3. Agents are able to interact with other agents. As stated before, agents are created with a predefined set of behaviors. These behaviors describe how an agent interacts with its environment. Agents are often equipped with the ability to "sense" other agents' properties and environmental properties.

This set of fundamental properties is what defines an agent in agent-based simulations. This open-ended definition of agents gives developers a high degree of freedom in designing and running their simulations. Other properties that agents commonly (but not necessarily) have are as follows [15]:

- Agents are often given goals that their behavior can follow. These goals can be manifested as an optimization problem, where an agent tries to maximize or minimize a predefined property, or as a set of criteria they use to make a behavioral decision.
- Agents often have the ability to learn from their decisions. By measuring the distance from their goal or using a quantitative analysis of the outcome of their decisions in a certain state, agents can adapt their decision-making strategy to ensure a higher "payout" the next time they are in a similar situation.
- Agents have attributes that can change based on their interactions with other agents and the environment. These attributes are often the focus of study when developing agent-based models to understand how a system works.

Agent-based modeling is a powerful paradigm for understanding decision-making processes within systems. By analyzing state-action pairs of individual agents, we can get a better understanding of what situations cause real-world entities to act in certain ways.

For example, suppose we have an agent-based model studying human population movement in an area, with each agent representing an individual human. Each agent would likely have logic determining whether they stay in a location, or leave the location in which case there would be more logic to determine another area to move to. This logic would be dependent on a variety of factors, such as population and resources in various locations. With a detailed enough simulation, we can answer questions about human behavior such as what factors cause people to leave an area, and what areas people prefer when moving to a new area.

As seen in the example, agent-based simulations can provide an intuitive understanding of agent behaviors. The discrete nature of agent-based simulations makes it easier to understand simulations with hundreds or even thousands of moving parts. By singling out a single agent and measuring its properties, we can get a more intuitive understanding of each individual component without needing to understand every relationship in the system as a whole [7].

Agent-based simulations do have their own shortcomings, however. Firstly, agent-based simulations often require a lot more computational power than system dynamics simulations [7]. This is due to the first property of agents, where each agent acts autonomously, independent of other agents. Agent-based simulations also do a poor job of representing macro-based concepts. For example, the movement of water between water sources can be represented in an agent-based model where each water source is an individual agent, however, the behaviors these agents exhibit have effectively no real-world intuition behind them. Due to these reasons, agent-based simulations are not a universally effective paradigm for creating simulations.

4.2.3 Hybrid Model

Both paradigms for complex system simulations have their own advantages and disadvantages over each other. Systems dynamics simulations provide an intuitive understanding of how the simulation environment evolves on a macro scale, while agent-based simulations provide an intuitive understanding of individual decision-making processes on a micro-scale. On the flip side, a system dynamics paradigm does not do a good job of modeling several independent yet interconnected components behaving in their own way, while an agent-based paradigm provides little to no intuition on how changes in the environment affect the system as a whole.

WIT wants to provide its developers and subsequent users with an intuitive approach to model and understand complex systems. As a result, we wish to combine the two simulation paradigms to create a hybrid paradigm that makes modeling a complex system as intuitive as possible, while making it easy to understand the system at both a micro and macro level.

Hybrid models using aspects from both agent-based modeling and system dynamics modeling have been studied in the past. One type of hybrid model commonly used splits the system into individual entities. These individual entities have internal states that are updated using a system dynamics model. The interactions between entities of the hybrid system are modeled using an agent-based model. Other approaches use a similar method, with one key difference. Instead of the state of entities being modified using a system dynamics model, the state of the system is modified using a system dynamics model, while the interactions between entities and the state of the entities are modeled using an agent-based approach [23].

Of these two common paradigms of hybrid models, the latter fits our use case the most. Since Unity provides various functionality to modify a GameObjects state, we felt that the state update of individual entities did not need to be modeled using a system dynamics approach. Instead, we focused on modeling the state of the system using a system dynamics approach. The following paradigm is the result of adapting this hybrid method to the WIT simulation layer:

- Individual components of the system that display behaviors that run on programmatic logic will be represented as agents. These agents will have physical components with positions and rotations tied to them, as they need a medium to interact with other agents.

- Components of the system that are purely meant for resource storage purposes will have a representation under the system dynamics paradigm. These components will not have individual behaviors but will be able to transfer stored resources between each other through user-defined functions.

- In order for the agent-based aspect of the system to interact with the system dynamics-based aspect of the system, each component within the system dynamics side of the system may have a physical representation with positional information as well as an interface to modify the component within the system dynamics side of the simulation. This way, the physical agents are able to directly interact with the system dynamics side of the simulation using the provided interface.

The hybrid model proposed here was designed with the intent of splitting up complex systems in a way that is easy to understand and model. This model will leverage the advantages of both system dynamics and agent-based modeling paradigms. By splitting the model into two separate sections, users can use their intuitive understanding of complex systems at both a micro-level and macro-level in order to develop a simulation of the complex system for students to use.

The following is how we envision what the development process of a complex system will look like in our hybrid model. We include an example of each step for the development of a foresting-based complex system:

1. The developer identifies which components in their system can be modeled as part of the environment. These are components of the system that do not explicitly exhibit "behaviors", but rather contain resources that can move/be converted between each other. These components of the system, as well as the movement of resources, are then modeled using the system dynamics interface that we provide. Example: within a foresting-based complex system, some components that fall in this category could include the number of trees in an area, the amount of water stored as clouds, rainfall, and the dispersal of tree seeds between two separate locations.

2. The developer then identifies which components in their system can be modeled as a collection of behaviours. These will likely be the complementary set of system components from the components identified in Step 1. These components will be modeled using the agent-based modeling interface that we provide. Example: lumberjacks that "decide" when to cut down trees in a forest, animals that run away from lumberjacks out of fear.

3. The components of the system dynamics side of the simulation that agents directly interact with must be identified. Physical representations of these components are then created, providing a way for agents to interact with the environment. Example: A physical forest is created that will eventually allow lumberjack agents to cut down trees when they are within physical distance.

4. Agent behavior is then defined by the developer using the provided agent-based simulation interface. This interface will allow a developer to specify how agents interact with each other and the physical representations of the system dynamics side of the simulation. Example: Lumberjack code adds logic that decreases the number of trees when they are within a specified distance of the forest.

We believe this process to be the most intuitive and straight forward way of breaking down and understanding a system in terms of its individual components. This hybrid model, however, does come with its downsides. We place certain constraints on the system simulation that allows this model to work. For example, all agents must have a physical representation in order to interact with each other. In some complex systems, agents could be abstract entities that modify the state of the world. An example of this is an agent that increases the price of each log based on the number of trees in the forest. This agent would intuitively not need a physical representation, as it is not using any physical information about the simulation in order to perform its behaviors, but would still require one with our method. While creating this agent would still be doable, since Unity allows for physical entities to become physically "invisible", it would still require some overhead and extra computation power. Another downside of this hybrid approach is the extra steps required to connect the agent-based simulation with the system dynamics simulation. Agents are not able to directly modify the system dynamics simulation, and must instead interact with a physical representation of the component they would like to interact with. Overall, however, we believe that these downsides to the hybrid model are minor disadvantages compared to the intuitive value that the model provides developers.

4.3 Interface

Once a model for representing complex systems was decided, the next step of the process was defining an interface that developers could use to create their simulations. As stated before, the goal of the interface was to provide developers a simple and intuitive way to break down and develop complex system simulations. In this section I detail the various methods provided to the target audience of this project, and provide discussion on the importance of these features. Implementation details and design decision will be discussed in a future section.

4.3.1 System Dynamics Interface

We intend for developers to create simulations in the steps detailed in Section 4.2.3. The first step in this process is to define the environment/system dynamics aspect of the simulation. The following functions are provided to developers to create the system dynamics side of their complex system:

```
* Creates a new collector (stock) with specified parameters
* @param name The name of the collector used for referencing
* @param startAmount Amount of resource in collector on simulation start
* @param minCapacity Minimum amount allowed to be in collector at any given
                      point in the simulation. Default value 0
* @param maxCapacity Maximum amount allowed to be in collector at any given
                      point in the simulation. Default value ∞
AddCollector(name, startAmount, minCapacity = 0, maxCapacity = ∞)

* Creates a flow between two collectors c1 and c2
* @param c1 Name of the collector from which resource is removed
* @param c2 Name of the collector to which resource is added
* @param flowFunc Function that returns 2 values: the rate to reduce c1 and the
                   rate to increase c2 on any given tick of the simulation. This can
                   be a function of any number of variables.
```

```
CreateFlow(c1, c2, flowFunc)

* Gets the amount of resources in a specified collector
* @param name A collector's name
* @return float The amount of resources currently stored in the collector
GetCollectorAmount(name)
```

These three functions are the fundamental required to create the system dynamics aspect of the simulation. This will allow developers to create a rudimentary stocks and flows model, as well as read data about the model at any given point in time. This interface, however, is only part of the interface provided to the users. While the three functions above can be used to implement any stocks-and-flows model, some aspects of system dynamics simulations are difficult to implement with these three functions alone with no prior understanding of the implementations of the functions. For example, suppose a developer wanted to add an infinite source into their simulation (e.g. an ocean in a water simulation). In theory, the developer could call the *AddCollector* function with a starting value of ∞. This, however, may be unintuitive for both the developer of the simulation, as well as future developers within the WIT team. As a result, to solve this problem and other similar issues, we provide further abstractions to the system dynamics portion of the simulation engine:

```
* Adds a new collector that contains an infinite amount of a resource. Adding or
    removing any amount of resource from this collector will not change its value.
AddInfiniteResource(name)

* Adds a new collector that contains an infinite amount of a resource. Resources
    can only be added or removed with a certain probability. This can be used for
    non-constant elements in a simulation (e.g. rain).
* @param name Name of the random resource
* @param timeInterval The interval in which the state of the random resource does
```

```
                    not change (e.g. length of time it rains)
* @param probability The probability that the collector is open to adding or
                    removing resources during a time interval
AddInfiniteRandomResource(name, timeInterval, probability)

* Gets a string representation of the current state of the simulation. This
  representation includes all finite collectors and the amounts of resources stored
  within them.
* @return string Representation of simulation state.
GetSimulationString()

* Gets the resource amounts in a given collector from a starting point in time
* @param name Name of the collector for which we want the history
* @param starttime The starting point of the history (in seconds)
* @param endtime The ending point of the history (in seconds)
* @return List[float] A list of the amount stored in the specified collector between
  the specified starting time, and the current time. The history of the collector is
  polled once every second.
GetHistory(name, starttime = 0, endtime = now)
```

This list of functions was determined by the WIT team to be the fundamental abstractions when creating a system dynamics-based simulation. It provides developers with a set of intuitive functions they can use to define their system, while still providing them with enough freedom to create relationships between components in their system in any way they desire. The list is not exhaustive, and in the future, there will likely be more functions to add to the system dynamics interface. For now, however, we believe that these functions are sufficient for a robust system dynamics simulation within our hybrid model of complex systems.

4.3.2 Agent Based Interface

As explained in section 4.2.3, the second step that a developer goes through when creating an agent-based simulation is defining what agents are part of the simulation. Through the interface we provide, developers must be able to define agent behaviors, create physical entities representing the agents, and connect those agents to the system dynamics side of the simulation.

Since agents have a high degree of freedom in terms of behaviors they can exhibit, we leave the behavior definitions up to the developer. As Unity itself is an agent-based game engine, we chose to give developers access to the full range of tools offered by Unity to define their agents. In order to assist new developers and make the process easier, we also provided developers with common abstractions used in agent-based simulations. Additionally, we added some requirements for running agent behaviors to ensure smooth integration into the system dynamics side of the simulation. The following is the interface we decided upon for building agents:

```
* This function is where agent behavior will be defined by the developer. It is called
on every tick of the simulation
* @param deltaTime The amount of time passed since Tick was called last
Tick(deltaTime)
```

This function is the only function required for developers to implement in agents. Other functions in the interface provide developers tools to make defining agents' behavior easier:

```
* Adds a collision handler between the current agent and agents of another type
* @param agentType The type of agent that collision behavior will be defined with
* @param action The function is called when the two agents collide.
OnCollisionWithAgent(agentType, action)

* Function called when the simulation starts. This function is called only on agents
that exist within the scene before the simulation is started. The function will be filled
in by the developer
```

```
OnSimulationStart()

* Function called when an agent is spawned into the simulation. This function will
be filled in by the developer
OnSpawn()

* Function called when an agent is despawned from the simulation. This function
will be filled in by the developer
OnDespawn()

* Moves the agent to a specified position in the simulation
* @param newPos The new position to move the agent to
SetPosition(newPos)

* Rotates the agent in a way specified by its euler angles.
* @param newRot The new rotation to rotate the agent to.
SetRotation(newPos)
```

This set of functions can be used/implemented by the developer in order to make the agent development process easier. There were additional discussions on other functions that could be added to the agent-based development interface, however, we came to the conclusion that many of these functions would simply be Unity one-liners. For example, one function that came up in discussion was changing the color of an agent. While it was possible to provide a function ChangeColor(newColor) for the developer to use, this would provide developers no additional benefit compared to the functions that Unity natively provides. As the WIT project expands and developers give feedback on functions that would be helpful for their development, this interface will also likely expand. For now, however, we believe that this interface will provide developers with the abstractions necessary to simplify designing their agents while providing enough freedom that they can define agent behavior however they like.

4.3.3 Hybrid Interface

The final step of the process that we need to provide developers with an interface for is connecting the agent-based aspect of the simulation to the system dynamics aspect of the simulation. There are two types of interactions that can happen between the agent-based layer and the system dynamics layer. The first is that an agent can directly modify the system dynamics. This is a fairly straightforward concept, with agents defining their interactions with the system in their behaviors. The second type of interaction is when properties of the system dynamics aspect of the simulation modify the agents. This would mean that some aspect of the system needs to be able to modify the agents as well. This is where we can leverage the definition of agents made in section 4.2.2. Since agents are independent entities, any action on a group of agents can be modeled as distributing the action amongst the individuals of the group. This means that any interaction that the system dynamics simulation has on the agent-based simulation can be modeled within the individual agents themselves.

This analysis of interactions means that any interaction in the system can be done through the use of agents. As a result, the interface we provide is meant to be called within the code of agents:

```
* Modifies the amount of a resource stored in a collector by a certain amount one
time
* @param name The name of the collector that will be modified
* @param deltaAmount The amount by which the collector will be modified (can be
                          negative)
ModifyCollectorAmount(name, deltaAmount)


* Modifies the flow rate between two collectors
* @param c1_name The name of the outflow collector
* @param c2_name The name of the inflow collector
* @param newFunc Function that returns two values which determine the rate that
                     the collectors change
```

```
ModifyFlowRate(c1_name, c2_name, newFunc)
```

* Spawns a new agent in the simulation with the given starting position and rotation.
* @param agentType The type of agent that will be spawned
* @param position The world position where the agent will be spawned
* @param rotation The orientation that the agent will be spawned in

```
SpawnAgent(agentType, position, rotation)
```

* Permanently deletes an agent from the simulation. This action cannot be undone.
* @param agent a reference to the agent that will be deleted

```
DeleteAgent(agent)
```

* Gets a list of references to all agents in the simulation of a given type
* @param agentType The type of the agent that will be retrieved
* @return List[Agent] List of all agents of the specified type

```
GetAgentsOfType(agentType)
```

* Gets the distance in world coordinates between two agents
* @param agent1, agent2 References to the agents whose distance needs to be measured
* @return float Distance between the two agents

```
GetDistanceBetweenAgents(agent1, agent2)
```

* Gets a list of references of all agents within a specified distance of a certain location
* @param position The location from which distance will be measured
* @param distance The radius of which agents will be searched for
* @return List[Agent] List of all agents within the specified distance of the starting location

```
GetAgentsWithinDistance(position, distance)
```

As with the other interfaces defined for the simulation engine, we believe that this set of functions provides developers with sufficient functionality for interfacing the agent-based side of the simulations with the system-dynamics-based side. Of these functions, however, not all are necessary to provide to the user. The last two functions, GetDistanceBetweenAgents and GetAgentsWithinDistance can be manually implemented by the developer using the other functions provided as well as native Unity functions. We decided to provide developers with these two functions in order to simplify their development process. It is highly likely that in the future of WIT, more functions similar to these two functions, which simplify the development of simulations but do not necessarily add functionality for users, will be added as patterns in the development process are explored.

4.4 Implementation

With the interface for simulation development now defined, the next goal was to create the interface within Unity. Two main factors affected design decisions within the implementation of the interface:

1. Ease of expansion. The interface defined in the above section contains a bare backbone of tools required for a developer to create a simulation within the WIT project. As the WIT project continues and more simulations are developed using the interface, certain patterns will likely form. In order to abstract these patterns away from the developer, new functions will be added to the interface to ensure that developers will not be required to write code with some basic functionality that we could provide. This will make it easier on the developer as a) the developer has less chances to make errors within their own implementations, and b) developers will not need to solve common problems faced by other simulation developers.

2. Ease of understanding. Since WIT is a multi-year project that will inevitably have a conclusion, the code will likely be expanded by both future members of the WIT team as well as external developers interested in adding core func-

tionality within the simulation engine. Since many others will need to read and understand the code before adding their own features to the simulation engine, it is important to implement the interfaces in a way that is easy for someone to understand. This includes properly modularizing components in the code in an easily understandable way, and stripping any major complexity from the programming logic by leveraging Unity and C#'s built in functions.

These two factors, as well as the base goal of creating a functional interface for developers as defined above resulted in the simulation engine being designed as discussed below. Since the interfaces for the system dynamics aspect and agent-based aspect of the simulation were disjoint, we decided to modularize the implementation of the interfaces in a similar way.

4.4.1 System Dynamics Implementation

The first module that was implemented was the system dynamics portion of the implementation. This was chosen as the first part of the engine to implement, since it operates independently of Unity's subsystems. As described in the interface for the system dynamics component of the simulation in section 4.3.1, this side of the simulation does not require the use of Unity's GameObjects, and thus much of Unity's functionality is not required.

There were two main abstractions within the system dynamics simulation that needed to be modeled: stocks and flows. Stocks are an abstraction that simply holds an amount of a certain resource, while flows are more complex abstractions which represents how much of a resource moves between stocks at any given time. Early on in the development stage, the goal was to modularize the system as much as possible. This would mean having separate classes for stocks and flows, and another class to link these two concepts together to make a full fledged simulation. The thought process behind this was that more modularization within the code would fall in line with the goals of ease of expansion and ease of understanding. With the former, any functionality that would need to be added would have to be categorized into three types:

1. Behavior of a stock

2. Behavior of a flow

3. Combined behavior of stocks and flows

Once the type of the added functionality is determined, a developer could simply extend the system dynamics simulation by adding the function directly into the class determined by the type of functionality. With the latter, the system dynamics code could be understood by first analyzing the code from the basic components (stocks and flows), and using the understanding of those components to understand the interactions defined by linking code between the two classes. Early attempts at implementations of the system dynamics simulation interface used this concept of high modularization between basic components.

This model of modularizing the system was quickly found to not be a feasible implementation model. This is because of two main issues that arose during the implementation process. The first is that the base components of the modularization were not necessarily expandable. Stocks and flows are very basic components that contain very little functionality within themselves. In the early implementation, stocks were objects that held a single state variable representing the amount resource within the stock, and flows were objects that held a function to determine how much of a stock needed to be transferred to another stock. Beyond the data, the only additional functionality that these objects needed were functions to change the stored. Since the functionality of stocks and flows are limited on their own and cannot be meaningfully expanded, the need to cater towards the goal of ease of expansion was no longer an issue. The second main issue is that object-oriented programming (OOP) paradigms do not follow the same model as the stocks and flows paradigm. Objects in OOP are meant to store data, and operations on that data are defined in methods within the object. While an object representing stocks would follow this model, with the data being the amount of resource within the stock, an object representing flows would not fit this model, as flows represent functions. Because of this difference between common object-oriented principles and the flow representation within a system dynamics simulation, keeping this initial implementation would undermine the goal of ease of understanding of the implementation.

Modifications were made to the initial attempts of implementation in order to resolve the issues that were detracting from our goals. Stocks and flows did not need to be modularized, since the base components had very little functionality on their own, and thus all stocks, flows, and the interactions between them were processed entirely through one file called the SystemDynamicsSimulator. This implementation, however, quickly grew to be complex. Several data structures were required to store all stock variables, all flow functions, map stock variables to flow functions, modify stock amounts based on the flow functions at any given time, etc. Additionally, since all data and computations were being stored in one file, it became difficult for us to understand our own code, meaning future developers would likely also trouble understanding the code. As a result, this method of using one file for the entire system dynamics aspect of the simulation was also discarded.

From these implementations, it was clear to us that running the full simulation from one file was difficult because of the complexity required to support the simulation, and that modularizing flows would be unintuitive to future readers of the code, the only other option was that stocks would be represented as their own separate objects. Even with this code structure determined, there were now several different ways of representing flows within the simulation. After some experimentation, the following code structure was decided upon. Note that this code structure is a high level pseudocode representation for understanding the relationships between stocks, flows, and the code that runs the simulation, and does not represent the depth of implementation of the actual code.

```
Class Stock {
    float amount
    List<Pair<Func, Stock>> outflows
    func Flow(currentTime):
        for flowFunc, destStock in outflows:
            amount -= flowFunc(currentTime).outAmount
            destStock.amount += flowFunc(currentTime).inAmount
}


Class SystemDynamicsSimulator {
    List<Stock> stocks
    func Tick(currentTime):     * This is called on every simulation tick
        for stock in stocks:
            stock.Flow()
}
```

This method of setting the flow functions to be a property of the source stocks was a deliberate decision made because of how the flow function interacts with stocks. One initial concept was to have the flow functions be stored within the simulator class itself. This, however, required multiple data structures to be used in order to implement the rest of the interface. For example, to implement just the Tick function alone, a designation for which flows affect which stocks, as well as the relationship graph between stocks, would have to be stored within the simulator class. As a result, it was decided that flow functions would be represented entirely within the stock class. There was an additional decision on which stock a flow function should be connected to - the source stock or the destination stock. In the end, it was decided to be more intuitive that a flow is represented on the source stock since a stock providing resources to another stock seemed more natural that a stock taking resources from another stock. This decision was made fairly arbitrarily, and if it is decided by future developers in WIT that flow functions are more intuitive to understand as part of the destination stock, this can be changed easily by swapping signs within the Flow function.

While this problem of determining the structure of the system dynamics code was an important design decision that would define how the system dynamics aspect of the simulation engine ran, several other design decisions were made to ensure the goals of ease of understanding and ease of expansion were reached.

When designing a simulation with the WIT team, several possibilities of simulations were discussed which required the expansion of the types of stocks provided. In the discussions of water systems, which modeled how water would move around several sources, features such as an ocean that provides an infinite supply of water, or rain which does not provide a steady flow but instead only flows in certain conditions, could not be modeled with the implementation discussed thus far. As a result, the implementation would need to be expanded to allow for these interaction types to exist within the system.

One common aspect of these features is that they all perform the same "flow" action as a basic stock. Oceans should still be able to give and receive water, even if there is no "amount" as required in the stock class, and rain should be able to switch on or off which the stock class does not implement. Since all of these structures need to perform the same core functionality, before creating brand new structures to support this behavior, this core functionality would need to be factored in to prevent code duplication.

C# supports the creation of abstract classes. Abstract classes are special classes within object-oriented programs that cannot be instantiated. Instead, abstract classes must be inherited by other classes. These other classes then receive functionality and data specified within the abstract class. What makes abstract classes different from standard classes is that abstract classes may include abstract methods. These are empty methods that must be implemented by the child class in order for the code to compile. Since we need all of our new structures to support the same functionality, but use their own implementations, we can leverage the use of abstract classes to solve this problem:

```
abstract class ResourceContainer {
    List<Pair<Func, ResourceContainer>> outflows
    abstract func Flow(currentTime)
}
```

Now that we have our abstract class defined, we can adapt our original stock class and system dynamics simulation class to use this abstract class:

```
Class Stock : ResourceContainer {
    float amount
    func Flow(currentTime):    * This is overridden from ResourceContainer
        for flowFunc, destStock in outflows:
            amount -= flowFunc(currentTime).outAmount
            destStock.amount += flowFunc(currentTime).inAmount
}

Class SystemDynamicsSimulator {
    List<ResourceContainer> resourceContainers
    func Tick(currentTime):     * This is called on every simulation tick
        for resourceContainer in resourceContainers:
            resourceContainer.Flow()
}
```

With this modification, the system dynamics simulator will be able to handle any structure that inherits from the ResourceContainer class. As a result, we can imagine defining other ResourceContainer types with custom flow behaviors. For example, suppose we wanted to model an infinite source that can give resources to other stocks. We could model this as:

```
Class InfiniteSource : ResourceContainer {
    func Flow(currentTime):    * This is overridden from ResourceContainer
        for flowFunc, dest in outflows:
            dest.amount += flowFunc(currentTime).inAmount
}
```

By using the functionality of abstract classes, we are able to add new structures to our system dynamics simulation. This works towards both goals of implementation design. So far, all structure types discussed by the WIT team have already been implemented using the abstract class and added to the system dynamics simulation interface. However, since flow functionality can be open-ended, it is likely that in the future, other generic structures for the system dynamics simulation will need to be created. The abstract class method makes this expansion of system dynamics structures easier for future developers by providing support for the use of generic flowing structures within the simulation.

The final major design decision that was made when implementing the system dynamics simulation was related to networked data. As discussed in Section 3, the simulation code would be running on a dedicated server, with data being sent to the headsets. The most important decisions for sharing data would be to determine what data needed to be sent to the headsets, as well as how that data would be structured.

The two main types of data that were being handled by the system dynamics simulation were data related to stocks and data related to flows. Data related to stocks include information such as how much of a resource is currently stored in the stock, or which stocks are connected to each other. Data related to flows includes information such as the current flow rate. To minimize network traffic, it was decided that only data which needed to be accessed by the headsets would be constantly updated over the network. This data was determined to only be data about the amounts of resources within any given stock. By having this data be networked, headsets can constantly keep track of stock amounts to perform any local computations necessary, such as modifying local visuals or sending Remote Procedure Calls (RPCs) to the server based on the amount of a stock.

This does not mean, however, that headsets are only limited to using stock amounts. If a headset needs other data about the system dynamics simulation, it may send RPC requests to the server, which will respond with the data in question. We expect that data not related to resource amounts will rarely be required by the headsets, so these RPC requests will not be called often. This way, headsets will have access to any data required, and network traffic will not be overloaded by syncing an entire simulation's worth of data to the headsets.

Photon Fusion provides developers with a method to synchronize certain data types across the network. Since resource amounts within the system dynamics simulation were represented as floats, the data structure used for synchronizing resource amounts across the network needed to either be floats or collections of floats. One important feature of networking data in Fusion is that the networked data needs to be attached to some networked object. Otherwise, if the data is instead attached to a local object, other devices across the network would not be able to access the data. If we were to simply network the amount of resources within each stock class, we would have to convert all stock objects into network objects. This would be unnecessary network overhead, and could cause synchronization lag over the network. As a result, we instead use a single networked object to store all data within a collection of networked floats and update this networked object on the server. This means that only one object would need to be spawned on the server, and all headsets would subsequently be able to access the synchronized resource amounts.

While the design decisions discussed in this section were the main decisions made when implementing the system dynamics simulation, several other small implementation decisions were made that are not significant to the design of the engine, and as such will not be discussed in this paper.

4.4.2 Agent Based Implementation

The next step in implementation was to create a framework for defining the agent-based simulation. The goals for the agent-based simulation built on the goals defined at the start of the section; on top of prioritizing ease of use and ease of expansion, we needed to ensure that the agent-based simulation interfaced nicely with the system dynamics simulation. Additionally, each agent in the simulation would end up having a virtual Unity GameObject attached, so the implementation of agents needed to support Unity features as well. The constraints that both Unity and the system dynamics implementation set on the agent-based simulation decreased the possibility space of implementations while making it more difficult to find a suitable implementation for the original two goals.

The initial attempt at the agent-based implementation followed a similar structure to the system dynamics implementation. The goal of the initial attempt was to create an implementation that could run completely independently from Unity's structure. This way, developers could define the simulation purely in C#, and would only need to interact with Unity to tie GameObjects to simulation components and affect visuals within the headsets. This would allow developers with little to no Unity experience to use the provided interface to create and run simulations from end to end, aligning with WIT's goal of creating frameworks that would simplify a developer's experience in creating a simulation as much as possible.

One of the advantages of using Unity for an agent-based simulation is that Unity is an agent-based gaming framework. GameObjects within Unity are independent entities that act autonomously based on predefined behaviors, similar to the definition of agents presented in section 4.2.2. Continuing this analogy further, creating a generic GameObject with some behaviors can be thought of as creating a generic agent with those same behaviors. Therefore, if we model a generic agent script similar to how the generic GameObject script works, then we can create agents as a child of the generic agent script similar to how GameObjects in Unity are children of a generic GameObject. Once the agent is instantiated within the simulation, we can transfer its behavior to an analogous GameObject to run in Unity.

Similar to the system dynamics model, where we had a generic ResourceContainer abstract class which could be implemented to make customized system dynamics structures, in the agent-based implementation, we created an abstract Agent class. Unlike the ResourceContainer, however, this Agent class would be used directly by a simulation developer to create customized behaviors for their own agents. This abstract class would contain both abstract methods that would need to be implemented by the developer, as well as regular methods hidden away from the developer in order to interface the agent with the system dynamics simulation.

The following was a preliminary structure for the Agent script. As before, this is a pseudocode implementation and does not expose the details of the actual implementation:

```
abstract class Agent {
    Vec3 position
    Quaternion Rotation
    GameObject linkedObject
    * This function performs any behavior defined in an agent script on object o
    func LinkToGameObject(GameObject o):
        linkedObject = o

    * This function is called on every tick of the simulation and will sync the gameobject with the simulation agent
    func TickAgent():
        Tick()
        linkedObject.transform.position = position
        linkedObject.transform.rotation = rotation
    * This function must be defined by the developer to define bahvior on the agent every tick of the simulation
    abstract func Tick()
}
```

The code displayed above is a very rudimentary version of the first attempt at implementing the agent framework. In order to create their own custom agent, a developer would simply inherit from this agent class and implement the Tick() function to display their desired behavior. An example of this is as follows:

```
class Frog : Agent {
    Tick():
        position += (1,0,0)
}
```

At a basic level, this method of implementing the Agent worked. GameObjects were abiding by the behaviors set by a developer in a derived Agent class. As more features were added to improve upon the interface, however, two issues became apparent. Firstly, the set of features required to model an Agent as a GameObject was enormous. In the example above, the only properties that are being synced between Agents and their linked GameObjects are position and rotation. However, GameObjects can have hundreds of properties that can be programmatically changed, each of which can have hundreds of properties themselves. As a result, creating an Agent script that synchronizes all of this data to the GameObject is infeasible and has a high chance of causing bugs and glitches within the simulation. Secondly, and more importantly, since Agents in this implementation are completely independent of Unity, they do not have any object representations attached to themselves. This means that any property related to the physical property of the object, such as color, cannot be modeled. This issue was initially discovered when trying to implement collision logic within the Agent script. Since these independent Agents do not have physical entity representations, collisions cannot be detected, and as a result, the interface defined in section 4.3.2 cannot be created.

The two aforementioned issues make the initial approach for implementing the agent-based simulation nearly impossible. In fact, since physical callback functions are only available on scripts attached to GameObjects in Unity, the freedom to design the agent-based simulation framework is very restricted. Any behavior-defining scripts created by the developer must be placed on a GameObject. While this is restrictive in a design sense, this opens up the possibility to use the entire Unity library for both development of the interface and the development of the simulation itself.

The next iteration of the agent-based simulation was the final implementation that was decided on. As stated previously, because of the restriction on how agents within the simulation needed to be attached to GameObjects, there was only one feasible way to implement agents. Similar to the first attempt, the Agent class is an abstract class we provide to developers to inherit and create customized agents from. This time, however, these agent scripts will not be independent of Unity, but will be instead attached to Prefabs created by the developer to act as a physical representation of the agent. The following is the final basic structure of the agent script of the simulation engine

```
abstract class Agent : NetworkBehavior{ * Necessary for spawning agent
    * This function must be defined by the developer with the desired agent behavior, and will be called on every tick of the simulation
    abstract func Tick()
}
```

While many design decisions were made when creating the agent-based implementation of the simulation engine, most of the decisions were minor, as they corresponded to design decisions of singular functions from the defined interface. The decision of keeping the agent script tied to a GameObject or keeping it independent was the only major decision made when implementing the agent-based simulation.

Chapter 5

Results

As the simulation layer of WIT was designed as a framework for building simulations, there are few quantitative measures to determine the effectiveness of the solution. Instead, the effectiveness of the framework was defined by the ability to meet the goals of the project, as defined in section 1.4.

In order to test the WIT framework, a demo was created using all three layers of WIT. This demo was meant to be a proof-of-concept, testing all features implemented. An image from the demo can be seen in Figure 5-1. An analysis of the goals of the simulation layer of this project, as presented in section 1.4, was done in the context of the creation of this demo:

1. Ability to run a simulation to modify GameObjects. The simulation was able to modify the behavior of all GameObjects in the scene. Examples of evidence of this are frogs jumping around on tables, and butterflies flying around the room.

2. Ability for a developer to define customized simulations. The demo was created externally from the code used to develop the framework itself. This means that none of the components of the demo are hard-coded within the framework, but instead, the functions provided by the simulation framework were used to create the demo.

3. Ability for the simulation layer to interact with other layers. Through the

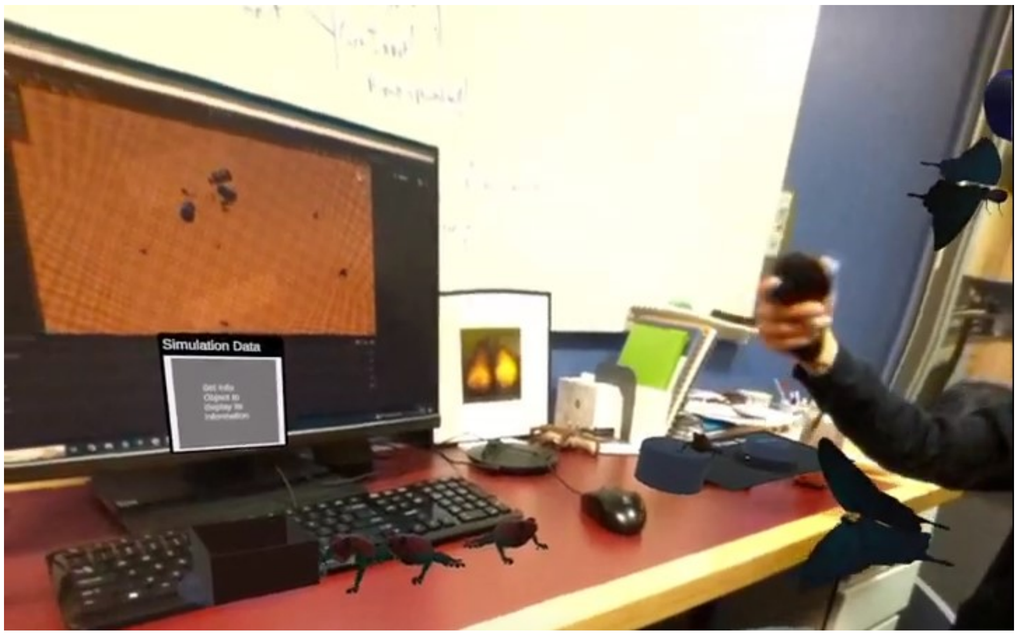

Figure 5-1: A still image from the demo created using the WIT framework.

demo, it was clear that the simulation was able to interact with both the view and interaction layer. The former was evidenced by the frogs' ability to jump and navigate around real-world objects without falling (recognize when they will jump off the real-world desk), which was a consequence of the simulation layer being able to read data from the view layer about the real world, and the latter was evidenced by the ability to change the rate of rain within the demo, or spawn butterflies within the simulation, which is the consequence of the simulation layer being able to identify when users are performing interactions in predefined ways.

4. As discussed in section 4.4, many of the design decisions made when creating the simulation layer were influenced by the need for the expansion of simulation components.

Overall, the current iteration of the simulation layer achieves the specifications required within the WIT project. Although the simulation layer is designed to be improved upon over the three-year span of the WIT project, it currently achieves its goals of being a robust platform for developers to build simulations on. This project will be an important stepping stone in the growth of immersive shared augmented reality experiences that will change the landscape of education in classrooms.

Bibliography

[1] Bolt vs pun. https://doc.photonengine.com/pun/current/reference/pun-vs-bolt.

[2] Feature overview. https://doc.photonengine.com/pun/current/getting-started/feature-overview.

[3] Fusion introduction. https://doc.photonengine.com/fusion/current/getting-started/fusion-intro.

[4] Fusion network simulation loop. https://doc.photonengine.com/fusion/current/manual/network-simulation-loop.

[5] Fusion vr host. https://doc.photonengine.com/fusion/current/technical-samples/fusion-vr-host.

[6] Passthrough api overview. https://developer.oculus.com/documentation/unity/unity-passthrough/. Accessed: 2022-12-03.

[7] Pros cons of agent-based modeling. https://educationalresearchtechniques.com/2020/08/07/pros-cons-of-agent-based-modeling/.

[8] Pun's structure. https://doc.photonengine.com/pun/current/getting-started/pun-intro.

[9] Spatial anchors overview. https://developer.oculus.com/documentation/unity/unity-spatial-anchors-overview/. Accessed: 2022-12-03.

[10] System dynamics. http://foresight-platform.eu/community/forlearn/how-to-do-foresight/methods/gaming-simulation-and-models/system-dynamics/.

[11] What-is-the-difference-between-shared-hosting-and-dedicated-server, Dec 2022.

[12] Larah Armstrong. Dedicated game server (dgs): Unity multiplayer networking, Feb 2023.

[13] Fikret Berkes. Fishermen and the "tragedy of the commons". *Environmental Conservation*, 12(3):199–206, 1985.

[14] U Blikstein, P. & Wilensky. An atom is known by the company it keeps: A constructionist learning environment for materials science using agent-based modeling. *International Journal of Computers for Mathematical Learning*, pages 81–119, 2009.

[15] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences*, 99(suppl_3):7280–7287, 2002.

[16] Giglioli IAC. Cipresso P. The past, present, and future of virtual and augmented reality research: A network and cluster analysis of the literature. *Front Psychol*, 2018.

[17] Roy D. Learning across realities: Virtual and augmented reality in education. page 59, 2021.

[18] Peppler K Danish J. Life in the hive: Supporting inquiry into complexity within the zone of proximal development. *Journal of Science Education and Technology*, pages 454–467, June 2011.

[19] Jay W Forrester. System dynamics and the lessons of 35 years. *A systems-based approach to policymaking*, pages 199–240, 1993.

[20] Anton Iancu. About netcode for gameobjects: Unity multiplayer networking, Apr 2023.

[21] Ripan Kumar Kundu, Akhlaqur Rahman, and Shuva Paul. A study on sensor system latency in vr motion sickness. *Journal of Sensor and Actuator Networks*, 10(3), 2021.

[22] Scott M Lewandowski. Frameworks for component-based client/server computing. *ACM Computing Surveys (CSUR)*, 30(1):3–27, 1998.

[23] Romina Martin and Maja Schlüter. Combining system dynamics and agent-based modeling to analyze social-ecological interactions—an example from modeling restoration of a shallow lake. *Frontiers in Environmental Science*, 3, 2015.

[24] Muaz Niazi and Amir Hussain. Agent-based computing from multi-agent systems to agent-based models: a visual survey. *Scientometrics*, 89(2):479–499, 2011.

[25] J M. Ottino. Complex systems. *AIChE Journal*, 49(2):292+, February 2003.

[26] Pereira A. Piovesan S., Passerino L. Virtual reality as a tool in the education. *CELDA*, 2012.

[27] Sara. Network topologies: Unity multiplayer networking, Mar 2023.

[28] Rüdiger Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings first international conference on peer-to-peer computing*, pages 101–102. IEEE, 2001.

[29] et al. Scogin S., Kruger C. Learning by experience in a standardized testing culture: Investigation of a middle school experiential learning program. *Journal of Experiential Education*, 40(1):39–57, 2017.

[30] John D Sterman. System dynamics modeling: tools for learning in a complex world. *California management review*, 43(4):8–25, 2001.

[31] Emma Anderson Susan A. Yoon. Teaching about complex systems is no simple matter: building effective professional development for computer-supported complex systems instruction. *Instructional Science*, pages 99–121, 2017.

[32] Hanel R. & Klimek P. Thurner S. Introduction to the theory of complex systems. *Oxford University Press*, 2018.

[33] TrendForce. Augmented reality (ar) and virtual reality (vr) headset shipments worldwide from 2019 to 2023 (in millions), 2022.

[34] Collela V. Participatory simulations: Building collaborative understanding through immersive dynamic modeling. June 1998.

[35] Stroup Vilensky. Learning through participatory simulations: Network-based design for systems learning in classrooms. 1999.

[36] Oculus VR. Mixed reality with passthrough. July 2021.

[37] Ellen Wang. Capturing worlds of play: A framework for educational multiplayer mixed reality simulations, Submitted: May 2023.

[38] Anna Weinstein. Designing student interactions to explore systems thinking in augmented reality, Submitted: May 2023.