Augmented Space Library 2:
A Network Infrastructure for Collaborative Cross Reality Applications

Gregory Smith

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science & Software Engineering

University of Washington

2020

Reading Committee:

Kelvin Sung, Chair

Yusuf Pisan

Dale Hamilton

Program Authorized to Offer Degree:

Computer and Software Systems

University of Washington

Abstract

Augmented Space Library 2:
A Network Infrastructure for Collaborative Cross Reality Applications

Gregory Smith

Chair of the Supervisory Committee:
Dr. Kelvin Sung
Computing and Software Systems

The Cross Reality Collaboration Sandbox (CRCS) research group is dedicated to studying issues related to supporting collaborations across geographical distances through different technological reality setups. These different setups can include immersive Virtual Reality (VR), Augmented Reality (AR), and traditional computers. The first version of Augmented Space Library (ASL), a thin network utility for supporting 3D object synchronization, was created to pursue this study. Over time, ASL evolved into an ad hoc collection of undocumented application programming interfaces, turning into a system that struggled to support investigative applications. The lack of an initial overarching architectural structure combined with over three years of non-stop modifications resulted in an unreliable and bloated system. It became evident that for CRCS to continue with its experimentation, a new library system, designed from the ground-up, was required.

The essential lessons learned from the first version of ASL are that the library must ensure straight forward network session establishment, be device-independent, and facilitate the rapid prototyping of simple ideas. These lessons guided the design effort and resulted in a new library, ASL 2.0, that is based on a hybrid of client-server and peer-to-peer architecture, is integrated with an existing device-agnostic front-end platform, and facilitates the synchronization and communication of distributed collaborative application states. With the design of ASL 2.0, application developers can focus on the fundamental issues of supporting collaboration and be free from the logistics of device-specific issues and the overhead of application state synchronization between remote users. The newly developed library went through two phases of testing to verify its completeness and usability. The initial round of testing explored the support for AR devices and resulted in a custom solution for resolving the different world-origin positions for collaborating AR devices. At the time of this thesis writing, all four projects in the second round of testing are entering their final phase with no major issues encountered. While there is still work to be done, the current ASL 2.0 release is in a healthy and stable state capable of supporting experimentations of remote collaborative applications with heterogeneous devices.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to personally thank my committee chair, Kelvin Sung, for his ever-lasting patience, perseverance, and guidance throughout this process. While I am glad it is finally finished, I will miss working with you. I would also like to thank my other committee members, Dale Hamilton and Yusuf Pisan, your questions and feedback have made this thesis stronger than it otherwise would have been. Next, I would like to thank all the individuals who helped test ASL and provided feedback on its various features. Without them, this thesis would be incomplete. Finally, I would like to thank my wife, Calli, for always supporting me, understanding the late nights, and for being my biggest fan.

Chapter 1. INTRODUCTION

While forms of virtual realities (VR)[1] and augmented realities (AR) have been around since the 1960s [1], [2], it wasn't until advancements in hardware and software technologies such as low-persistence displays, positional tracking, and a greater field of view, that both VR and AR became popular with the general public [3] – [5]. This popularity shows no signs of slowing down with some estimates, such as the one from FinancesOnline and the data they gathered from Statista, suggesting that the VR and AR industry will be worth $117 billion by 2024 [6].

Part of the reason for this predicted explosion of growth in the coming years is due to the vast amount of use cases and exciting opportunities VR and AR technologies offer to the world. While the use cases of VR and AR are arguably infinite, ranging from video games and education to health care and retail, there are still some common research areas that overlap all these vast fields. One of these most critical areas of research is understanding the necessary support and implications of remote collaborations where participants may or may not be equipped with the same or similar devices [7], [8]. An example of this type of interaction could be the use case of participant A using a VR device together with participant B who is working on a traditional computer.

It is the goal of the Cross Reality Collaboration Sandbox (CRCS) research group to analyze and understand how to facilitate the remote collaboration of users with such heterogeneous device configurations [7]. It was this research goal that led to the first version of the Augmented Space Library (ASL), a library designed to support the creation of remote collaboration applications [7].

In its initial phase, ASL was a library that evolved based on the needs of CRCS's on-going projects. These projects ranged from a dynamic AR obstacle course, with one user seeing the environment through an AR device and another user on a computer [9], to setting up a virtual classroom where students could experience the virtual learning environments together through a variety of VR and AR devices [10]. Through these projects, the essential functionalities of ASL emerged and are listed as follows:

- Facilitate user connections and data sharing
- Support heterogeneous devices
- Facilitate rapid prototyping

As ASL evolved to support these key functionalities with ever changing requirements from individual projects, it increasingly became a patch work of modules, with the ability to rapidly prototype and library documentation taking the hardest hits. Through this project, ASL has been redesigned and implemented from the ground up to meet all the discovered requirements.

The Augmented Space Library version two (henceforth referred to as just ASL) is designed to support the rapid prototyping of remote collaborative applications on heterogeneous devices. This library supports a simple lobby connection system, which hides network setup details and

[1] A list of all the acronyms and their meaning can be found in Appendix A

supports easy session specific connections. Additionally, the library was built based on a device agnostic application so that it could be device independent. It also maintains a minimal application state, thus helping facilitate a rapid development and test cycle. Finally, it is built with a vast array of tutorials and accompanying documentation to ensure its usability and longevity.

The testing of ASL went through two phases. The first focused on testing AR devices and their technology-specific challenges. Here, the need for conveniently defining and synchronizing a common world origin was learned and implemented. The second phase consists of four parallel projects. Two of these four projects are straightforward VR collaboration applications. The other two projects are AR-based projects that focus on supporting users in various technological settings, e.g., interacting on a computer or VR setup. The on-going projects from the second phase of testing verify the fundamental functionality of the redesigned ASL system.

This thesis will first present the motivation behind ASL's primary objectives and how multi-user VR and AR network infrastructure can be developed along with how the first version of the Augmented Space Library was implemented. Then, the next section will discuss the specifications and design of ASL. This will be followed by a section on how ASL's frontend is implemented and a section on how its backend is implemented. After that, section six will cover the results of this project, going over the simple tutorials that ASL implements and how these simple demonstrations can be combined to create several non-trivial applications that the developer of ASL did not implement himself. Finally, section seven concludes this thesis with a summary of ASL and what future work needs to be accomplished on ASL.

Chapter 2. MOTIVATION AND RELATED WORK

This chapter will discuss the motivation behind creating ASL, including the details on each requirement and why they are a requirement. The chapter then follows with the analysis of the strengths and weaknesses of different network architectures for supporting the goals of ASL. Finally, this chapter will review the previous version of ASL and describe the need for this new version.

2.1 MOTIVATION

Through the many projects that were built with the first version of ASL and reflecting on the challenges involved in working with an evolving library, it became clear that a networking library must accomplish four goals for CRCS to continue exploring collaboration across heterogeneous devices for the foreseeable future. These four goals, along with their reasoning and motivations, are discussed in the following subsections.

2.1.1 *Connection and Communication*

The defining feature of any networking library is its ability to connect users together and then present them a medium in which they can communicate. If ASL is to allow CRCS to further investigate remote collaboration on heterogeneous devices it needs to define a way for its users to find, connect, and share data with each other. It is therefore the primary goal of ASL to:

- Allow users to locate each other
- Allow users to connect to each other
- Allow users to share data

2.1.2 *Device Independence*

The second and almost as important goal of ASL is that it must be device independent. By being device independent, CRCS can continue its primary objective of exploring how collaboration can occur amongst heterogeneous devices. While ASL aims to be as device independent as possible, the following categories are the types of devices that are involved in CRCS's investigations:

- Traditional computers
- Mobile system AR devices – e.g., Android or iPhone devices
- Immersive VR devices – e.g., headset devices like the Valve Index or HTC Vive
- Immersive AR devices – e.g., headset devices like the HoloLens

Due to the rapidly evolving nature of VR and AR technologies, it is expected that devices in these categories will rapidly become obsolete and must be replaced. This observation and the

objective of exploring collaborations based on heterogeneous device configurations are the main reasons that ASL must be device independent.

2.1.3 *Rapid Prototyping*

The third goal of ASL is that it must allow its users to rapidly implement, test, and refine their applications. Here at the University of Washington Bothell, most projects will be built by students in the 10-week academic term. During these 10 weeks, students must learn ASL, create their collaborative application, test, refine, and then present their results.

To minimize the amount of time a user spends on ASL related work and to maximize the amount of time spent on their application, ASL must satisfy the following requirements:

- Quick installation – ASL must be straightforward to install for users, allowing them to quickly begin evaluating their ideas and applications
- Easy connections – ASL must provide a straightforward and easy method for users to connect their players together[2]
- Application state management – ASL must provide necessary application state communication to assist with the synchronization of applications
- Create a unified application state – ASL must support a user's ability to create a single unified application state for all players

By allowing users to rapidly prototype their applications, ASL becomes a library that supports idea exploration. Exploring different possibilities in different reality spaces is an important interest to CRCS and can only be supported if ASL helps users define a single unified application state for all distributed applications. By letting users create only one application state for all players, users can explore different ideas more quickly.

2.1.4 *Documentation*

The final goal of ASL is that it must be a well-documented software development kit (SDK). This is important for general usability and maintainability. To ensure users can begin development readily with, take full advantage of, and to help future developers improve ASL, the following must be documented with care:

- API functions
- Initial setup and configuration guide
- Simple tutorials focusing on individual functionality
- Elaborate tutorials highlighting how individual functionality can be combined

[2] Throughout this paper, "player" is used to refer to the individual who interacts with a CRCS application, and "user" is used to refer to the individual who develops the ASL-based applications.

2.2 RELATED WORK

There are three dominating architecture models for designing networking applications, or in this case, a networking library. These three approaches, their strengths and weaknesses, their typical use cases, and how they relate to ASL's requirement of rapid prototyping are discussed next. The following subsections only focus on the rapid prototyping goal of ASL as all three methods provide a way for users to connect and communicate data, are device independent, and none of them effect how well ASL will be documented.

2.2.1 *Client-Server Model*

While the client-server model has many different variations, in general it comprises of a server that clients can connect to and then communicate data to and through.

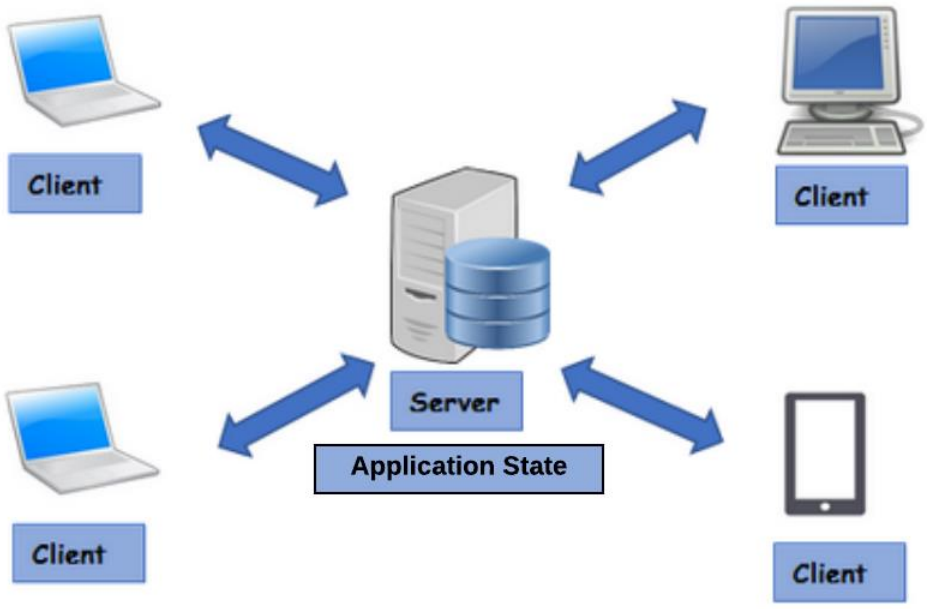


Figure 2.1. An example of a generic client-server model[3]

This general client-server model can be summarized in Figure 2.1 and has the following strengths and weaknesses [11], [12].

- Strengths:
  - Known point of connection
  - Application state management (centralized control)
  - Maintainable

[3] Figure has been modify from its original source [11]

- Weaknesses:
    - Single point of failure
    - Bottleneck issues
    - Server setup is different from client setup
    - Expensive to scale

Typically, the client-server model is deployed for web services such as hosting websites and email servers, as well as online multiplayer games [13] – [15]. By keeping the application state on the server, as shown in Figure 2.1, the client-server model facilitates the sharing of application state information amongst all clients while preventing clients from accessing client-specific data that is not their own [16], [17]. The client-server model also, once designed and implemented, can allow any number of clients to easily connect and is relatively easy to maintain [18] – [20]. However, the client-server model does have some drawbacks.

The first and most troublesome drawback is that the since the server holds the application state and is the communication platform for users, if it goes down, then the entire application will go down. Along these same lines, if too many users are online then the server may become bottlenecked, slowing down all users. Both problems can cause a substantial loss of revenue for a company, but both thankfully can be solved through scaling up, or adding more servers; unfortunately, this process is financially costly [18] – [20]. The last drawback of the client-server model is that one must essentially create two separate applications to implement an idea. The first application is the one that runs on the server and the second application is the one that runs on the client. This is necessary because the application state resides only on the server end, but the client must still be able to influence that state. Unfortunately, this two-prong system design means an increase in development time as well as system complexity.

While the client-server model has been shown to work with collaborative applications [21] – [23], it doesn't perfectly fit all of ASL's objectives. The maintainability, scalability, single point of failure, and bottleneck characteristics that help define the client-server model do not pertain to what CRCS and ASL are attempting to accomplish because of the short life span of CRCS applications and the relatively small number of users CRCS applications typically support (often times less than 10 players). However, the features that do pertain to CRCS and ASL are the known point of connection, application state management, and having to create two separate applications.

It is difficult to rapidly prototype applications when using the client-server model even though it quickly and easily allows for users to connect and communicate data between players due to its known point of connection. This is because as mentioned previously, since the client-server model stores the application state on the server, one effectively must create two applications: one for the server and one for the client. While ASL could provide helper methods to get both applications up and running faster, the user would still have to customize their server and client separately. Having to create essentially two separate applications will increase the amount of time users will have to spend on making their application multi-user and decrease the amount of time they can spend on exploring collaboration. So, while the client-server model typically benefits applications due to its application state management, in CRCS's case, it is a weakness.

2.2.2 *Peer-to-Peer Model*

Just like the client-server model, there are many variations of the peer-to-peer (P2P) model. However, in general, a P2P architecture is where every user is a client and a server [24].

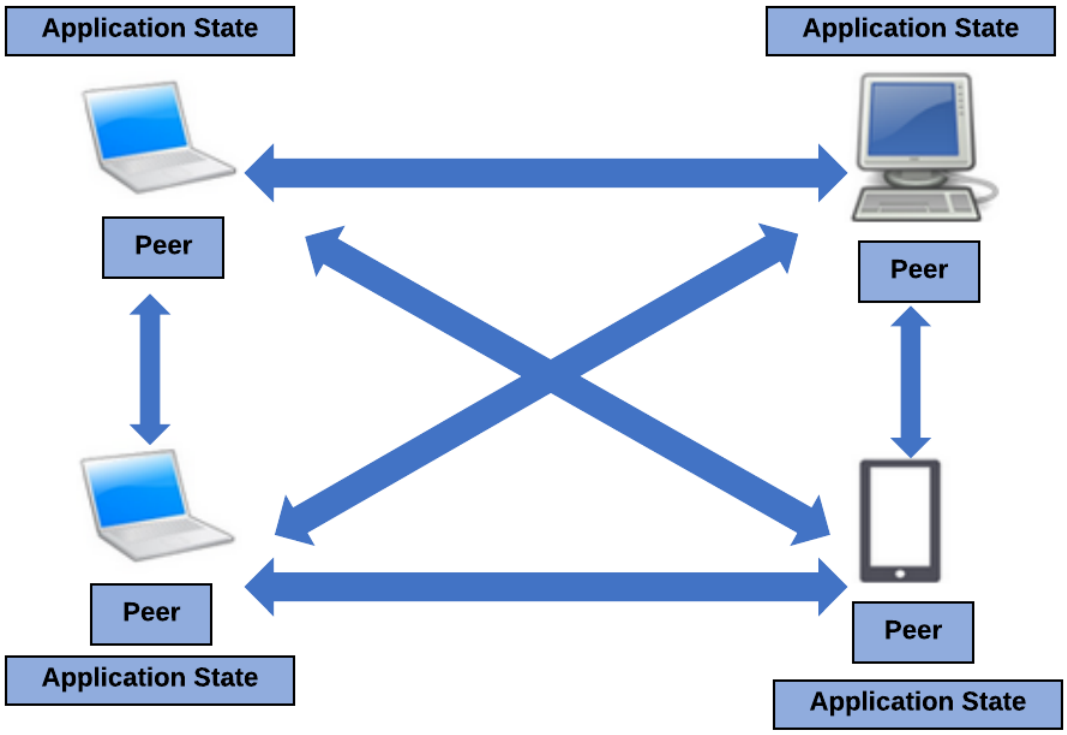


Figure 2.2. An example of a generic peer-to-peer model[4]

A typical view of the P2P model can be seen in Figure 2.2 and has the following strengths and weaknesses [26], [27].

- Strengths:
  - Inexpensive
  - Robust
  - Scalable
- Weaknesses:
  - Difficult to find peers
  - Little security
  - Difficult to synchronize application state

The P2P model is most known for its file sharing capabilities, with some of the most popular applications being Napster, Gnutella, and BitTorrent [27], [28]. These file sharing systems work well because of how the P2P model works. To begin with, the P2P model when compared to

[4] Figure has been modify from its original source [11], [25]

the client-server model is inexpensive to setup and maintain [27], [29]. This is because peers bring their own processing power with them, allowing the system to scale automatically with each peer that is added [27]. Also, since each peer acts as a client and a server, the system itself becomes quite robust. If any one peer drops out, another peer can just pick up the slack [25]. Therefore, unlike the client-server model, there is no single point of failure in a P2P system [27]. However, just like the client-server model, the P2P model is not flawless.

One of the significant weaknesses of the P2P model is its difficulty locating fellow peers and maintaining a list of available peers. This weakness often requires either a server to help initially connect players [27] or for players to modify their router settings so that network connections can occur [30]. There is also the issue of security. Since every peer can transfer data to other peers, if one peer has a virus, it could easily spread it to another peer – knowingly or unknowingly [29], [31]. The last major hurdle for P2P systems is the potential lack of any state management support. As seen in Figure 2.2, every peer maintains the state of the application themselves, separate from other peers. In file-sharing systems this means there are no file backups; if the peers holding file A leave the network, then no one will have access to file A until a peer with file A joins again. This is also the reason why online interactive graphical applications have steered away from using the P2P model for so long; with no application state management it is straightforward for a user to change the state of the application for all peers in a negative way [16], [17].

Though P2P architectures are most often related to file sharing systems, there are lots of examples of multimedia applications built with the P2P model and therefore creating ASL with one cannot be ruled out right away [28]. While being inexpensive and robust is a bonus, it is not a primary goal of ASL. Having little security and being scalable also have little weight as ASL will not be used to send files and as discussed in the client-server section, will have a small audience.

The most relevant factor for ASL is that in the P2P model, every peer maintains their own application state; thus, ASL users do not need to create two separate applications like they would for the client-server model. Therefore, even though not having a centralized state manager is often seen as a downside of the P2P model as it makes application state synchronization harder, it is actually of value to ASL. To ensure users have a consistent state amongst each other, ASL would need to provide some method to ensure all peers are properly synchronized such that true collaboration can occur between players. Luckily, ensuring everyone's application state is synchronized is a well-known issue in P2P architectures and can be accomplished through various means, such as shared object locking and timestamps [22], [26], [32]. Unfortunately, the time gained by not having to create and maintain two applications, one for the server and one for the client, gets offset by the time lost trying to initialize a connection to other peers.

In P2P systems, especially pure P2P systems, it can be difficult to find fellow peers. By forcing developers and their users to change their firewall settings to allow for NAT traversal and, if the pure P2P route is taken, to know ahead of time where they can find each other, ASL no longer remains as flexible and easy to integrate and setup as desired. Because of this, like the client-server model, the P2P model also struggles with fulfilling the rapid prototyping goal of ASL.

2.2.3 *Client-Server/Peer-to-Peer Hybrid Model*

The hybrid model can consist of any combination of elements from the client-server model and the P2P model. This flexibility and application specificity of the hybrid model means there is arguably no generic model like those that existed for the client-server (Figure 2.1) and the P2P (Figure 2.2) model. While no hybrid generic model exists, there are still generic strengths and weaknesses of the hybrid approach. Based on the survey and analysis of existing hybrid models, these strengths and weaknesses can be summarized as:

- Strengths:
    - Flexible
- Weaknesses:
    - Complex

Most often, the hybrid model is implemented when a user needs to overcome the limitations their current model comes with, but they do not have the money or time to do so strictly in that model. For example, the flexibility of the hybrid model is frequently found in heavily trafficked client-server models such as video game servers. Here, developers can reduce the work load of their servers, effectively solving the scaling weakness of the client-server model by either pushing some of the state management to the clients [16], or by turning some clients into servers themselves [17], [33], [34]. Another example of the hybrid model's flexibility is its ability to speed up the player connection process in the P2P model. While the flexibility the hybrid model grants and encourages is substantial, this flexibility comes at a cost.

To implement the hybrid architecture, one must have a notable amount of knowledge on both the client-server system and the P2P system. Without this knowledge, one might accidently give players the ability to access private state information [16] or reduce the performance of their application [34]. While how these hybrid systems work can be simple in the end, the amount of planning and careful execution that must go into them inherently complicates the system and is why they often have special use cases.

The flexibility the hybrid model offers to ASL is a huge boon. It is this flexibility that can allow the rapid prototyping goal of ASL to be accomplished. Any strengths or weaknesses from the client-server or P2P model that prevented ASL from accomplishing its rapid prototyping goal can just be integrated via the hybrid model. In turn, ASL must ensure that the potential complexity of the hybrid model does not slow down users by keeping its user interface, or the code that the user will call in their application, simple, straightforward, and repeatable.

2.3 PREVIOUS WORK – ASL 1.0

The first version of ASL was built based on the Photon Unity Network (PUN) library [7]. This choice mandated the client-server model. By using PUN, matchmaking, or connecting to other players, would occur through PUN's servers and then any data processing would be pushed to the

host client [35]. This host client, or “MasterClient” as PUN calls it, would then be for all intents and purposes the server and everyone else in the match would be a client [36].

This demand for separate server and client applications ended up being the main reason ASL 2.0 was built. While PUN allows user’s code to be physically organized in the same application, there was still a logical separation that had to be performed between the functionality of the server, or MasterClient, and normal clients. This separation caused a slowdown in development for ASL users.

In addition to creating two separate applications, users also had to learn two separate libraries. Since ASL was built and evolved as CRCS’s need demanded it, it was not based on an explicit architecture. This ‘develop as you go’ style of building ASL meant that PUN was not well shielded inside of ASL. This inevitably led to users having to learn ASL as well as PUN, effectively forcing users to learn two separate libraries to create a single multi-user collaborative application. This increased the amount of time users spent working on making their application multi-user and reduced the amount of time they contributed to developing collaboration features.

Evolving the system without an explicit architecture also led to a lack of a coherent interface and documentation. While the PUN library is well documented with tutorials and examples, ASL’s documentation was not. At best, ASL would contain a description of a class or function with no example on how to use it, and at worst, ASL would contain no documentation at all for that class or function [37]. This lack of documentation slowed down development as no one knew how to work with ASL without first analyzing the implementation source code.

As the system continued to evolve it was discovered that ASL could not be used to quickly get CRCS collaborative applications built. Having to learn two separate libraries, create two logically separate applications, and a general lack of documentation prevent students from creating the exploratory applications CRCS desired. These discoveries led to the creation of ASL’s third goal – the ability to create rapid prototypes.

To help support the goal of creating rapid prototypes, ASL must include documentation that not only allows users to rapidly familiarize themselves with the library, but also allows for the library to continue to live on after the initial developers have left. Thus, ensuring that as ASL evolves, it evolves properly and in line with its architecture. Taking the lessons learned from the first version of ASL and keeping all ASL’s objectives in mind, the new version of ASL was created.

Chapter 3. SPECIFICATION AND DESIGN

The following section will start by discussing why ASL's requirements led to choosing the hybrid model. Next, this discussion will follow with what this hybrid model looks like. Then, the chapter will end with why this particular hybrid model not only satisfies ASL's goals, but also allows it to be technology independent, capable of being built on top of any graphical engine and backend system provider.

3.1 Choosing the Hybrid Model

The hybrid model best supports the objectives of ASL. While all three models could be used to connect and communicate data, were device independent, and could be well documented, the hybrid model best supports ASL's users in rapidly prototyping their application.

While the client-server model appeared to be a good fit initially for ASL as it allowed for quick and easy connections, it does not lend itself to rapid prototyping because of the two applications that needed to be created for each collaborative application, a server version, and a client version.

The P2P model overcomes this two-application version problem by pushing state management to the clients. However, a straightforward P2P model would require much logistic overhead to find and connect users.

The hybrid model can combine the quick and easy connection process of the client-server model with the state management process of the P2P model. Thus, taking what ASL benefits from both models and combining them into one architecture. The hybrid model is the best model for allowing ASL users to rapidly prototype their applications and is therefore the model that ASL will implement.

3.2 Designing the Architecture

The hybrid model ASL will implement is structured around the client-server model but implements a key P2P model behavior to ensure users can still rapidly prototype their application.

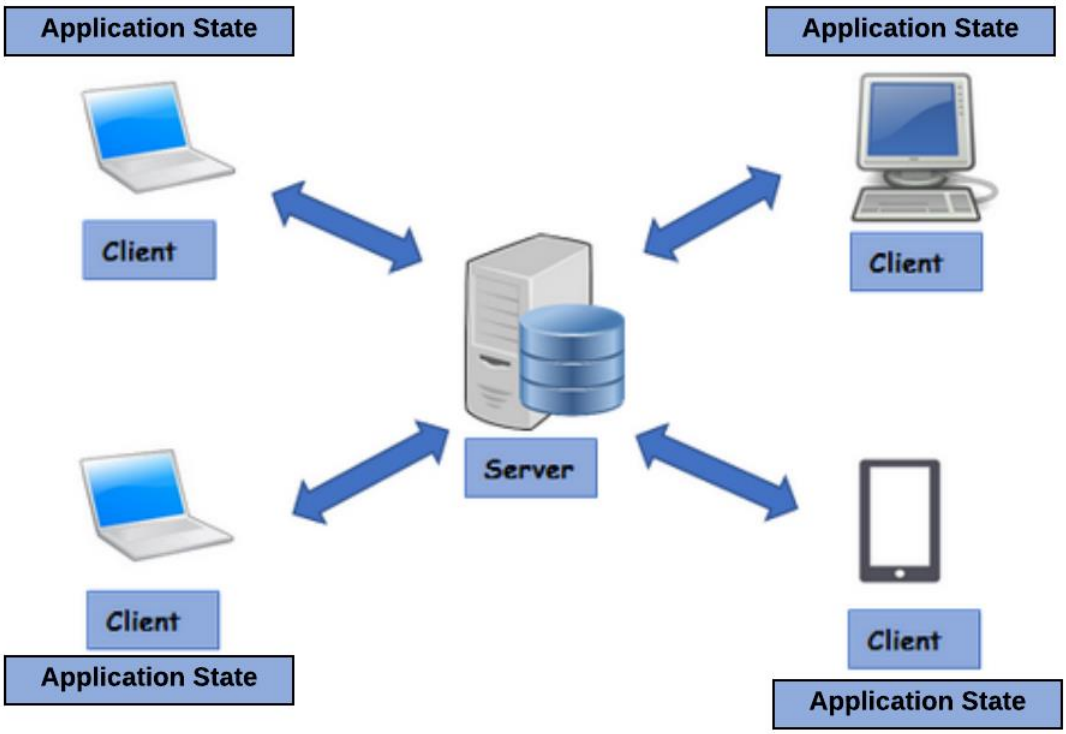


Figure 3.1. A diagram of ASL's hybrid model[5]

Taking advantage of the main benefit of the client-server model, players will be able to find and connect to each other by first logging into the ASL server. From here, players can communicate to each other via a relay server that contains minimal application state information. This architecture is depicted in Figure 3.1.

Following the P2P model, the ASL hybrid model delegates the application state management to the clients. The client state synchronization challenge is then met by dedicating unique identification tags for every shared object, ensuring that the server can identify, lock, and assign ownership to these objects. Although lock-and-share synchronization is not unique [22], being the only form of state that the ASL server maintains, this design is unique in supporting 3D object synchronization.

By keeping track of player ownership on shared objects, the server can then approve or deny requests for manipulations of those objects. This is the only state information that the server maintains to ensure all shared objects are properly synchronized.

This minimal state management system allows ASL users to design and focus on the functionality of a single application so that they may rapidly prototype ideas. However, it is also the case that the success of an object manipulation request depends on the current ownership of that object. This means, for example, that shared objects with autonomous behavior, e.g., results of a physics system simulation, require dedicated logic for locking and updating. Even if claimed before performing these autonomous manipulations, users must perform extra steps to properly

[5] Figure has been modify from its original source to reflect ASL's hybrid model design [11]

synchronize these actions across all users. Additionally, in the situation when multiple players attempt to lock and manipulate the states of a large number of shared objects simultaneously and continuously, the corresponding network traffic and denied requests due to ownership contention could result in significant network delays.

To support a single application state for rapid prototyping, ASL is designed to support collaboration in relatively static environments and does not attempt to address the condition where there are a large number of shared objects with autonomous state changes, such as A.I. behaviors or physics simulations. For example, it is straightforward for an ASL based application to investigate the collaborative exploration of a building with relatively static objects inside. However, if in this same building, there are large number of objects that are under some simulation with autonomous state changes instead, then ASL may not be the best library to support the investigation.

3.2.1 *Object Ownership and Sharing Model*

Traditionally, there are two ways lock-and-share synchronization can be implemented [38]. The first is distributed control, where the current owner must release a lock voluntarily [22], [39]. The alternative is a centralized control, where locks are granted and removed by the server [38]. ASL once again, choses a hybrid approach by allowing owners to voluntarily release a lock when they are done with the object and at the same time forces lock releases if another user requests that object. It is the server that issues the lock release command and it is the server that arbitrates lock ownership with a queue of exactly one client.

ASL implements this hybrid lock-and-share synchronization system because of observations from ASL 1.0 applications. Players typically take turns to manipulate shared objects with other collaborators observing. While the turn period varies, in a collaborative environment, the transfer of ownership is typically an orderly sequence, and thus it is very seldom for players to aggressively compete for ownership. ASL is optimized for this ownership transferring behavior. It is expected that a user would release ownership when they are done with the object. In the case where a user should forget to release ownership, or when a friendly collaborator should decide to interfere and desires to manipulate the object themselves, the server will then break the lock and actively re-assign ownership.

At any instant, if the server should receive more than one ownership requests, only one will be honored and the rest denied. There are two important reasons behind this strategy. The first is to guarantee definitive responses to the interactive client applications. Without an ownership wait queue in the server, the client application is guaranteed a success or failure response within the expected network delay. With this deterministic system in place, each player either has the control or not, and is therefore never in a state of owning a lock sometime in the unknown future. The second reason is simplicity—without explicit queues, the server state management can be kept simple and elegant.

3.2.2 *Claiming an Unowned Object*

This scenario describes what will happen if a user claims an object that the server owns, or an unowned object.

Time Step	Client	Server	Server State	
			Object Owner	Outstanding Claims
1	Player A Claim Object →		Server	None
2		Recieved Player A's Claim	Player A	None
3		Player A is Owner ←	Player A	None

Figure 3.2. When a player claims an objected owned by no one

In Figure 3.2, Player A attempts to claim an object that currently has no owner. In this scenario, Player A's claim is approved by the server, and the server sets Player A as the new owner and then sends a message back to Player A informing them that they are now the owner and can manipulate the object.

3.2.3 *Claiming an Owned Object*

This scenario describes what will happen if a user attempts to claim an object that is owned by another user.

Time Step	Client	Server	Server State	
			Object Owner	Outstanding Claims
1	Player A Claim Object →		Player B	None
2		Recieved Player A's Claim	Player B	Player A
3		Player B Release Claim ←	Player B	Player A
4	Player B Releases Claim →		Player B	Player A
5		Recieved Player B's Ownership Release	Player A	None
6		Player A is Owner ←	Player A	None

Figure 3.3. When a player claims an objected owned by another player

In Figure 3.3, Player A attempts to claim an object that is currently being owned by Player B. Since ASL is built for collaboration and not competition, Player A can obtain ownership from Player B. Once Player A's claim reaches the server, the server makes note that there is an outstanding claim on that object and then tells Player B to release its ownership. This occurs even if Player B just received its ownership. Player B will then perform any last actions on that object by clearing its successful claim callback queue and then release its claim. Once this release message is received by the server, Player A will be informed that they now have control of the object. Once Player A receives this message, just like before, they can then perform whatever networked action they want on the object.

3.2.4 *Claiming an Owned Object That Has an Outstanding Claim*

In this scenario, two users attempt to claim an object that is owned by another user.

Time Step	Client	Server	Server State	
			Object Owner	Outstanding Claims
1	Player A Claim Object →		Player B	None
2		Recieved Player A's Claim	Player B	Player A
3		Player B Release Claim ←	Player B	Player A
4	Player C Claim Object →		Player B	Player A
5		Recieved Player C's Claim	Player B	Player A
6		Player C Claim Rejected ←	Player B	Player A
7	Player B Releases Claim →		Player B	Player A
8		Recieved Player B's Ownership Release	Player A	None
9		Player A is Owner ←	Player A	None

Figure 3.4. When two players claim an object owned by another player

In Figure 3.4, a similar scenario to that of Figure 3.3 occurs except that there is now a third player who is also attempting to claim the same object. In this scenario, when the claim from Player C reaches the server, the server denies Player C's claim as there is already an outstanding claim for that object – Player A's claim.

By implementing the claim system in such a manner, it allows applications to stay synchronized by keeping the server deterministic. Players needs to know whether they can interact with an object. If Player C were instead queued behind Player A, Player C would hang until Player A finally released the object. By that point, they may have already moved on to something different or the manipulations they wished to perform after successfully claiming the object may now be invalid. By only keeping a queue of one outstanding claimer, ASL can ensure synchronization across all applications.

3.2.5 *Discussion*

In Figure 3.4, while it is possible for Player C to never receive ownership, the potential for this starvation is mitigated because it takes an undetermined amount of time, due to network round trip delays, for a user to break and receive ownership. This unknown delay time results in different outcomes for identical request patterns. The chances of the same player always receiving the same results from ownership contentions is low, especially since the number of players in an application at any one time will be relatively small.

When a player disconnects, the server will release all object ownership for that player and cancel all their outstanding ownership claims. To maintain simplicity, the server will assume the ownership of all objects that are relevant to the disconnecting player, including those that are undergoing ownership transitions.

3.3 SATISFYING ASL'S REQUIREMENTS

ASL's hybrid model architecture satisfies all the ASL's requirements that were discussed in Section 2.1. It allows users to connect and share data with other users by allowing users to locate each other through ASL's server. Once located, it allows users to connect and pass information to each other using ASL's server as a relay server, literally forwarding each received packet to all users, including the sender.

The hybrid model also allows ASL to be device independent. If a device has access to an internet connection it can be used with ASL as ASL's architecture does not depend on any specific hardware setup. This means ASL can work on traditional computers, mobile system AR devices, immersive VR devices, and immersive AR devices.

ASL's hybrid model facilitates the rapid prototyping of applications. It allows users to quickly install ASL as no setup is necessary other than importing ASL's SDK into their own project. It also gives users the ability to easily connect players due to its integration of a server that all users are already point to when they import ASL. Finally, and most importantly, because of ASL's choice in pushing all but the minimum state management onto the clients, users only have to create one application version for their collaborative application – the client version.

In terms of documentation, while ASL's hybrid model does not accomplish the goal of being well documented by itself as it is just an architecture, it does encourage the creation of a well-documented system. By knowing the architecture and what it will support and how it works before implementing the library, documentation can be generated as the library gets implemented without fear of the system changing drastically.

Finally, while not an ASL requirement, it is important to note that this architecture design is technology independent. All that is required is a server to help facilitate connections and object claims. This server can run on any system and can be connected to any platform. The next section will cover the platforms and the reasons that they were selected, as well as how ASL's hybrid architecture was implemented.

Chapter 4. FRONTEND IMPLEMENTATION

This chapter will discuss how ASL was implemented from the client's, or user's, perspective and how it stayed true to ASL's architecture. The first subsection will cover why Unity was chosen as the engine to build all ASL applications. The second subsection will cover how users can share objects by discussing the claim system implementation, the generic ASL object manipulation methods, and the generic ASL synchronization functions.

4.1 Choosing a Frontend Engine

As CRCS does not have the time or resources to create their own frontend platform for building collaborative applications, a graphical application engine was needed. This platform would need to present ASL functionalities through a friendly graphical user interface where 3D worlds can be interactively constructed and edited to facilitate the creation and investigation of remote collaborative applications.

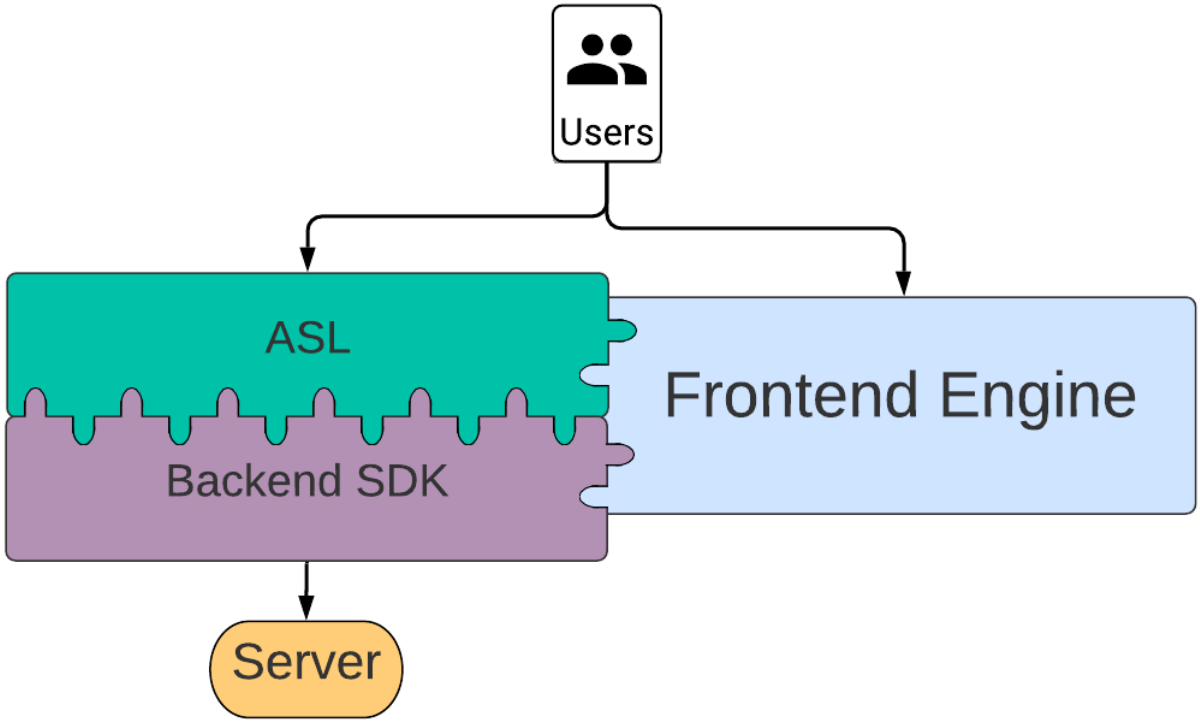


Figure 4.1. A high-level view of ASL's system implementation

A high-level view of ASL system implementation can be seen in Figure 4.1. ASL is designed to be a library of an existing graphical application – the Frontend Engine in Figure 4.1. In this way, ASL allows its users to take advantage of the frontend engine's functionality while supporting the creation of remote collaboration applications that satisfy CRCS's needs. This design

hides the network server and data sharing details via the Backend SDK. The Backend SDK and server implementation will be discussed in Chapter 5.

Since one of the main goals of ASL is to facilitate the collaborations of VR and AR setups, the hosting Frontend Engine should support as many VR and AR devices as possible. This platform should also facilitate rapid development by providing good documentation and a fast testing cycle. After examining the potential Frontend Engines for hosting ASL, including Unity, Unreal Engine, and Amazon Lumberyard, it became clear that Unity is the best option for ASL and CRCS.

Unity supports all major VR and AR platforms, has good documentation, offers special accessing plans for educational institutions and students [40], and plans to continue to improve its support for these and future devices [41]. It has a healthy user community with a plethora of online tutorials [42], [43]. These factors, and the fact that CRCS already has experience working with Unity, make Unity a great choice to build ASL on top of and thus also to become the backbone of CRCS's exploration into heterogeneous multi-user collaboration applications.

It should be noted that while Unity is a game engine, ASL applications may not be games. CRCS is interested in studying how multiple users, each on various devices can collaborate in shared and remote environments. These types of environments often include game-like elements, including real-time player interactions with virtual 3D objects [44]. These elements are meant to assist the exploration of collaboration across the different hardware mediums.

4.2 ASL FUNCTIONS

The functions an ASL user can call to ensure their player's application states stay synchronized are discussed in the following subsections. The first subsection will cover how objects can be shared, synchronized, and manipulated. The next subsection will discuss some of ASL's unique functions that either perform a specific task for a user or act as catch-all function that can allow users to accomplish various tasks. Finally, the last subsection will discuss ASL functions that are not related to any particular object. For more information on the type of functions ASL provides its users, see [45].

4.2.1 *Object Sharing*

From previous experience it was learned that the key to supporting state synchronization amongst all peers centered around sharing non-static 3D virtual objects. Since ASL uses Unity, the method ASL implemented for determining if an object is a shared object is to have users attach a script called ASLObject, to each object they want to synchronize. ASL objects are given a unique identifier upon their creation that is then shared with all other users, so that every user will have the same unique identifier for the same object and can therefore perform synchronized actions on the correct object. To call an ASL function that effects a shared object, that object must have the ASLObject script attached to it. To manipulate a shared object, a player must have ownership of that object.

To claim an object, a user must call the SendAndSetClaim() method. It allows users to claim an object and then once that claim is successful, to manipulate it with one of the other ASL methods.

```
AnASLObject.GetComponent<ASL.ASLObject>().SendAndSetClaim(() =>
{
    //Perform an ASL function here
});
```

Figure 4.2. How a user can claim an object and then manipulate it

Figure 4.2 illustrates how a user can call SendAndSetClaim(). AnASLObject is a Unity GameObject [46] with the ASLObject script attached to it. The first line of this code accesses the ASLObject component and then calls SendAndSetClaim(). The SendAndSetClaim() method will then send a request to the server, as described in Sections 3.2.1 through 3.2.3. Depending on the success of the claim request, the user's code in "$//Perform\ an\ ASL\ function\ here$" may or may not be executed. If the claim was unsuccessful, the claim rejection callback method will be executed.

The simple pattern of claiming an object and then performing any ASL manipulation on that object also helps ASL users rapidly prototype their application as it is easy to repeat and remember. If a user wants to manipulate a synchronized object all they must do is claim it and then call their ASL manipulation methods inside their claim method. To maintain synchronization, players do not perform these manipulations, e.g., actually move their object, until they receive the message from the server, even if they are the player who called that manipulation in the first place.

This method of claiming an object before manipulation works well as it guarantees application state synchronization; however, it does have some drawbacks. The separated lock and then manipulate network steps mean that there is a potentially noticeable delay between a player's action and when they see the results of that action. The slower the player's internet connection, the more pronounced this delay becomes. Therefore, this system is not well-suited to handle applications that require quick response reactions or interactions from the player. Such interactions are typically vital for video game applications, especially when there are a large number of autonomous objects affecting each other's state, e.g., collisions between projectiles and objects. However, in a collaborative environment, where objects are typically passively manipulated by users, the network roundtrip delay is much less of a problem.

The second drawback is that this system adds a level of complexity for the user to deal with when a claim is rejected. The system invokes different callback methods depending on the success of an ownership request. The user must define a proper behavior for failed ownership requests, which not a typical concern for common interactive applications. A simple approach to overcoming this complexity is to ignore claim rejection and instead continuously attempt to claim the object.

As mentioned, upon claiming an object, a user can manipulate it via ASL methods. It was learned from the first version of ASL that most interactions in CRCS applications revolve around

changing the transform (position, rotation, and scale) of shared objects, therefore most ASL methods involve performing these generic manipulations in some form.

To help communicate ASL's simple naming and actual object manipulating schemes, the following subset of ASL methods are presented. Each of these methods would be called from inside the SendAndSetClaim() method, or where "*//Perform an ASL function here*" is in Figure 4.2.

- SendAndSetLocalPosition(Vector3 _newLocalPosition)
- SendAndSetLocalRotation(Quaternion _newLocalRotation)
- SendAndSetLocalScale(Vector3 _newLocalScale)

Other manipulations include the ability to set an object's world position, rotation, or scale and the ability to increment these values, both locally and worldly. By choosing to implement object transformations on an individual level, ASL reduces the amount of potential wasted calculations and minimizes packet size. As the number of times each transform component is changed often differs, e.g., an object's position is more likely to be updated than an object's scale, it makes sense to separate the update of transforms into individual components.

ASL only allows explicit object modification and does not offer a listening system that supports indirect object updates. To support other transform manipulation systems, like a physics system, additional steps are required. While ASL is not designed specifically to support applications with physics simulations, in general, it is often desirable to be able to smoothly modify an object in a believable manner and having to take extra steps to synchronize these objects is arguably the weakest aspect of ASL.

4.2.2 *General State Modifications*

Sometimes a user may want to perform an action on an object that does not involve manipulating its transform. As CRCS cannot know every action a user may want to synchronize, ASL needed the capability to allow users to implement their own network functionality. Previously, ASL accomplished this by allowing users to add their own functionality to ASL. While this is a good thing when that functionality will be used by multiple projects, it was often the case that after a project was completed, that functionality never got used again which slowly lead to ASL suffering from feature bloat. To avoid feature bloat, ASL came up with the idea of a SendCharArray() method.

The SendCharArray() method would allow users to send any char array. This char array would be linked to a predefined user method that upon being received could perform any action the user implemented based on the sent char values. However, this method had a couple of problems that prevented it from ever being implemented. While it was highly flexible, it was extremely complicated from a user's perspective. Users would need to perform data conversions in and out of the char array, implement a system to spilt their char array appropriately for their contained data, and create and link the method they wish to execute upon receiving the char array. While this highly flexible method could implement any synchronized functionality the user

desired, the complexity of this method was deemed too high. To still give users the capability to synchronize various non-transform actions, a similar, but simpler method was implemented, the SendFloatArray() method.

The SendFloatArray() removes the complexity of having to perform data conversions and array splitting but loses the ability to send any data type. However, as most state information is float based, the SendFloatArray() method still gives users the ability to synchronize almost any action they desire.

```
public static void MyFloatFunction(string _id, float[] _myFloats)
{
    //Grab the object that was used to send these floats - in this example, the float array only contains 4 elements
    ASL.ASLObject MyObject;
    if (ASL.ASLHelper.m_ASLObjects.TryGetValue(_id, out MyObject))
    {
        //Determine what to do with float values based on the first float sent
        switch (_myFloats[0])
        {
            case 0: //Debug the floats
                Debug.Log("The values sent were: " + _myFloats[0] + ", " + _myFloats[1]
                                                + ", " + _myFloats[2] + ", " + _myFloats[3]);
                break;
            case 1: //One way to move an object via Physics System
                MyObject.GetComponent<Rigidbody>().MovePosition(new Vector3(_myFloats[1], _myFloats[2], _myFloats[3]));
                break;
            case 2: //Send how many objects a player has picked up
                myObjectCounter = (int)_myFloats[1];
                break;
            case 3: //Pause the application
                Time.timeScale = 0;
                break;
            case 4: //Resume the application
                Time.timeScale = 1;
                break;
            default:
                Debug.LogError("Error. No cases implemented for this key value: " + _myFloats[0]);
                break;
        }
    }
}
```

Figure 4.3. How a user could use the SendFloatArray() method

An example of how the SendFloatArray() method can be utilized is shown in Figure 4.3. Just like the other methods that have been discussed, the SendFloatArray() method is called from inside a claim method. Once the sent float values are received by a peer, that peer (and all other peers when they receive it as well) will call the float method associated with that ASL object, or in the case of Figure 4.3, will call MyFloatFunction(). The id parameter is the id of the shared object associated with that float method and the float array parameter contains the float values that were sent.

By using SendFloatArray(), the user can create any synchronous action they desire by simply assigning float values to trigger those actions and then sending those float values when they want said actions to occur for all users. The only requirement is that they create their own MyFloatFunction() method with the same parameters as MyFloatFunction() and that they assign the float callback method e.g., MyFloatFunction(), to the proper ASL object so that all users can execute that callback method when they receive floats associated with that ASL object.

While SendFloatArray() is one of the most powerful and flexible methods in ASL a user has in their arsenal, that power and flexibility comes at the cost of complexity. Though this complexity is lower than what the SendCharArray() would have been, this method and how it works in tandem with its callback method are still one of the most complicated processes an ASL user will have to deal with. To help users overcome this complexity, the SendFloatArray() method is one of the most documented procedures.

In some cases, however, it makes more sense for ASL to explicitly offer users state manipulation methods than to continue to force them to implement their own MyFloatFunction(). Two such cases are the SendAndSetObjectColor() method and the SendAndSetTexture2D() method.

The SendAndSetObjectColor() is a method that since it was being used often enough by users, was converted into its own method to simplify its execution. This method allows the user to change the color of the ASL object that calls it for that object's current owner and the color for every other player. This method is most often used when a user wants to show players who currently owns an object.

The SendAndSetTexture2D() method was also converted for simplification purposes. This method will take a 2D texture and send it to all users. This essentially allows users to share images with each other, which, for example, can come in very handy when attempting to make all users see the same photo realistic projection in an AR collaborative application [47].

4.2.3 *General Synchronization Support*

ASLHelper is a static class that allows users to perform synchronization actions that are not tied to any specific ASL object. While these functions are global functions, they perform very specific actions for the user and increase ASL's usability and simplicity by not forcing users to create a manager reference class object everywhere they want to perform these static functions. There are three main functions a user can call from this class:

- CreateARCoreCloudAnchor()
- InstantiateASLObject()
- SendAndSetNewScene()

Typical AR devices define the initial physical position of the physical camera as the world origin. For this reason, the world origin positions of collaborating AR devices are located at different physical positions, causing virtual objects to appear at different physical positions. Therefore, it was important for ASL to provide a way for virtual objects to appear in the same location for all AR users. This is accomplished using cloud anchors which generate feature points to help align the objects users create to the same physical location for every AR user. ASL provides this method, and the ability to set the world origin for AR users, through the CreateARCoreCloudAnchor() function. Users can call this function by simply passing it a location to spawn a cloud anchor which is usually generated via a finger touch on an AR associated plane.

As every cloud anchor has an id, all ASL does is ensure that once this id is created through Google's ARCore SDK [48], it is shared with other users so that they may find the same cloud anchor on their application. The world origin among all AR applications are synchronized by selecting and dedicating an anchor as the reference to all other objects. This causes all objects to appear to be in the same location, even though they are not.

This approach of simply referencing an AR anchor as the origin allows users to have the same world origin, however, this solution does not work on a non-AR device as they do not have access to anchor functionality. This downside means that currently if an ASL user wishes to synchronize non-AR devices with AR devices, they must instead use the approach of parenting all objects to a single object. This will then allow their objects to remain synchronized if they only use local transforms [49] instead of world transforms [50] to manipulate objects.

The InstantiateASLObject() function offers the ability to spawn a new ASL object for all users during runtime This function has multiple overloads and defaults parameters. In its simplest form, this function can create a primitive object (e.g., a Unity cube), set its position, and set its rotation. In its most elaborate form, this function can spawn a Unity prefab, set its initial position, rotation, and parent, attach an extra component, and assign the callback function for after object creation, the claim rejected callback function, and the float callback function (e.g., MyFloatFunction()). In all cases, all peers will create the same object with the same information attached to it.

There are two drawbacks to spawning ASL objects using this methodology. The first is that a user must wait for the server to tell them to create that object (just like other manipulation functions). This means that the user does not have a handle to this object in the same code location that they created the object like a typical Unity object creation function allows. If the user wishes to perform any actions on that shared object right after creation, they must utilize the game object created callback parameter and assign their object handle in that callback function. This disconnect from creating an object and then waiting to get access to it can cause some confusion as it is not how users are accustomed to getting an object's handle.

The second drawback is, like the SendFloatArray() drawback, that this function is complicated. There are many different versions of it that a user can execute, but it is by providing the different overloads to the user that allows them to create the exact object they need.

Both of the drawbacks are the result of using a hybrid system where states are distributed and not centrally maintained, but as this function contains a large amount of documentation compared to the simpler ASL functions, there are ways for users to overcome these hurdles.

Lastly, the SendAndSetNewScene() function, grants users the ability to change the Unity scene [51] for all users. This is important from an ASL standpoint as it allows the user to transition from the lobby room scene, where players can find and connect to each other, to the initial starting scene of the user's application. But it is also important for the user as it allows them to create multiple scenes, or levels, in their application and ensure that all players move onto the next zone with each other.

To change scenes a user can simply call SendAndSetNewScene() and pass in the name of the next scene they want to load. The function will then asynchronously load that scene and once it has finished loading that scene, will inform all other users that it is ready to transition scenes. Once all users are ready to move to the next scene, a message is sent to inform users that they can finally transition to the new scene. This wait for all users method ensures that every user will transition to the new scene at approximately the same time, but more importantly, it ensures that all users will transition to the next scene once they are all capable of doing so.

This function's main drawback is that it does not provide the player any information on when the scene will finish loading. In other words, ASL does not provide a loading bar to the user. This can give the impression that the current scene is stuck or frozen. To ensure users that their application is not freezing when they attempt to transition scenes, ASL actually loads into an empty scene with text informing the user that they are either currently loading or that they are waiting for other users to finish loading. Once all users have finished loading, users are transitioned from this middleman scene to the scene designated in the SendAndSetNewScene() parameter.

Chapter 5. BACKEND IMPLEMENTATION

This chapter will cover the specific requirements a backend service must provide for ASL to successfully facilitate application state amongst all peers. After discussing what a service must provide, this chapter will then focus on what backend services were used, why they were selected, and what other services these platforms provide ASL and by association, CRCS.

For ASL to successfully implement its hybrid model, the claim system, and the rest of functionality discussed in Chapter 4, ASL's backend implementation must guarantee that packet arrival order is the same as packet sent order. To ensure this occurs, ASL must maintain its connection during the lifetime of the application and ensure that all packets arrive.

Time Step	Client	Message & Recipient	Server
1	Player A	Server: Move Object to (0, 0, 0) →	
2	Player A	Server: Move Object to (1, 1, 1) →	
3			Recieved move to (0, 0, 0)
4		Player A: Move Object to (0, 0, 0) ←	
5		Player B: Open Connection ←	
6		Player B: Move Object to (0, 0, 0) ←	
7		Player B: Close Connection ←	
8			Recieved move to (1, 1, 1)

Time Step	Client	Message & Recipient	Server
9	Player A: Recieved Move to (0, 0, 0)		
10		Player A: Move Object to (1, 1, 1) ←	
11	Player B	Player B: Open Connection ←	
12		Player B: Move Object to (1, 1, 1) ←	
13	Player A: Recieved Move to (1, 1, 1)		
14	Player B: Recieved Move to (1, 1, 1)		
15	Player B: Recieved Move to (0, 0, 0)		
16		Player B: Close Connection ←	

Figure 5.1. How peers can become desynchronized if connections are not maintained

A client must maintain its connection to the server and the server to that client the entire time the ASL application is running. If the connection is not maintained, as the example in Figure 5.1 demonstrates, packets can arrive in a different order for different users. In this scenario, the server passes along each packet once it receives it. However, because the backend implementation does not maintain connection, a new connection must occur for each packet sent for Player B; thus, the order is not maintained amongst those packets because they were sent with different connections. Therefore, while the correct end position for the object should be $(1, 1, 1)$ as that is the last message Player A sent, the end position for Player B is $(0, 0, 0)$ because that packet was

received last. By maintaining connection, only one connection must be made for the entire collaboration session, reducing network traffic, and helping packets arrive in the same order for all users. While it is true that the ASL could implement an atomic connect, claim, and manipulate structure instead of ensuring connections are maintained, just maintaining connections is more efficient and easier to implement.

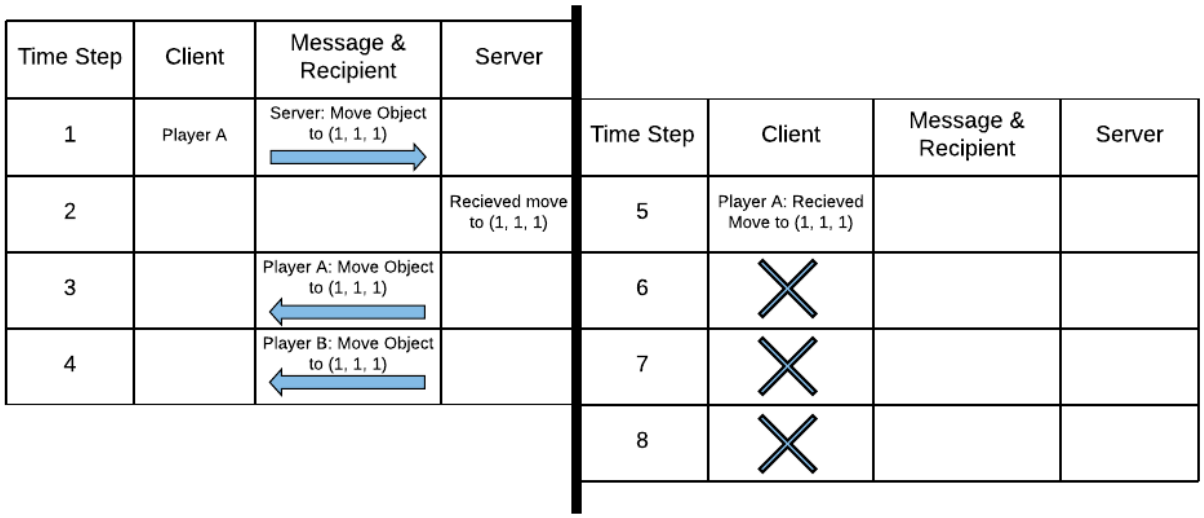


Figure 5.2. How peers can become desynchronized if packets are lost

On top of maintaining connection, to ensure packet arrival order is the same as packet sent order, ASL's backend service must also guarantee that no packets are lost. While Figure 5.1 showed how ASL can become desynchronized if connections are not maintained, Figure 5.2 demonstrates how ASL can become desynchronized if packets are do not arrive. In this scenario, Player A sends a move object message to the server and then moves to that position once it receives it from the server. However, Player B never receives this message, and therefore Player B never moves to $(1, 1, 1)$. Since the server does not maintain the state of the application with regular updates to the user, if even one packet is missed, peers will be out of sync for the rest of the collaboration session. To ensure packet order is the same for all peers, it is vital that packets are guaranteed to arrive at their destination.

ASL's backend service should implement TCP sockets to ensure that connections are maintained and that no packets are lost, thus ensuring that that packet sending order matches the packet arrival order. If sockets are not used or UDP is used instead of TCP, ASL's minimal state management system will not work, and players would quickly become out of sync as Figures 5.1 and 5.2 have demonstrated.

5.1 Choosing a Backend Service

While CRCS could create a backend service, this approach was not feasible due to the same reasons why building a frontend engine was not feasible. Therefore, ASL would need to select a backend service that offered the capability to create TCP sockets and, ideally, was inexpensive to

operate and relatively easy to install and setup. The first service that fit these qualifications was GameSparks. The second service that met these requirements and is also ASL's current backend service platform is Amazon Web Services (AWS).

5.1.1 *GameSparks*

GameSparks [20] is a relatively small company that was known as one of the leading backend service providers for video games and was trusted by many Triple-A studios such as Square Enix, Ubisoft, and Bandai Namco [52]. They had a strong and responsive community supporting them and contained lots of tutorials on how users could get started with their products [53]. They also offered, at the time ASL was looking for a service provider, a free tier for users to test out their services [54].

While other backend services have strong community support and have been well tested by known applications and companies, ultimately CRCS chose GameSparks because it provided a quick and free way to allow users to find, connect, and communicate with one another. On top of those factors, GameSparks also fit nicely with ASL's architecture style by giving its users the ability to create a lightweight server. The lightweight server ASL created with GameSparks was done so via a custom script that listened to incoming packets and relayed them to all users, including the sender. The script also managed who owned what object like the claim system described in Chapter 3, thus ensuring ASL's applications could stay synchronized. As the first round of testing on ASL demonstrated, GameSparks could ensure all peers stayed synchronized and was thus deemed an effective backend service for ASL. However, soon after completing the first round of testing, ASL's network performance began to degrade.

While nothing had changed from ASL's perspective, GameSparks started to show connection and performance issues starting in December 2019. Upon investigating, signs began showing that GameSparks was gradually being abandon. A lack of blog and forum posts along with the fact that they were now one year into their acquisition by Amazon which offered very similar services, lead to the belief that ASL should be moved off GameSparks and onto another service before it was too late [55], [56].

5.1.2 *GameLift*

After ASL's experience with GameSparks, it was determined that ASL should be migrated to a backend service that was less likely to be bought or shut down by another company. For this reason and the fact that Amazon Web Services (AWS) offered similar functionality, AWS was chosen to host ASL. Using AWS's GameLift service, users were once again able to find, connect, and communicate with each other.

Similar to GameSparks, GameLift offered its users the ability to create a custom lightweight script that their servers can then run. Since ASL went with a technology-independent architecture, the lightweight script that GameSparks used was easily converted into the lightweight

script GameLift now uses. For this reason and the fact that this backend technology migration caused limited alterations in the frontend implementation, CRCS is confident that the first round of applications that tested ASL and passed with GameSparks would also work on GameLift.

While GameLift is not free, it remains cost efficient to operate. Also, since it is a part of AWS, it gives CRCS the opportunity to use other Amazon services if the need ever arises. For this reason, and that it successfully supported the second round of testing, GameLift will remain ASL's backend service for the foreseeable future.

5.2 BACKEND SDK

To use GameLift in junction with Unity, ASL imported GameLift's Unity SDK and then proceeded to build its core library functions on top of this SDK. This process is similar to what ASL did previously with GameSparks and once again highlights ASL's technology-independent architecture. As shown in Figure 4.1, ASL's architecture has ASL sitting on top of a Backend SDK, which in turn connects to ASL's server (GameLift) and is also integrated into the frontend engine (Unity). This Backend SDK is GameLift's Unity SDK and it gives ASL the ability to connect to and send and receive data from AWS's servers. As ASL wraps this SDK, users make use of the connect and transfer data functions through ASL instead of learning the much more elaborate and complex GameLift's Unity SDK. This level of separation helps set ASL apart from its predecessor and makes it more robust for future projects.

However, because ASL is built on top of AWS's SDK, if AWS ever changes how its services receive information, ASL may no longer function properly. While this is a breaking issue CRCS must be concerned about, it is very unlikely to happen without notice. If the version of GameLift's Unity SDK that ASL is using does become deprecated, ASL will be able to once again migrate or update its service as its core concepts do not depend on any specific technology implementation. So, while CRCS should keep an eye on ASL's backend service and its SDK, it does not need to worry about ASL itself being deprecated because one of its services was deprecated.

5.2.1 *Wrapping the Backend SDK*

As mentioned previously, ASL wraps the functionality found in GameLift's Unity SDK so that users only have to learn one library. There are essentially two functionalities ASL wraps. The first one revolves around connecting to the GameLift servers and finding other users. The second thing GameLift's Unity SDK allows ASL to do is send and receive data.

To create a connection to other users, ASL utilizes the AWS Lambda function service [57]. Described as serverless functions, ASL's Lambda functions allow users to turn on ASL's server if its offline, connect to the GameLift server, find other users, and connect to these peers based on their in-application input.

ASL wraps the find and connection functionality Lambda provides into a Unity scene that users must include in their application. By doing so, users no longer have to learn how to call a Lambda function, instead all a user must do to perform the actions of finding and connecting to other users is one of two things. Either the user must attach the QuickConnect script to a GameObject and give it the name of the scene they wish to load into after connecting to other users, or they must launch their application with the ASL_LobbyScene as the first scene to execute in their application. Implementing either of these two options will allow the user find, connect, and launch into their application with other peers through ASL's connection GUI, or lobby scene.

ASL greatly simplifies the process of finding and connecting to other peers by using a GUI. While using a GUI reduces the amount of information users need to learn about ASL, it does make debugging harder to perform when a connection fails. To help make this debugging easier, ASL includes a textbox in its GUI that will output any error messages that occur while attempting to find or connect to other players. This error message can in turn then be employed by users to help pinpoint why their connection is failing.

The second main functionality ASL wraps from GameLift's Unity SDK is the ability to send and receive messages. Users indirectly use GameLift's Unity SDK's ability to send messages via the functions discussed in Chapter 4 while incoming messages are completely hidden.

Inside any ASL function that transmits data to other peers, the data the user passed in is translated into a combined byte array with minimal meta data to help deconstruct it properly when it is received. This byte array is then used to create a payload that is then sent to the GameLift server via a TCP socket. From there, as discussed previously, this packet is relayed back to all users, including the original sender.

To successfully receive a message, GameLift's Unity SDK listens to the open socket that it created on a separate thread as to not tie up any Unity calculations that need to occur. As Unity is a single-threaded application, when GameLift's Unity SDK does receive a packet from the server, it locks, queues, and then unlocks, a data structure Unity has access to. Then, whenever Unity detects that this queue is not empty, it locks the queue and empties it, effectively opening the packet it received from the server. Once the queue is empty, Unity removes its lock on the queue, thus freeing it up to be used again by the thread dedicated to listening to ASL's socket.

Instead of having to understand how to convert their data into a packet and then how to deconstruct that packet and reform the data into what it was originally and then finally perform the correct action on that data, users only have to call the simple ASL functions previously discussed. By wrapping the send and receive functionalities, ASL dramatically reduces the complexity users must deal with when creating their online collaboration application. However, by removing the user from how this system works, it makes it harder for them to debug the system when it does not work as expected. To overcome this drawback, documentation is included to give users a brief overview of what ASL does behind the scenes and how they can see packet information via the debug window in Unity if they turn on ASL's debugging option in code. How to turn on and read this extra level of debugging is also documented.

Chapter 6. RESULTS

To ensure ASL can properly support CRCS's goal of exploring remote collaboration across heterogeneous devices, the implementation has gone through three distinct steps of testing: unit, stress, and user testing. The unit tests involved verifying each of the ASL features separately. The stress tests systematically examine and attempt to discover ASL's limitations. The final step, user testing, was carried out in two phases: phase one in Fall 2019 and phase two in Spring 2020. This chapter will discuss each step and what they demonstrate. At the end of this chapter, the results of a survey of ASL users from the second phase of user testing will be discussed.

6.1 UNIT TESTING

For any API, it is vital to ensure correctness. In the case of ASL, a separate Unity scene was created for verifying each functionality. These scenes contain the minimum amount of code to ensure only a single aspect of ASL is tested. While originally created to verify ASL functionality, these simple test cases are now demonstration tutorials for users. These simple tutorials show users both from an application standpoint and a code standpoint how ASL can be utilized in their projects.

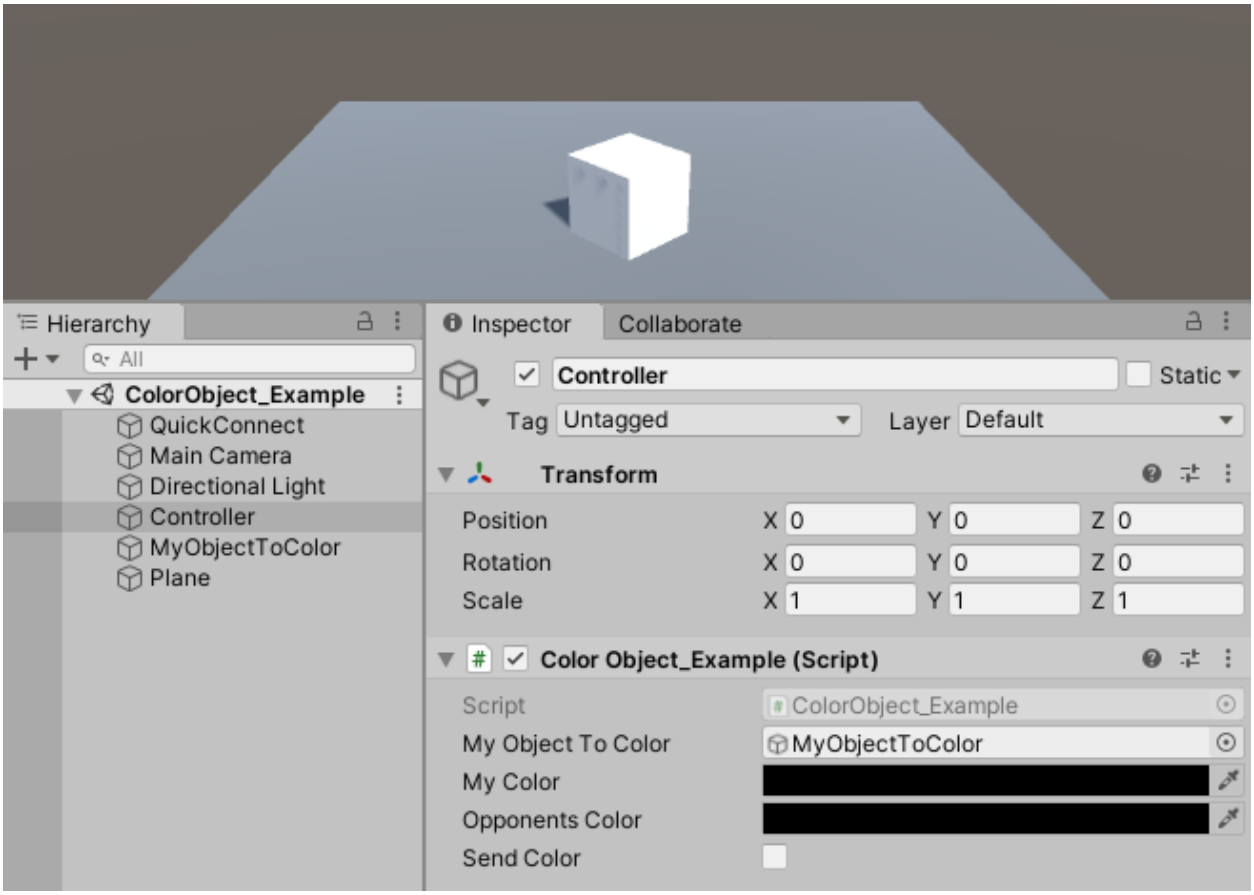


Figure 6.1. Example of how the user can interact with a simple demonstration tutorial

All simple demonstration tutorials have a similar setup to maintain consistency for readability. An example of the setup can be seen in Figure 6.1. This example shows that the simple demonstrations are meant to be examined in the Unity editor where the users have graphical interface access to the scene elements. Every tutorial has the QuickConnect and the Controller objects for straightforward connection to other players and the manipulation of scene shared objects.

As Figure 6.1 illustrates, this simple tutorial is the ColorObject_Example , which showcases the SendAndSetObjectColor() method. Through the interaction with the Controller object, this tutorial demonstrates that users can set an object's color according to its ownership.[6]

A separate, simple tutorial, set up similarly to the ColorObject_Example, is defined for every ASL functionality, including:

- ARCloudAnchors – demonstrates how AR Cloud Anchors can be utilized
- ARWorldOrigin – demonstrates how the AR World Origins can be synchronized
- ClaimObject – demonstrates how to claim an object
- ColorObject – demonstrates how to change an object's color
- CreateObject – demonstrates the different ways a user can create an object
- DeleteObject – demonstrates how to delete an object
- LoadScene – demonstrates how to load a new scene for all users
- MixedRealityToolKitSetup – demonstrates how to get started with VR
- QuickConnect – demonstrates how to launch into the ASL_LobbyScene from any starting scene
- Send2DTexture – demonstrates how to send a 2D texture
- SendFloatArray – demonstrates how to synchronize a variety of non-ASL actions
- SliderBar – demonstrates how to implement a synchronized UI element
- TransformObjectViaLocalSpace – demonstrates how to set and increment position, rotation, and scale from a local space perspective
- TransformObjectViaWorldSpace – demonstrates how to set and increment position, rotation, and scale from a world space perspective

Simple demonstrations, though originally implemented to show that ASL's functions are working as expected, now serve the greater purpose of helping educate ASL users. By running these simple demonstrations, users can examine and gain insights into the corresponding functions by interacting with their scenes in real-time.

6.2 STRESS TESTING

The second step of testing was designed to stress and examine ASL's network capabilities. These test cases also showcase how simple functionalities could be combined and thus serve as excellent

[6] "Opponent" refers to other players of this application.

tutorials on how to create standalone applications. Each stress test is its own scene and is designed to execute automatically after all users have connected. The three stress tests are discussed next.

6.2.1 *150 ASL Objects*

This stress test allows the user to load and interact with 150 ASL objects. The default 150 shared objects are meant to demonstrate the complexity that ASL is designed to support. Besides ensuring ASL could handle this number of shared objects, it was also the goal of this stress test to examine how many networked objects ASL can handle before it starts to show visible signs of performance degradation. While 150 networked objects did not cause a slowdown on any CRCS devices, a performance degradation could be observed with larger number of objects, especially on the less powerful hardware platforms such as mobile AR devices.

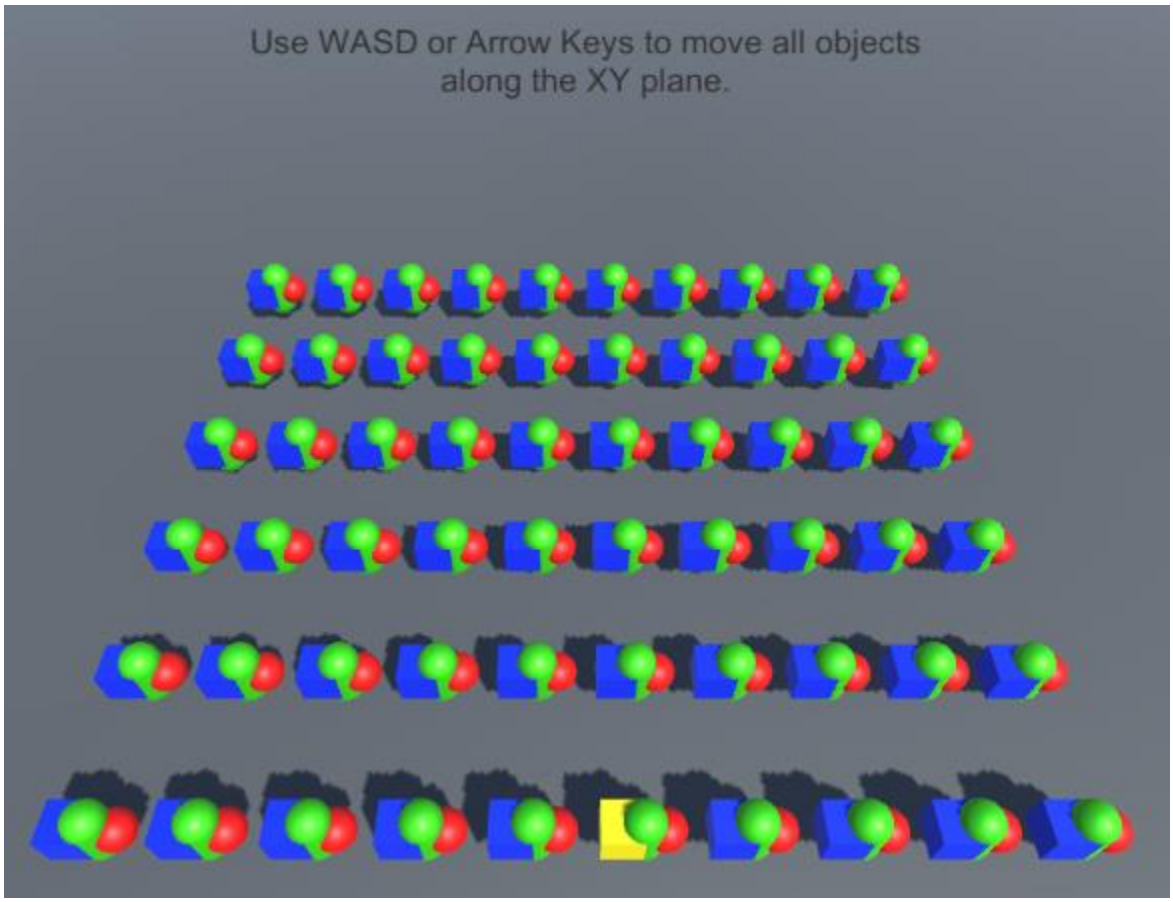


Figure 6.2. Screenshot of the 150 ASL Objects stress test

As illustrated in Figure 6.2, users can click on an object and then observe its color change to yellow, but more importantly, they can use keyboard inputs to move all objects simultaneously along the XY-plane. By allowing the user to manipulate the objects, it gives the user a sense of how fast the application is responding to their inputs, thus helping them determine if and how much they are stressing their system.

To help users determine the personal capabilities of their devices, users can add or remove the number of ASL objects in this test to any amount they desire. In this way, this test can be personalized to each user, helping them determine exactly what their hardware and ASL can accomplish. Users can also examine the code of this application to determine how they can change the color and the transform of a shared object.

6.2.2 *Create Delete*

The Create Delete stress test was built to ensure proper creation and deletion behaviors. This test performs three simple operations at random time intervals: create, move, and delete an object. By automating and executing these commands at small time intervals over long periods of time, ASL's capability to maintain consistent application states can be verified. This test case ran for 12 consecutive hours with three connecting players where in the end, all players observed a consistent application state.

Users can now utilize this test case to examine how multiple simple functionalities can be combined into a single application, such as creating an object, getting a handle to that object, manipulating that object's transform, and then deleting that object – all with different peers.

6.2.3 *Fight Over Five Objects*

The final stress test is Fight Over Five Objects. This test spawns five ASL objects that all peers in the application compete continuously for ownership, and on a successful claim, then attempt to move said object. This test also has the capability to delete a random object and to suspend all operations so that object positions can be compared across peers for consistency.

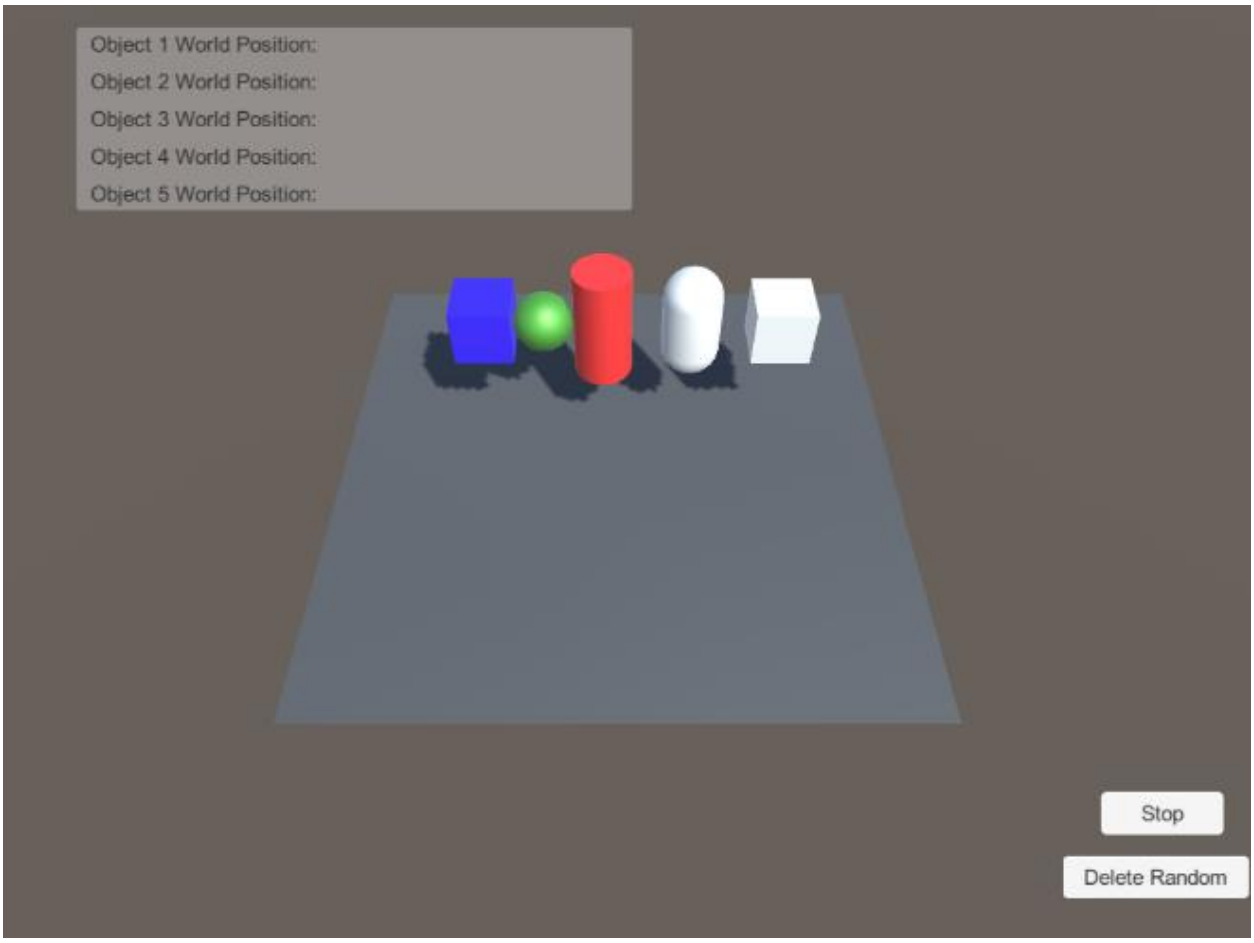


Figure 6.3. Screenshot of the Fight Over Five Objects stress test

This stress test, similar to the Create Delete stress test, was built to ensure proper synchronization across all peers. The initial application state of this test can be seen in Figure 6.3. This test ran for 48 hours with four different peers where each peer attempted to claim and move a random object at a random time between zero and two seconds. Halfway through this test, a peer was forcefully disconnected to ensure proper disconnect handling. At the end of this test, the numeric values of all positions of all objects were compared across all peers and their consistency was verified, showcasing that ASL can synchronize multiple peers over a long period of time and thousands of network calls.

This stress test can now be used by users as a way to learn how to claim and manipulate an object's position, delete an object, and how to use the `SendFloatArray()` method to synchronize various actions in their application, such as suspending all operations across all peers.

6.3 User Testing

ASL has gone through two phases of user testing. The first phase happened while ASL was still using GameSparks and the second phase is currently wrapping up at the time of writing this thesis.

The first phase of user testing was targeted specifically at AR capabilities and the second phase focused more on ensuring ASL had accomplished its four goals: user connection and data sharing, device independence, rapid prototyping, and documentation.

6.3.1 *Phase One*

ASL integrated the ARCore's Cloud Anchors [48] to support AR devices and the ability to synchronize world origins. Cloud anchors allow AR devices to recognize important physical features and then use those features to synchronize virtual objects across multiple devices. However, with this approach, all objects created are based on referencing the cloud anchor, in other words, the cloud anchors are used as local world origins. Using an arbitrary position as the world origin can create confusion for non-AR devices with no reference to ARCore or any cloud anchors. To overcome this drawback, ASL added the ability to change an AR device's world origin location.

ARCore defines the initial position of the physical camera as its world origin. In this way, with multiple AR devices the world origins are located at different physical positions. To solve this issue and to allow users the ability to synchronize objects in the physical world, ASL implemented the ability to change the world origin in AR devices. This is accomplished by transforming all objects with respect to the position and orientation of a dedicated cloud anchor selected by the application.

There were two projects that tested ASL during phase one to ensure it supported AR devices properly.

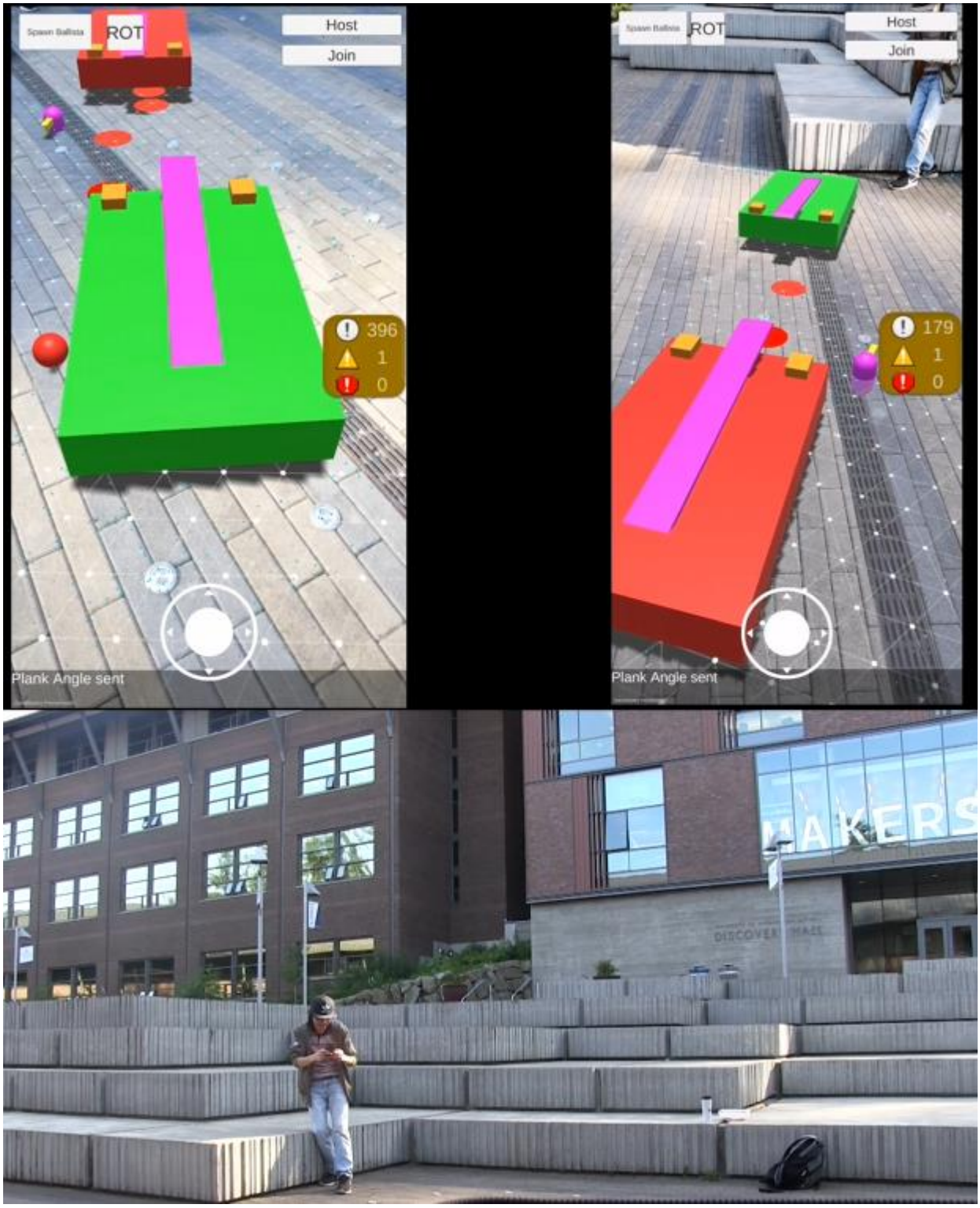


Figure 6.4. Saiful Salim's project involved multiple AR devices and simple physics

The first one, shown in Figure 6.4, was created by Saiful Salim and involved creating an application where users controlled a ballista and could see where their shots landed and rolled to.

The player controlling the green ballista can be seen in the bottom portion and the top-right portion of Figure 6.4. This project demonstrated how cloud anchors could be used to ensure virtual objects appear in the same physical location across multiple AR devices.

Figure 6.5. Jacob Lefeat's project that involved ASL, AR devices, and PC

The second project, shown in Figure 6.5, involved projecting a lava field onto the floor for all users and then having users build a bridge to cross the lava field. This project demonstrated that the AR world origin could be synchronized and that the ASL supports players on an AR device collaborating with a PC player.

While these projects focused on ensuring ASL could support AR's specific needs, they also demanded the SendFloatArray() method for synchronizing general application state and the SendAndSetTexture2D() method for sharing the lava texture. These two methods were thus included in the library.

6.3.2 *Phase Two*

It was shortly after phase one of user testing, that the problems with GameSparks were discovered. After the migration to GameLift, ASL was ready for the second phase of user testing.

In this phase there are four user testing projects. Two of these are VR based while the other two are AR based. Together, there are nine different users working on these projects: three teams of two and one team of three members. None of these users have prior experience with ASL, VR, or AR development.

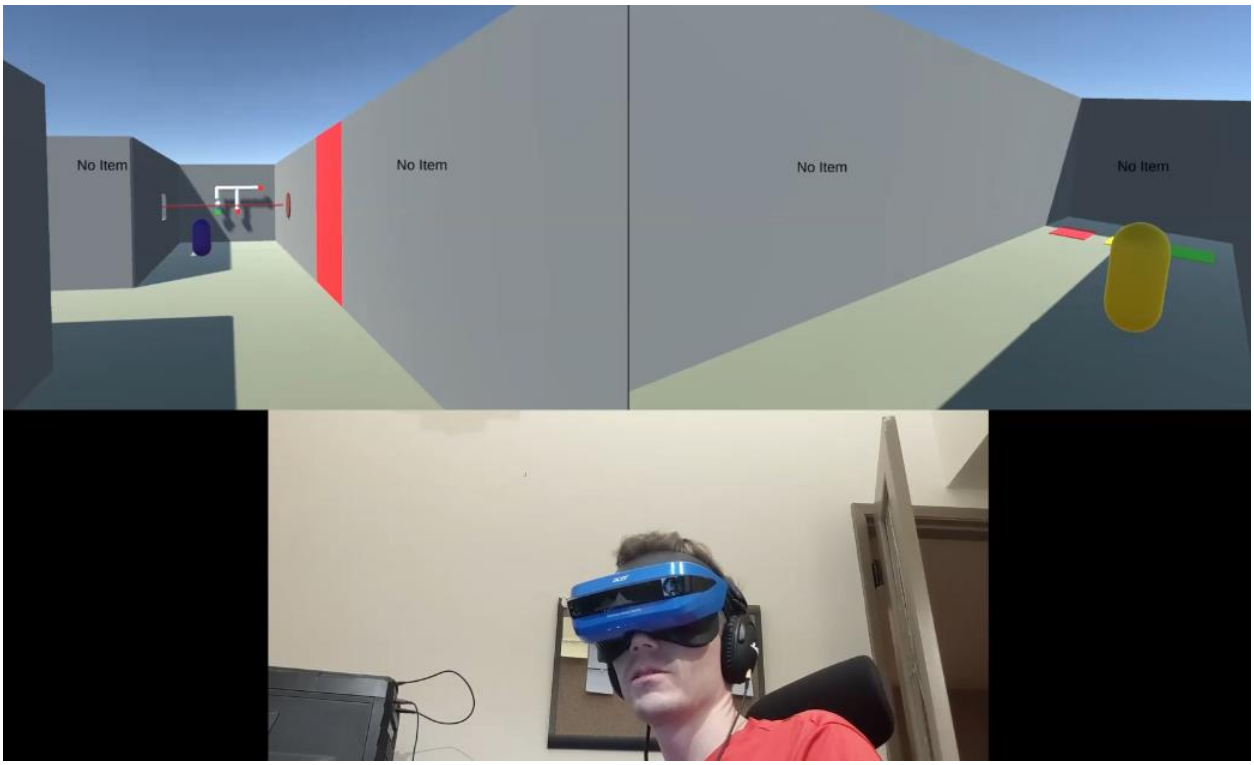


Figure 6.6. Escape VR – Players must work together to leave a virtual escape room

Escape VR, created by Cody Thayer, Isaiah Snow, and Yuto Akutsu, shown in Figure 6.6 is the first VR project. In this application, players are tasked with leaving a virtual escape room with puzzles that require multi-person collaboration to solve. In Figure 6.6, the top views are from two player perspectives, where the purple and yellow capsules represent the two players. The bottom is a view of one of the players.

This application utilizes ASL's capability to send transforms to show player movements (the capsules in Figure 6.6), move mirrors to reflect laser beams, and shuffle puzzle boards to configure pictures. When a puzzle is solved, the SendFloatArray() method is utilized to ensure all users observe the same output of the solved puzzle, such as a new door opening or a new puzzle becoming uncovered. Escape VR is able to synchronize every manipulatable object (laser beams, players, buttons, doors, and other miscellaneous puzzles) through ASL.

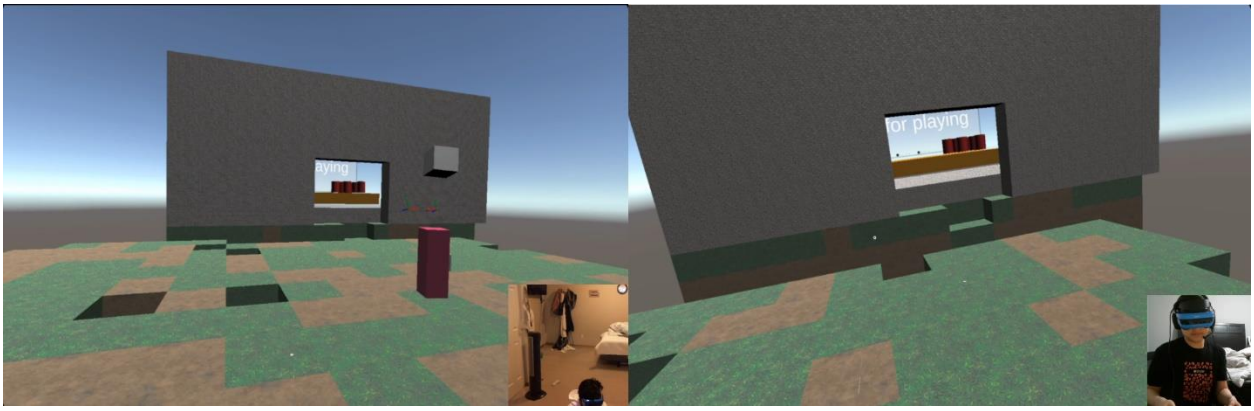


Figure 6.7. Miniature Minecraft – Players work together to reach the castle

The second VR project, though a completely different application, shows similar success in synchronizing multiple players and their interactions via ASL. In this project, Jonathan Cho and David Kim use ASL to create a miniature version of Minecraft as shown in Figure 6.7. Every block (dirt, grass, or stone) is an ASL object that players can mine and place into their inventory. Together, the two players must mine enough materials to build their way towards the castle (the building structure shown in Figure 6.7).

Figure 6.8. Cup Pong – Players attempt to throw their ball into cups

Once there, they can then play Cup Pong, a game where each player attempts to throw their ball into the cups opposite of them. The first player to land in all cups, wins. This mini game can be seen in Figure 6.8.

This VR application showcases ASL's ability to handle and manipulate a moderately large number, approximately 500, of shared objects. This project also demonstrates ASL's ability to synchronize various continuous player actions: create, mine, store in inventory, and stacking. In

addition to the blocks, ASL also synchronizes player positions, as shown by the red rectangular box in the left side of Figure 6.7, representing the right player's world position. Lastly, using the SendFloatArray() method, ASL allows the synchronization of results from the physics-based simulations in the Cup Pong mini game.

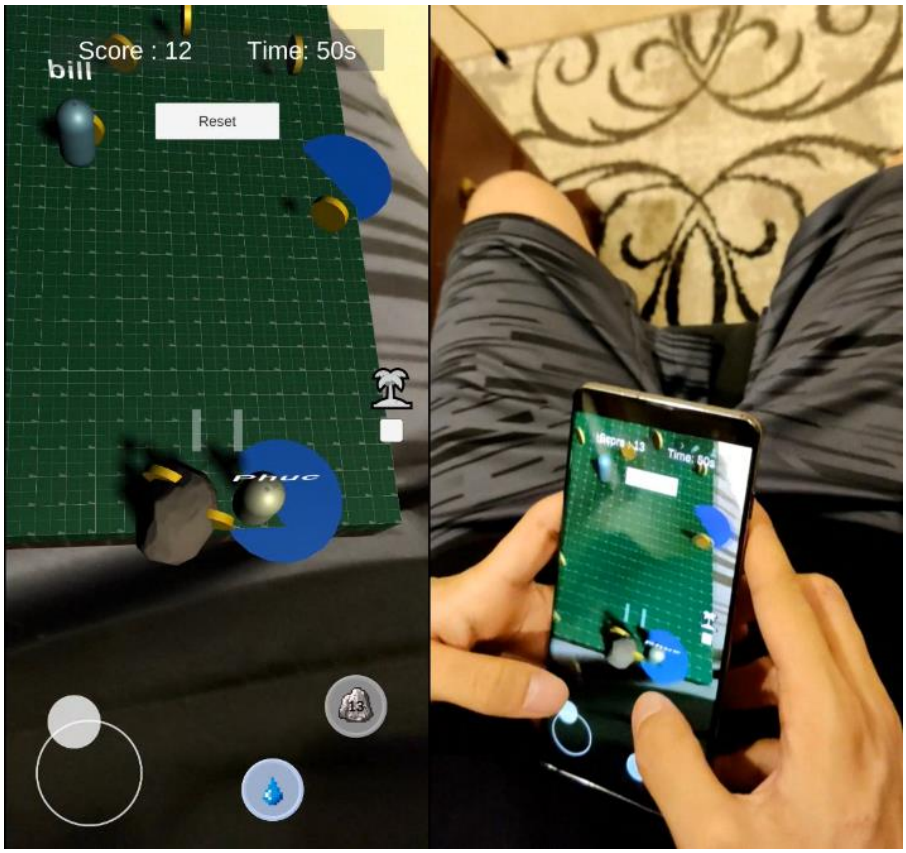


Figure 6.9. Coin Collector – Players attempt to collect as many coins as they can

In Coin Collector, an AR application designed and built by Bill Pham and Phuc Tran, players compete in an augmented world, attempting to collect as many coins as possible while preventing other players from doing likewise. Similar to the phase one lava crossing project, this AR project also involves heterogeneous devices. In this case, Player 1, bill, is on an android device (as seen via the right side of Figure 6.9), and Player 2, Phuc, is on a computer. By setting the world origin on the AR device, both the AR device and the computer can see the virtual objects (coins, player avatars, rocks, spell zones, etc.) appearing in the same virtual space. The application begins with the AR player scanning and creating a platform (the green tiles in Figure 6.9) on a physical object and setting the world origin. The geometry of this platform is then communicated to all players and the play-interaction can begin. As can be observed in Figure 6.9, the player bill is about to pick up a coin and has placed a speed trap, the blue circle, to slow down Phuc.

These coins, traps, and other spells are all synchronized via ASL support. Player movement is implemented via the SendFloatArray() method to take advantage of Unity's physics simulation

results. Whereas the coins, traps, and spells are synchronized via the ASL shared object functionality.

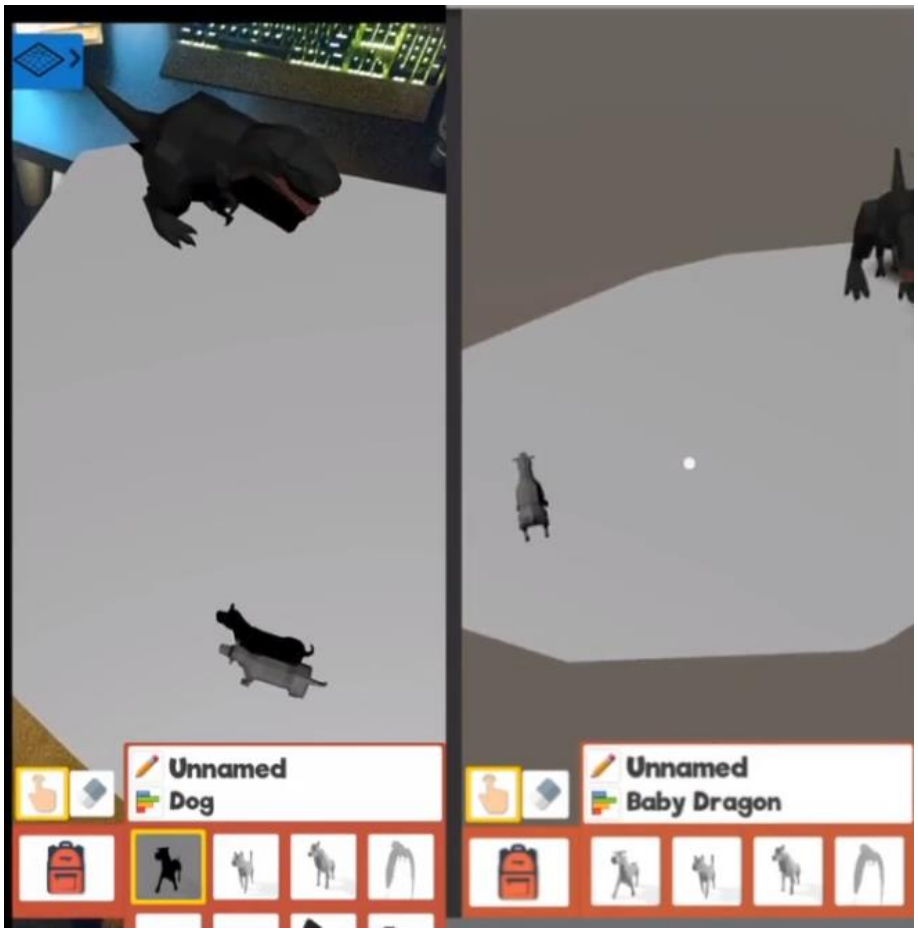


Figure 6.10. AR Pets – Application where players can show off their various AR pets

The last phase two project is the AR Pets application. This application, designed and built by Sean Miles and Marc Skaarup, allows players to spawn various pets with names that can wander around, and need food, exercise, and affection. The goal of this application is for a player to be able to show-off their pets to friends on platforms augmented on the surrounding physical objects. As in the previous AR cases, this project is also designed to work on a mobile AR device and a computer. In Figure 6.10, the view on the AR device is shown on the left side and the view on the computer is on the right side. As in the previous case, the application begins with the AR player scanning, generating, and sharing the platforms and the world origin. Figure 6.10 shows that once the platform is shared, all players can then begin placing and showing-off their pets.

As in all previous test cases, object synchronization is based entirely on ASL. Pet spawning is accomplished via the creation of shared ASL objects. The SendFloatArray() method, in addition to sharing the platform, enables the support for showing the food, exercise, and affection levels of the selected pet. Lastly, pet movements are accomplished via the ASL transformation manipulation methods.

The projects in phase two support multiple players in coherent shared environments across geographic distances, involve different combinations of AR, VR and PC device configurations, are based on ideas and implementation efforts constrained by the 10-week academic quarter, and, are built off of initial prototypes that were constructed within days of beginning of the project. These characteristics verify that in its current state, the ASL has successfully accomplished the four stated goals: supports straightforward user connections and data sharing, allows the collaboration of heterogeneous devices, facilitates rapid prototyping, and, is a well-documented SDK.

6.4 Survey Results

Towards the end of their projects, the nine (five VR, and four AR) second phase users where surveyed regarding their experience working with ASL. Opinions are surveyed based on the Likert scale with 7 being the highest. These questions and their responses are summarized in Table 6.1. The entire survey and its raw data can be found in Appendix B.

Table 6.1. A Summary of the survey questions and results

Question	Mean	Median	Population Standard Deviation
1. ASL makes it easy to find and connect players	6.11	6	0.87
2. It was easy to get started with ASL	5.44	5	0.83
3. The time it took to compile and test my code in minutes[7]	VR: 1 AR: 1.75	VR: 1 AR: 1.5	VR: 0 AR: 0.82
4. ASL was easy to debug	VR: 3.2 AR: 5.75	VR: 3 AR: 6	VR: 2.03 AR: 0.43
5. ASL's documentation explained functions well	4.88	5	0.87
6. ASL's simple tutorials let me prototype quickly	5.55	6	1.64
7. I would use ASL again	4.55	5	0.49

As shown in Table 6.1, when asked about ASL's lobby system, or the system that allows users to find and connect to each other, users agreed that it was straightforward and intuitive. Users also agreed that the application they had to create the first week of working with ASL was straightforward to create. These first two questions show that ASL is an easy library to pick up and start learning, reinforcing the idea that ASL can be suitable to support the rapid prototyping of applications.

When it came to compiling and testing their code, most VR users stated that the amount of time it took to start their application after making a change to it was tolerable, or approximately 1 minute long. However, most AR users say that this time is distracting (approximately 2 minutes)

[7] This question used time ranges as its options instead of Likert scale values. One user selected it took longer than 2 minutes to compile and test their code. To represent this value, the number 3 is used. Other number values are selected based on the time elapsed (0.5 for 30 seconds, 1 for one minute, etc.)

or unacceptable (more than 2 minutes). It is important to note that the much longer time involved is independent from ASL, it simply takes a longer amount of time to load projects onto AR devices.

The results from Question 4 shows that most users who said it was hard to debug were using VR devices. This spilt in opinion between VR and AR users suggests that VR users tend find debugging harder even though they have extra debugging tools because of their reliance on a second library, the Mixed Reality Toolkit (MRTK) [58], that runs parallel to ASL and provides users the ability to interact with VR controls. While it can be argued that no network application is easy to debug, ASL should strive to be as debug friendly as possible and therefore CRCS will be looking into ways to make it more debug friendly, including better VR integration.

The results from Questions 5 and 6 show that, in general, most users agree that ASL's documentation helped them understand how ASL functions work and that the simple tutorials sped up their development process. Between the two, more users found the examples helpful, with a mean of 5.55, than the explanations of ASL's functionality, which has a mean of 4.88.

When asked how ASL could be improved[8], most users looked for better VR integration. Due to the fast pace technological improvements, the current release of MRTK has a lack of documentation. Other requests involved more elaborate examples and better physics integration.

Finally, and somewhat encouragingly, when asked if users would select ASL as their networking library (Question 7) for their next project, independent from CRCS, most users slightly agreed, getting a mean of 4.55 and a population standard deviation of only 0.49. Users also stated that ASL has a "very feature-rich start" and that its lobby system is "quite intuitive". These comments and the overall attitude towards ASL suggest that while ASL is not a perfect library and can be improved, it does assist users in building and accomplishing their goals.

[8] The two questions regarding ASL improvements were short response questions and therefore not included in Table 6.1

Chapter 7. CONCLUSION

ASL successfully met its four goals. It allows users to connect and communicate with each other, it is device independent, allows users to rapidly prototype their ideas, and it is a well-documented SDK. By meeting these four goals, CRCS can continue its study of the issues relating to supporting collaborations across geographical distances through different technological reality setups.

Whereas most networking libraries for 3D virtual applications require users to implement server-side code to create remote applications [19], [59], [60], ASL does not. By creating ASL with a network hybrid model and a hybrid lock-and-share model, ASL stands out as a unique library, capable of supporting users in the creation of their collaborative applications. Without these hybrid models, ASL would not be able to achieve its goal of allowing users to rapidly prototype their applications.

By using a hybrid network model, ASL users can create video game like applications without the need for a centralized state manager, greatly increasing the speed at which they can develop applications by reducing the amount of time they must spend learning to micromanage two application states (server-side and client-side), and by reducing the amount of network programming they have to know.

By using a hybrid lock-and-share model, ASL ensures applications can stay synchronized even though each state is managed individually. Additionally, by using this hybrid approach, ASL differentiates itself from other lock-and-share methods by allowing users to give up their claims voluntarily but forcing them surrender their claim if another user requests that object.

7.1 FUTURE WORK

While ASL accomplished all its goals and allowed nine users to create exploratory applications within a 10-week time frame, ASL can still be improved. In general, the following is a list of improvements CRCS could implement for ASL.

- Documentation additions
- Support Autonomous Objects and Behaviors
- Time synchronization
- Allow late joiners
- World origin support for non-AR devices
- Better VR integration
- Encrypt data packets
- More use case features

7.1.1 *Documentation Additions*

There is a variety of small documentation improvements CRCS can apply to ASL. To begin, users mentioned that sometimes it could be hard to find the specific function they were looking for, or that the description of a function was lacking. These can be improved by ensuring the documentation website can be easily searched using keywords and by better describing the what and the why of an ASL function. Additionally, users requested more specific simple tutorials, such as how to move a player via Unity's physics system, and more elaborate tutorials to help showcase how all ASL functions can work in tandem to create an application.

7.1.2 *Support Autonomous Objects and Behaviors*

ASL did not include any functions or methods based around synchronizing autonomous behaviors, such as physics simulations or A.I. interactions, due to the vast amount of different ways a user may want to implement such systems. However, it has come to CRCS's attention that at the very least, ASL should include some basic autonomous behavior support that users can then build off of if desired. By implementing such methods or functions, users will be able to spend less time crafting their own version via the SendFloatArray() method and more time creating their application.

7.1.3 *Time Synchronization*

Currently ASL does not support any time synchronization. Implementing such a feature can ensure that players with slower connection speeds can remain synchronized if they fall too far behind. This feature could ensure that timed events happen at precisely the same time for all users, regardless of their internet speeds, such as every user loading into a scene at the exact same time, instead of joining the scene once they receive the packet that lets them know all users are ready to move into that scene like they do currently. Time synchronization can also help ensure autonomous objects of any kind can stay synchronized.

7.1.4 *Allow Later Joiners*

To keep things simple, ASL does not allow any user, new or returning, to join an application after it has started. Ideally, ASL players would be able to join an application after it has started and find themselves synchronized with the latest state information. ASL players should also be able to rejoin an application if they were disconnected from the original session.

7.1.5 *World Origin Support for Non-AR Devices*

While setting the world origin synchronizes AR devices' world origin positions nicely, it requires extra steps to be used with non-AR devices. Since non-AR devices do not have access to cloud

anchors, when AR devices are used in tandem with VR or PC players, ASL users are forced to implement extra functionality to ensure all players are synchronized properly. To make development faster for combined AR and non-AR applications, ASL should implement such functionality inside its preexisting `CreateARCoreCloudAnchor()` function.

7.1.6 *Better VR Integration*

Perhaps the biggest improvement CRCS can make to ASL is better VR integration. While ASL can be used to create heterogeneous applications, due to the ever-changing nature of VR and AR technologies, ASL does not integrate these technologies as well as originally hoped. Specifically, as evident by user feedback, VR integration could be improved. Unity recently changed their architecture to better support VR and AR integration and by doing so, forced VR SDK providers to update their libraries as well. This process is still on-going and will be for the foreseeable future. However, just like with AR technology, ASL can still implement some of the specific VR technology to help aid users in the creation of their applications. Currently, ASL does not prevent MRTK from working, but users, due to a lack of documentation on MRTK's part, struggled with getting started in VR development. To help aid future users, ASL can provide simple tutorials for MRTK, ranging from controller inputs to the specific UI elements VR provides.

7.1.7 *Encrypt Data Packets*

Due to unknown reasons, AWS was not able to support data encryption for the packets ASL sends. While this is not a major problem because no person information is sent over the network, it is still a concern as security should always be taken seriously. Investigation into how AWS can encrypt ASL packets should be continued and once resolved, implemented as soon as possible.

7.1.8 *More Use Case Features*

While ASL has be shown to work well with collaborative 3D applications, it also can be utilized as a collaborative data visualization tool. To fully realize this potential, ASL should incorporate more UI specific functionalities as well as specific functions to transfer spatial data information, such as GPS. By including such functionalities, ASL will become a more powerful and flexible library, capable of creating collaborative applications and tools that encompass a multitude of research fields. Specifically, some applications ASL could create if those features were incorporated are the ability to communicate firefighter's positions during a wildland fire to firefighting aircraft, better VR classroom integration, and collaborative data visualization and manipulations.

BIBLIOGRAPHY

[1] S. Mann, T. Furness, Y. Yuan, J. Iorio, and Z. Wang, "All Reality: Virtual, Augmented, Mixed (X), Mediated (X,Y), and Multimediated Reality," 2018, [Online]. Available: https://arxiv.org/abs/1804.08386v1.
[2] "A head-mounted three dimensional display." https://dl.acm.org/doi/pdf/10.1145/1476589.1476686 (accessed May 04, 2020).
[3] "Not-quite-live blog: panel discussion with John Carmack, Tim Sweeney, Johan Andersson - The Tech Report." https://techreport.com/review/25533/not-quite-live-blog-panel-discussion-with-john-carmack-tim-sweeney-johan-andersson/ (accessed May 04, 2020).
[4] "Valve to Demonstrate Prototype VR HMD and Talk Changes to Steam to 'Support and Promote VR Games' – Road to VR." https://www.roadtovr.com/vr-headset-valve-virtual-reality-steam/ (accessed May 04, 2020).
[5] "30 Minutes Inside Valve's Prototype Virtual Reality Headset: Owlchemy Labs Share Their Steam Dev Days Experience – Road to VR." https://www.roadtovr.com/hands-valves-virtual-reality-hmd-owlchemy-labs-share-steam-dev-days-experiences/ (accessed May 04, 2020).
[6] "62 Virtual Reality Statistics You Must Know in 2020: Adoption, Usage & Market Share - Financesonline.com." https://financesonline.com/virtual-reality-statistics/ (accessed May 04, 2020).
[7] M. Tanaya *et al.*, "A Framework for Analyzing AR/VR Collaborations An Initial Result," *2017 IEEE Int. Conf. Comput. Intell. Virtual Environ. Meas. Syst. Appl. CIVEMSA*, pp. 111–116, doi: 10.1109/CIVEMSA.2017.7995311.
[8] António Correia *et al.*, "Meta-theoretic Assumptions and Bibliometric Evidence Assessment on 3-D Virtual Worlds as Collaborative Learning Ecosystems - Learning & Technology Library (LearnTechLib)," *J. Virtual Worlds Res.*, vol. 7, no. 3, Accessed: May 06, 2020. [Online]. Available: https://www.learntechlib.org/p/178143/.
[9] "AR Obstacle Course | Cross Reality Collaboration Sandbox Research Group." https://sites.uw.edu/crcs/projects/ar-obstacle-course/ (accessed May 06, 2020).
[10] "Teaching Materials with AR/VR Collaborations | Cross Reality Collaboration Sandbox Research Group." https://sites.uw.edu/crcs/projects/teaching-materials-with-ar-vr-collaborations/ (accessed May 06, 2020).
[11] "What is Client-Server? Definition and FAQs | OmniSci." https://www.omnisci.com/technical-glossary/client-server (accessed May 09, 2020).
[12] "Advantages And Disadvantages of Client application server." https://www.esds.co.in/blog/advantages-and-disadvantages-of-client-application-server/ (accessed May 09, 2020).
[13] "Client-Server Model Definition." https://techterms.com/definition/client-server_model (accessed May 09, 2020).
[14] Xuefang Wu and Ru Gao, "The Design and Analysis of High Performance Online Game Server Concurrent Architecture," *2012 Int. Conf. Comput. Sci. Serv. Syst.*, Aug. 2012, doi: 10.1109/CSSS.2012.416.
[15] Cai Lan Zhou, Yu Kui Wang, and Sha Sha Li, "Design Of Server For Network Casual Games," *2011 Int. Conf. Comput. Sci. Serv. Syst. CSSS*, pp. 2193–2196, doi: 10.1109/CSSS.2011.5972196.

[16] Fábio Reis Cecin, Rafael de Oliveira Jannone, Cláudio Fernando Resin Geyer, Márcio Garcia Martins, and Jorge Luis Victoria Barbosa, "FreeMMG," *NetGames 04 Proc. 3rd ACM SIGCOMM Workshop Netw. Syst. Support Games*, p. 172, doi: 10.1145/1016540.1016567.
[17] Jared Jardine and Daniel Zappala, "A hybrid architecture for massively multiplayer online games," *NetGames 08 Proc. 7th ACM SIGCOMM Workshop Netw. Syst. Support Games*, pp. 60–65, doi: 10.1145/1517494.1517507.
[18] "Dedicated Game Server Hosting - Amazon GameLift - Amazon Web Services." https://aws.amazon.com/gamelift/ (accessed May 09, 2020).
[19] "Photon Unity 3D Networking Framework SDKs and Game Backend | Photon Engine." https://www.photonengine.com/en-US/PUN (accessed May 09, 2020).
[20] "GameSparks." https://www.gamesparks.com/ (accessed May 09, 2020).
[21] Erwin Peters *et al.*, "Design for Collaboration in Mixed Reality Technical Challenges and Solutions," *2016 8th Int. Conf. Games Virtual Worlds Serious Appl. VS-GAMES*, doi: 10.1109/VS-GAMES.2016.7590343.
[22] Hiroko Suzuki and Runhe Huang, "Virtual Real-time 3D Object Sharing for Supporting Distance Education and Training," *18th Int. Conf. Adv. Inf. Netw. Appl. 2004 AINA 2004*, doi: 10.1109/AINA.2004.1283950.
[23] Xia Luyao *et al.*, "Development and Application of Virtual Collaborative Experiment Technology Based on Unity Platform," *2018 IEEE Int. Conf. Saf. Prod. Informatiz. IICSPI*, doi: 10.1109/IICSPI.2018.8690340.
[24] Silvia Rueda, Pedro Morillo, and Juan M. Orduna, "A Peer-To-Peer Platform for Simulating Distributed Virtual Environments," *2007 Int. Conf. Parallel Distrib. Syst.*, doi: 10.1109/ICPADS.2007.4447819.
[25] "What are P2P (peer-to-peer) networks and what are they used for? | Digital Citizen." https://www.digitalcitizen.life/what-is-p2p-peer-to-peer (accessed May 09, 2020).
[26] Karsten Loesing and Guido Wirtz, "An Implementation of Reliable Group Communication based on the Peer-to-Peer Network JXTA," *3rd ACSIEEE Int. Conf. Comput. Syst. Appl. 2005*, doi: 10.1109/AICCSA.2005.1387076.
[27] Siu Man Lui and Sai Ho Kwok, "Interoperability of peer-to-peer file sharing protocols," *ACM SIGecom Exch.*, doi: 10.1145/844339.844350.
[28] Jin Li, "Peer-to-peer multimedia applications," *MM 06 Proc. 14th ACM Int. Conf. Multimed.*, pp. 3–6, doi: 10.1145/1180639.1180641.
[29] "What is peer-to-peer (P2P)? - Definition from WhatIs.com." https://searchnetworking.techtarget.com/definition/peer-to-peer (accessed May 10, 2020).
[30] Konstantin Pussep, Matthias Weinert, Aleksandra Kovacevic, and Ralf Steinmetz, "On NAT Traversal in Peer-to-Peer Applications," *2008 IEEE 17th Workshop Enabling Technol. Infrastruct. Collab. Enterp.*, doi: 10.1109/WETICE.2008.10.
[31] "Teach ICT - GCSE ICT - network topologies, network hardware, hubs, switches, routers, repeaters, bridges, modems, WAP, network cards." http://www.teach-ict.com/gcse_new/networks/peer_peer/miniweb/pg5.htm# (accessed May 10, 2020).
[32] "A Peer-To-Peer Architecture for Real-Time Distributed Visualization of 3DCollaborative Virtual Environments," *2009 13th IEEEACM Int. Symp. Distrib. Simul. Real Time Appl.*, doi: 10.1109/DS-RT.2009.21.
[33] Ignasi Barri, Concepció Roig, and Francesc Giné, "Distributing game instances in a hybrid client-server/P2P system to support MMORPG playability | SpringerLink," *Multimed. Tools Appl.*, pp. 2005–2029, doi: 10.1007/s11042-014-2389-0.

[34] Luong Quy Tho and Ha Quoc Trung, “P2P shared-caching model,” *SoICT 13 Proc. Fourth Symp. Inf. Commun. Technol.*, pp. 222–226, doi: 10.1145/2542050.2542090.
[35] “Introduction | Photon Engine.” https://doc.photonengine.com/en-us/pun/v1/demos-and-tutorials/pun-basics-tutorial/intro (accessed May 10, 2020).
[36] “Photon Unity Networking 2: Photon Network Class Reference.” https://doc-api.photonengine.com/en/pun/v2/class_photon_1_1_pun_1_1_photon_network.html#a5579a36ac089093b89baaf01c8dac519 (accessed May 10, 2020).
[37] “Summer 2018 ASL Documentation - API - ASL Namespace.” https://kelvinhsung.github.io/2019.CRCS-WebSite/Documentation/api/ASL/ (accessed May 10, 2020).
[38] Chen-chi Kuo, John Carter, and Ravindra Kuramkote, “MP-LOCKs: replacing H/W synchronization primitives with message passing,” *Proc. Fifth Int. Symp. High-Perform. Comput. Archit.*, Aug. 2002, doi: 10.1109/HPCA.1999.744381.
[39] Hongfei Fan, Hongming Zhu, Qin Liu, Yang Shi, and Chengzheng Sun, “Shared-locking for semantic conflict prevention in real-time collaborative programming,” *2017 IEEE 21st Int. Conf. Comput. Support. Coop. Work Des. CSCWD*, Oct. 2017, doi: 10.1109/CSCWD.2017.8066690.
[40] “Best VR Game Engine Software in 2020 | G2.” https://www.g2.com/categories/vr-game-engine (accessed May 16, 2020).
[41] “Unity XR platform updates - Unity Technologies Blog.” https://blogs.unity3d.com/2020/01/24/unity-xr-platform-updates/ (accessed May 16, 2020).
[42] “Online and in-person courses & training in 2D, 3D, AR, & VR development | E-Learning.” https://unity.com/learn (accessed May 16, 2020).
[43] “Unity - Manual: Unity User Manual.” https://docs.unity3d.com/Manual/index.html (accessed May 16, 2020).
[44] “Projects | Cross Reality Collaboration Sandbox Research Group.” https://sites.uw.edu/crcs/projects/ (accessed May 16, 2020).
[45] Gregory Smith, “ASL Documentation.” https://uwb-arsandbox.github.io/ASL_Master/ASLDocumentation/Help/html/16d360d9-1284-4cae-b7c9-c114d2c074ee.htm (accessed May 17, 2020).
[46] “Unity - Scripting API: GameObject.” https://docs.unity3d.com/ScriptReference/GameObject.html (accessed May 17, 2020).
[47] “AR Collaboration | Cross Reality Collaboration Sandbox Research Group.” https://sites.uw.edu/crcs/ar-collaboration/ (accessed May 17, 2020).
[48] “ARCore.” https://developers.google.com/ar/ (accessed Jun. 02, 2020).
[49] “Unity - Scripting API: Transform.localPosition.” https://docs.unity3d.com/ScriptReference/Transform-localPosition.html (accessed Jun. 05, 2020).
[50] “Unity - Scripting API: Transform.position.” https://docs.unity3d.com/ScriptReference/Transform-position.html (accessed Jun. 05, 2020).
[51] “Unity - Manual: Scenes.” https://docs.unity3d.com/Manual/CreatingScenes.html (accessed May 17, 2020).
[52] “GameSparks Case Study.” https://aws.amazon.com/solutions/case-studies/gamesparks/ (accessed May 24, 2020).
[53] “GameSparks - Learn.” https://docs.gamesparks.com/ (accessed May 24, 2020).

[54] “GameSparks Self-Service Pricing FAQs.” https://www.gamesparks.com/pricing/faqs/ (accessed May 24, 2020).
[55] “Blog | GameSparks.” https://www.gamesparks.com/blog/ (accessed May 17, 2020).
[56] “Amazon confirms that it has acquired GameSparks | TechCrunch.” https://techcrunch.com/2018/03/05/amazon-confirms-that-it-has-acquired-gamesparks/ (accessed May 17, 2020).
[57] “AWS Lambda – Serverless Compute - Amazon Web Services.” https://aws.amazon.com/lambda/ (accessed May 25, 2020).
[58] “MixedRealityToolkit-Unity.” https://github.com/Microsoft/MixedRealityToolkit-Unity (accessed Jun. 01, 2020).
[59] “Microsoft Azure PlayFab.” https://playfab.com/ (accessed Jun. 02, 2020).
[60] “Brain Cloud.” https://getbraincloud.com/ (accessed Jun. 02, 2020).

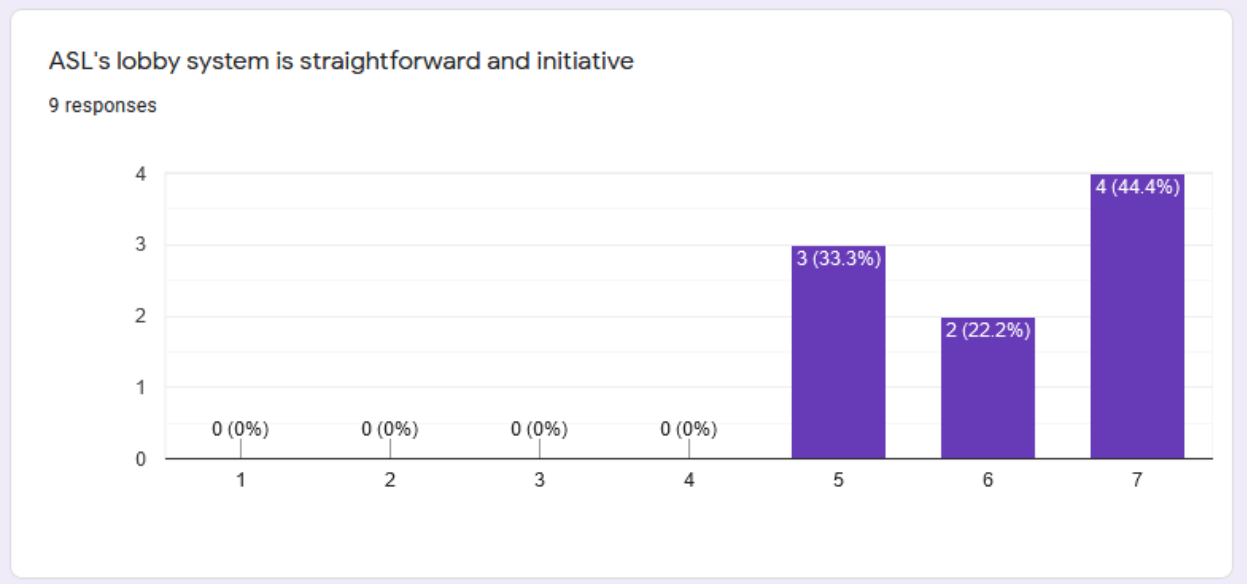
APPENDIX A

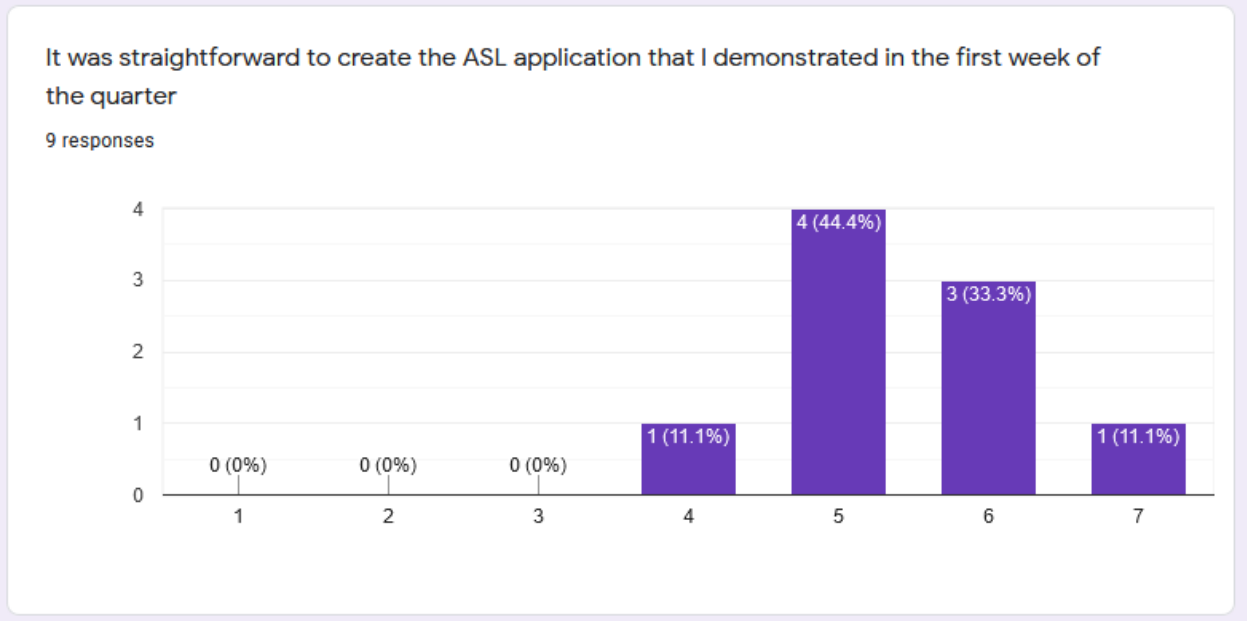
The following is a table containing the list of acronyms used throughout this paper and their meaning.

Acronym	**Meaning**
VR	Virtual Reality
AR	Augmented Reality
CRCS	Cross Reality Collaboration Sandbox
ASL	Augmented Space Library
SDK	Software Development Kit
P2P	Peer-to-Peer
PUN	Photon Unity Network
AWS	Amazon Web Services

APPENDIX B

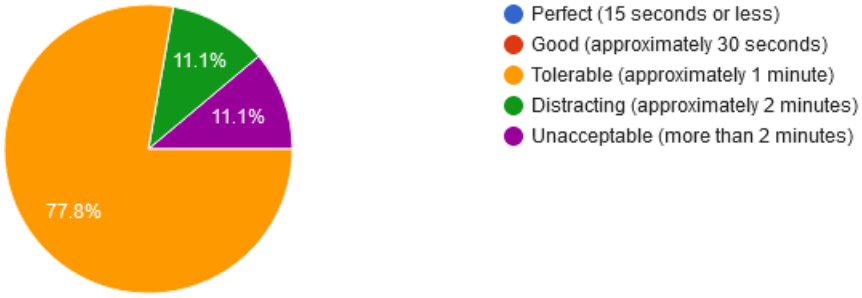
The following is the complete survey phase two users were asked to fill out and their answers. This survey and its responses form the information found in Table 6.1. The number of responses each question has can be seen directly below its question. To aid in readability, user responses to short answer questions are copied independently.





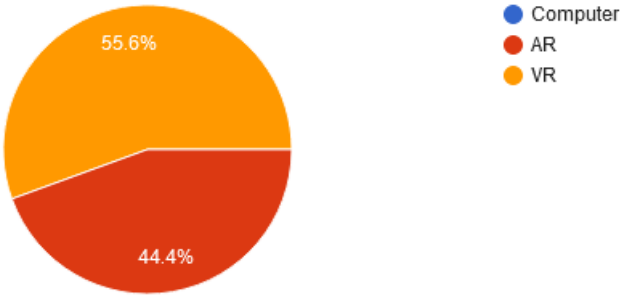
When debugging my application, after I fix my source code, the time it took to start the application to test my fix was on average

9 responses



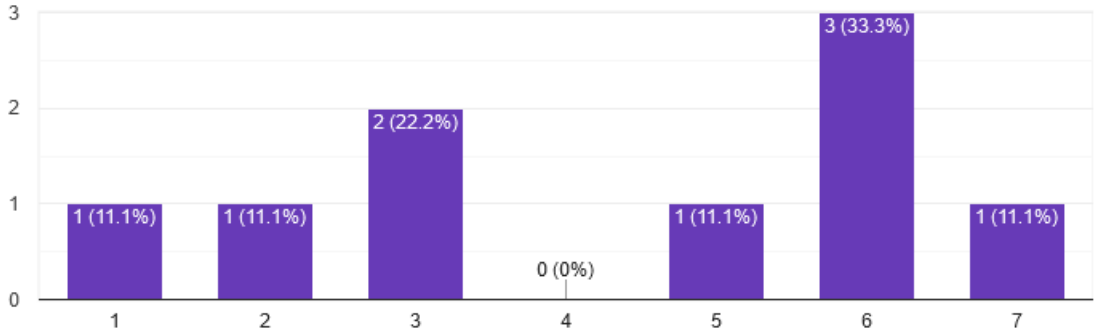
In regards to the previous question, what platform were you primarily testing on?

9 responses



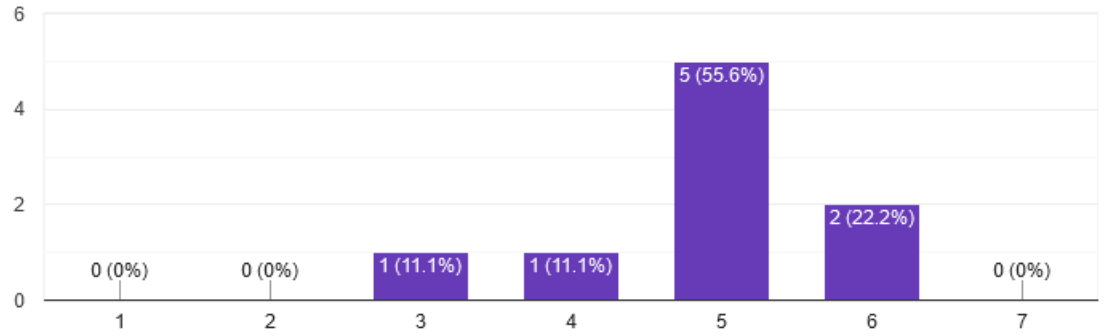
When something was not working with ASL, it was easy to debug

9 responses



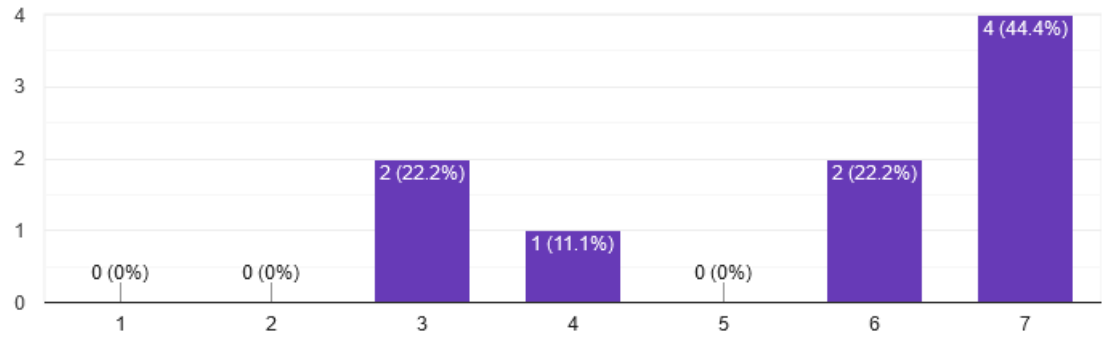
ASL's documentation did a good job of explaining how ASL functions work

9 responses



ASL's simple demos and documentation examples allowed me to rapidly understand how to use ASL

9 responses



Is there anything you wish ASL had? E.g., a function that did ____, a player movement class, etc. Please be as specific as possible

9 responses

Maybe a function that applies physics to an object

After InstantiateASLObject, being able to get a reference to the object that was instantiated.

N/A

A way to test a game (AR) without having to connect to ASL AWS servers, it's really annoying to test something but then have to wait for ASL servers to start up before connecting. Perhaps a offline feature?

Object manipulation with VR

a function(s) that simply update any type of variables for all users

Some sort of built-in integration for physics, even just synchronized gravity would be very useful.

At first, I wasn't sure if it made sense to have a "Player Movement class" because different projects might have very different needs in terms of this functionality. But the more I thought about it, the more I thought that it might make sense to have some sort of "basic" VR movement controller set up as a demo so that people could try it and build off of it if it fit with what they were doing. This is mostly because figuring out some of the nuances of setting up a player controller can take a lot of time to figure out. It might also be useful to add or expand on some of the functionality for managing ASL objects. For example, it might be good to have some sort of ASLObjectManager class that all ASL objects in the scene are registered with which can then be used to easily apply operations over a whole group.

I think it's great if there is an ability to turn on/off ASL visualization (surface, particle)

Is there anything you would improve about ASL? E.g., better documentation, more examples, better VR integration, etc. Please be as specific as possible

9 responses

Better VR integration

Better documentation on general object creation/deletion, other general pages not relating to a specific function possibly?

better VR integration. Additionally, MRTK is difficult to use

An example showing how you can use ASL's SendFloatArray function to send a Vector3 so that a player object can be moved using rigidbody physics with the send out vector3

Standard assets to use with MRTK like picking up objects with VR

more examples

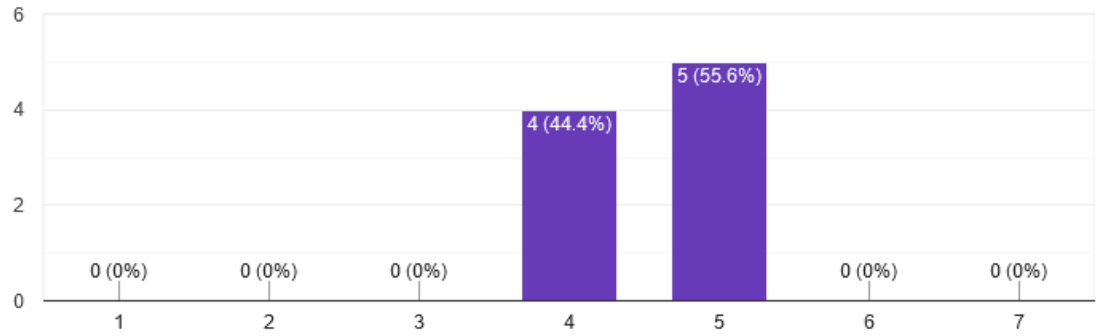
The documentation could use a bit of work. I found that when I went looking for functionality that I knew it had, it was still hard to find without checking the examples. Also, the flow of object creation and referencing could be simpler, without the need of a callback function. If the InstantiateASLObject returned a reference to the created object, it would be much simpler for those who don't need the callback.

I have a couple of fairly minor suggestions. First, the documentation isn't bad, it just isn't always as clear as it could be. It is fairly comprehensive in covering what exists, but it often doesn't explain much about how/why a class is used (especially for the "less major" things like some of the delegates). I think that even just a couple more sentences on some of the descriptions of the classes might help a lot. Reading the method and data descriptions in the class documentation can be very helpful, but not always as useful as exploring the demos or just trying things out yourself. It would also be great if there was a section outside the class descriptions that gave some use case scenarios or explained some of details of the demos. This brings me to point 2, the examples that are in the library are helpful but rather limited. This is totally reasonable given the constraints, but it might be good to have a few more that are a more involved. Overall though, the demos are very helpful. Third, VR integration isn't necessarily a problem, but it can be difficult to sift through how everything should be setup. I think that has a lot more to do with the MRTK than ASL, though. There were a bunch of things that were either wrong or outdated in the MRTK documentation which added to the confusion of using it. Now that the teams are compiling some of the things that we learned over the course of using these libraries, I think it might be good to include some of that in the documentation (or a link to it from the documentation). Again, those issues aren't faults of ASL, it just might help future users to have easy access to that information. One last thing, it would be GREAT if the lobby and quick connect had VR support because it is kind of clunky trying to start a session while juggling controllers, headset, and keyboard/mouse input.

For method sendfloatarray(), I think there can be a tag variable in both sendfloatArray() and callback method, so that user can separate the switch/case and the data

If I am to build another online multi-person application, I am likely to use ASL for the project (either working with or not with Kelvin, for personal or school)

9 responses



Is there anything else you would like to tell us?

4 responses

Server issues/ASL scaling up can be annoying at times, but probably not much that can be done on your end.

Only reason not to use ASL is because it cost Kelvin money

Overall, it felt like a very feature-rich start. I stand by the comment I made above about simplifying the flow of object manipulation. If returning a reference is not possible, even returning the ID of the object to then be retrieved through lookup in the ASLHelper.m_ASLObjects without the convolution of the callback in situations where it is unneeded.

A lot of my ratings are somewhat neutral and might not be as helpful because of it, so I wanted to clarify a little bit. I think my previous comments hopefully helped explain that the documentation isn't bad, it just could use a bit more information.

In terms of the lobby system, I think that using it as a User is quite intuitive but, getting it set up correctly could be a bit tricky, especially with VR and MRTK.

I marked the debugging question as a 3 but I really debated whether that was fair. I honestly think it has more to do with some of the other factors than ASL itself. For example, we can't use the Visual Studio debugger with MRTK because it can't build the project. That isn't ASL's fault, but it does add a layer of difficulty to debugging. One thing that I will add about debugging ASL is that the ASLObject class is a bit monolithic. I completely understand why it would be that way but it can be daunting to try to track something down in it. Generally speaking, I didn't actually have to debug that class but, a few times when I was debugging other things, I read through a bunch of it to try to make sure there wasn't a problem with it. In those cases, it could be challenging to find what I was looking for or know where to start. That being said, big classes happen and it is fair to say that sometimes it is necessary to have them that way.

APPENDIX C

The following is screen shots of the various documentation ASL offers. All documentation can be found here: https://uwb-arsandbox.github.io/ASL_Master/ASLDocumentation/Help/index.html

ASLObject Class

ASLObject: ASLObject Partial Class containing all of the functions and variables relating to server actions - actions that affect all players instead of just a single player. use this class to communicate object information to other players. If you are looking for where to create an object for all users at runtime, check out InstanitateASLObject(String, Vector3, Quaternion) and it's other variations
ASLObject_LocalFunctions: ASLObject Partial Class containing all of the functions and variables relating to local actions - actions that affect a single player instead of all players

▷ **Inheritance Hierarchy**

Namespace: ASL
Assembly: Assembly-CSharp (in Assembly-CSharp.dll) Version: 0.0.0.0

▷ **Syntax**

The ASLObject type exposes the following members.

▷ **Constructors**
▷ **Properties**
▷ **Methods**
▷ **Fields**
▷ **See Also**

◢ Methods

	SendAndIncrementLocalPosition	Sends and adds to the local transform of this object for all users
	SendAndIncrementLocalRotation	Sends and adds to the local rotation of this object for all users
	SendAndIncrementLocalScale	Send and add to the local scale of this object for all users
	SendAndIncrementWorldPosition	Sends and adds to the world transform of this object for all users
	SendAndIncrementWorldRotation	Sends and adds to the world rotation of this object for all users

Syntax

C# VB C++ F#

```
public void SendAndIncrementLocalPosition(
        Nullable<Vector3> _localPosition
)
```

Parameters

_localPosition
Type: System.Nullable<**Vector3**>
The value to be added to the local position of this object

Examples

```
void SomeFunction()
{
    gameobject.GetComponent<ASL.ASLObject>().SendAndSetClaim(() =>
    {
        gameobject.GetComponent<ASL.ASLObject>().SendAndIncrementLocalPosition(new Vector3(1, 2, 5));
    });
}
```

APPENDIX D

The following are screenshots of snippets of the code that makes up the ASLObject script, or the script users will interact with the most. To see all the code involved with ASL, see https://github.com/UWB-ARSandbox/ASL_Master

```
/// <summary> Claims an object for the user until someone steals it or the passe ...
74 references
public void SendAndSetClaim(ClaimCallback callback, int claimTimeOut = 1000, bool resetClaimTimeout = true)
{
    if (Time.timeScale == 0) { return; } //Time scale is set to 0 when not all ASL objects have been assigned an id yet - once all assigned, time will resume
    if (!m_Mine) //If we already own the object, don't send anything and instead call our callback right away
    {
        if (!m_OutStandingClaims)
        {
            m_OutStandingClaims = true;
            RTMessage message = GameLiftManager.GetInstance().CreateRTMessage(GameLiftManager.OpCode.Claim, Encoding.ASCII.GetBytes(m_Id));
            GameLiftManager.GetInstance().m_Client.SendMessage(message);
        }
        m_ClaimCallback += callback;
        m_OutstandingClaimCallbackCount++;
    }
    else
    {
        callback();
    }
    if (resetClaimTimeout)
    {
        m_ClaimReleaseTimer = 0; //Reset release timer
        m_ClaimTime = claimTimeOut; //Reset claim length
    }
}
```

```
/// <summary> Send and sets this objects color for the user who called this func ...
12 references
public void SendAndSetObjectColor(Color _myColor, Color _opponentsColor)
{
    if (m_Mine)
    {
        byte[] id = Encoding.ASCII.GetBytes(m_Id);
        byte[] myColor = GameLiftManager.GetInstance().ConvertVector4ToByteArray(_myColor);
        byte[] opponentsColor = GameLiftManager.GetInstance().ConvertVector4ToByteArray(_opponentsColor);
        byte[] sender = GameLiftManager.GetInstance().ConvertIntToByteArray(GameLiftManager.GetInstance().m_PeerId);
        byte[] payload = GameLiftManager.GetInstance().CombineByteArrays(id, myColor, opponentsColor, sender);

        RTMessage message = GameLiftManager.GetInstance().CreateRTMessage(GameLiftManager.OpCode.SetObjectColor, payload);
        GameLiftManager.GetInstance().m_Client.SendMessage(message);

    }
}
```

```
/// <summary> Deletes this object for all users
5 references
public void DeleteObject()
{
    if (gameObject && m_Mine)
    {
        byte[] id = Encoding.ASCII.GetBytes(m_Id);
        RTMessage message = GameLiftManager.GetInstance().CreateRTMessage(GameLiftManager.OpCode.DeleteObject, id);
        GameLiftManager.GetInstance().m_Client.SendMessage(message);
    }
}
```

```
/// <summary> Sends and sets the local transform for this object for all users
11 references
public void SendAndSetLocalPosition(Vector3? _localPosition)
{
    if (m_Mine) //Can only send a transform if we own the object
    {
        if (_localPosition.HasValue)
        {
            byte[] id = Encoding.ASCII.GetBytes(m_Id);
            byte[] localPosition = GameLiftManager.GetInstance().ConvertVector3ToByteArray(new Vector3(_localPosition.Value.x, _localPosition.Value.y, _localPosition.Value.z));
            byte[] payload = GameLiftManager.GetInstance().CombineByteArrays(id, localPosition);

            RTMessage message = GameLiftManager.GetInstance().CreateRTMessage(GameLiftManager.OpCode.SetLocalPosition, payload);
            GameLiftManager.GetInstance().m_Client.SendMessage(message);

        }
        else //Send my position as is, not a new position as there was no new position passed in.
        {
            byte[] id = Encoding.ASCII.GetBytes(m_Id);
            byte[] localPosition = GameLiftManager.GetInstance().ConvertVector3ToByteArray(new Vector3(transform.localPosition.x, transform.localPosition.y, transform.localPosition.z));
            byte[] payload = GameLiftManager.GetInstance().CombineByteArrays(id, localPosition);

            RTMessage message = GameLiftManager.GetInstance().CreateRTMessage(GameLiftManager.OpCode.SetLocalPosition, payload);
            GameLiftManager.GetInstance().m_Client.SendMessage(message);
        }
    }
}
```

```
/// <summary> Send and set  up to x float value(s). The float value(s) will then ...
21 references
public void SendFloatArray(float[] _f)
{
    if (m_Mine) //Can only send a transform if we own the object
    {
        byte[] id = Encoding.ASCII.GetBytes(m_Id);
        byte[] floats = GameLiftManager.GetInstance().ConvertFloatArrayToByteArray(_f);
        byte[] payload = GameLiftManager.GetInstance().CombineByteArrays(id, floats);

        RTMessage message = GameLiftManager.GetInstance().CreateRTMessage(GameLiftManager.OpCode.SendFloats, payload);
        GameLiftManager.GetInstance().m_Client.SendMessage(message);


    }
    else
    {
        Debug.Log("Cannot send floats - do not have ownership of object");
    }
}
```