The following is taken from Greg Smith's Thesis: Augmented Space Library 2: A Network Infrastructure for Collaborative Cross Reality Applications. If you wish to read the entire thesis, you may find it here: https://drive.google.com/drive/u/2/folders/1p1tudLBxWkj66fHlUXVCMgHCZciuA64u

## 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.