**RESEARCH PAPER**

**Scene Manager for Real-Time**

**Concurrent Scene Manipulation**

**2018**

**By James Schmidt**

**Version History**

**Version 1.1**

Purpose

The purpose of this research is to aid programmers by giving them the ability to concurrently manipulate a scene in real-time. Doing so will make working on a scene easier and more efficient by allowing for fast iterations, near zero merge conflicts, and promoting collaboration amongst developers/designers.

Introduction

With the ever-growing demand for gaming studios to publish their games on time, developers and designers have been left to work harder to get changes submitted before their deadline. Is this due to a miscalculation of time on the project managers side? Or is the process in which developers and designers submit their changes outdated? Say a developer/designer wants to make changes to a scene in their game. They have a few choices on how to proceed. They could checkout the scene and assets from the build to make changes and submit. Doing so, however, locks anyone else out from updating that scene until it’s checked back in. They could go the naïve way and change the scene in their editor and submit their changes in small chunks to make sure the master is always up to date. This however, will lead to a ton of merge conflicts between other developers/designers who are also working on the same scene. Lastly, they could create a new branch on their source control and make changes to the scene there. Then submit their changes to the server and create a pull request to have it merged in with the master.

So far, every one of these choices is very time consuming. Every change submitted requires approval before it can be accepted to the master branch. Those who don’t take the time to check a change before approval risk having to revert to a previous build which could potentially destroy weeks of work. To combat a lot of this, many game companies utilize collaboration tools to keep their developers/designers on the same page. Collaborating saves time when multiple people can complete their tasks side by side without worry of merge issues or checking individual work for approval. So, if collaboration can save developers/designers time, then why not add collaboration into an editor/engine?

The Engine Makes the Engineer

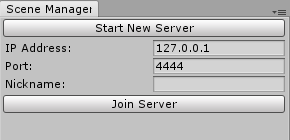
Thus, with that question in mind, I embarked on a journey to create an editor/engine than can utilize proper collaboration. My journey hit an abrupt stop however after I realized that I would first need an existing engine to make a collaborative editor. At this point my choices were: Unity, Unreal, Lumberyard, or make a custom engine. Luckily, I had some prior experience with Unity doing a project for AI, so I decided to go with it. With the engine picked, I started to make plans on how to implement the collaboration.

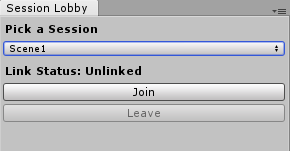
Functionality

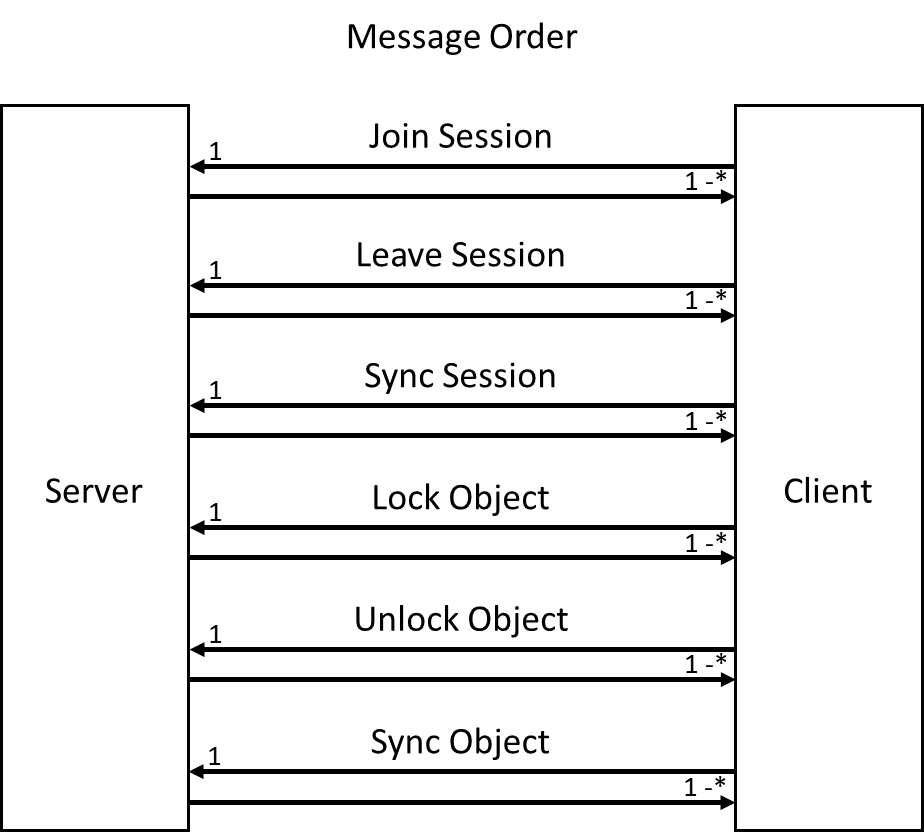
It was at this time that I remembered the “Shared World Shooter” (1) talk from GDC 2015 given by Justin Truman at Bungie. His talk mentioned how they managed to utilize a hybrid of dedicated servers and host swapping peer-to-peer architecture to make their server footprint small while giving their game states low latency and seamless updates. Though the peer-to-peer seemed a little overkill for my project, I did however take away how they utilized their server-based architecture to handle managing states. Coming to the idea of making a host to store the master scene state while other clients can send manipulations to that state lead me to my next issue; what happens when two clients want to manipulate the same object?

That one ended up being a little easier after relating it to mutex locking from multithreading. All I’d have to do is store a locked list generated by the objects being manipulated and the client that’s using them. Objects that are locked can’t be manipulated by other clients, thus relinquishing the issue. Also, since there’s only ever one master state, the client that gets their request to lock an object in first becomes the owner of that object. Giving only one client access to update an object at a time successfully allows the master scene state to update without worry of collisions between object ownership. With those ideas in hand I pleasantly pushed forward and drafted some helpful diagrams to fine-tune the functionality.

Functional Diagrams

 First things first, I need to address the need to spawn a server to store the master scene state. The starting display should display a **Start New Server** button, which when pushed, should spawn the editor as a host server for other client editors to link with. This button will change to **Stop Server** if a server was spawned. If a user wants to instead join an existing host server, then all they will need to do is add in the IP address, Port, and create a nickname of the server that they wish to join. Once they click the **Join Server** button, they will be redirected to the server lobby.

 Next is the server’s session lobby. Here, the client will be able to choose from a drop down of sessions and click the **Join** button to join. If a session hasn’t been spawned yet, then the server will spawn it on the first client join. Also, if there are no more clients in a session, then the server will save the session and close it. If the client wants to leave a session, then they can click on the **Leave** button. Subsequently, if the client changes the selected session, it will automatically kick them out of their current session.

Lastly, the communication between the server and client needs to deal with messages. But when should they be handled? I figured the **Join Session** message should be the first to be handles since it takes care of adding a client to a session and then sending that client info to other clients. Naturally, then to follow would be the **Leave Session** message. Knowing if a client has left will also help when unlocking any objects that they had been using. Next, I decided to handle the **Sync Session** message. This message is sent right after a client joins a session so that they can grab all the existing objects in the session to duplicate and pair ID’s for syncing. After that, the **Lock Object** message should handle locking any objects that are being handled by clients. Following that, the **Unlock Object** message should handle unlocking objects that are no longer selected for manipulation. Then ultimately, the **Sync Object** message should be handled so that the server’s scene state can updated the object and propagate the change to all clients. An object will only be able to sync if the owning client was the one who sent the sync message.

Where the Fun Began

Finally, after all of that planning it was time to start implementing, and by implementing, I mean pulling my hair out. Unfortunately, Unity wasn’t built with access to some of the background functionality that I was hoping for. Something as simple as figuring out if a property value was changed and where it was changed from in the Inspector window required many days of searching through forums just to find that I could grab the info from the Undo class’s “postprocessModifications” event. Yet, not every change is guaranteed to be handled there since there’s a totally different function, that when called, will apply property changes without sending an undo. But I digress, the Undo event worked well for what I needed, especially since I needed a way to post a change log.

After finishing the change log, I focused next on creating a Session class. A Session needs to store a corresponding scene, list of client info, and the updated locked list for local querying. The corresponding scene is helpful for grabbing objects from a scene, needed for syncing sessions. Being able to add and remove clients to the client info list allows clients to know who’s logged into their session. Having a local locked list that gets updated by the server helps clients know which objects they can or can’t select before they even get the chance to manipulate any properties. However, GameObjects can still be selected and changed from within the hierarchy window. This is a feature of Unity’s editor so that users can’t be locked out of selecting GameObjects to see their components. Luckily, thanks to the locked list, even if the client changes the object and sends a sync object message to the server, the server will reject the change since they aren’t the object’s owner. The server will then send a sync object message back to the client with the true property settings from the master scene state.

The Headache of Properly Syncing Objects: Part 1

Speaking of sync objects, dealing with them was next on the list. I first started off by testing if components could be copied from one object to another. Unfortunately, there was no explicit way of doing this. I could get a component from one GameObject, but there was no way to correlate it to another component from a different GameObject. Everything (that is serialized) in the engine has a generated InstanceID that is editor specific. Which means no two editors will be guaranteed to have the same ID’s for their GameObjects. I then concluded that the only way a synced ID would work would be store every object with a NetworkID. Only to find out the next day that Unity has such a thing for its runtime networking, but it itself is a component that I’d have to attach to a GameObject and it’s only usable during runtime. So, after some time debating, I decided to just use the O(n2) approach of syncing components after finding a forum post that mentioned an internal function call that I could make from UnityEditorInternal to copy and paste components. Luckily, I’ll end up coming back to this issue later on with a more practical implementation.

Graphical User Interface (GUI)

Once I was able to sync changes between existing objects, I was ready to tackle the GUI. Implementing editor window logic, buttons, and text fields were actually pretty simple thanks to the tutorials on Unity’s website. However, more advanced features like popup selections seemed to be in a totally different place than everything else. Enabling and disabling buttons also felt weird given that it was using a state machine, however it was a minor inconvenience given that a GUI done this way makes for less bloated GUI elements. One thing I wish I would have spent more time figuring out though is how to change the offsets and spacing of certain GUI elements. The default settings make the layout look too spaced out and hard to type into if the window shrinks.

The Headache of Properly Syncing Objects: Part 2

Now with all the GUI elements in place, it was time to revisit the issue with syncing objects. So, I asked myself, how does the editor handle accessing object properties? The editor needs to iterate over all objects and properties to serialize each of them into a scene file. Hopefully there must be a way to access individual properties like they do. Luckily there was. After finding a forum talking about property serialization, I was able to conclude that I could just grab the individual property changes and sync them to their corresponding object. Then later I realized that the Undo class from before that gave me the property modifications could also be utilized in this way. If I grab the modifications made by the Undo class and then send that property path to be serialized I could then in turn apply that property modification to any given object. It really was quite the eureka moment. The only thing left then was to find a way to link object ID’s between given scenes. I found that the easiest way to solve that problem was to just destroy any pre-existing objects from the client session and just push all the objects from the server into the client session. Doing so made it easy to link object ID’s since I was able to map the newly instantiated cloned object ID to the corresponding server object ID. However, is there a better way than just deleting everything from an existing session? Why not just make an empty session first?

The Empty Session

My first attempt at making an empty session started with having the editor handle making an empty scene and linking that to the session. Unfortunately, the only way to make an empty scene from scratch in the editor is to call the NewScene function from the EditorSceneManager. This seemed to work well at first, until I couldn’t find a way to change the name of the Untitled scene. Not to mention that any time the client wanted to save, the SaveAs window would pop up and be annoying. I ended up compromising and just made an empty scene file called Session in the asset folder to be imported when needed. Pro’s to this means that a client could save their session and keep it on hand in case they need a local backup. Con’s to this means still having to wipe the scene of all objects if the scene was saved with objects.

No Networking Niceties

What’s not to love about networking? Unity had a lot of networking infrastructure built into it to help developers. Only catch is, majority of that infrastructure is for runtime applications. Not to mention how it’s almost impossible to find forum posts for help on the subject when every query brings up the runtime implementations. Fortunately, I was able to finally find a post that talked about utilizing the given networking infrastructure behind the scenes bypassing the runtime requirement. Unfortunately, there wasn’t enough time left before the project submission to get it properly implemented. I was however, able to create the server/client infrastructure for easy conversion with networking for future iterations. That being said, my submitted project is still able to sync sessions between the local server and client created within the Unity editor.

Conclusion

In the end, the project was a lot of fun. I learned a ton about Unity in the process and even learned different ways of implementation things. I also learned that sometimes it isn’t always easy to develop in a code base that doesn’t have the functionality that you’d like already built into it; or at least readily accessible to developers. Things like having built-in networkID’s into GameObjects instead of using a component and allowing object syncing between editors would be a few of the changes I’d like to see. That’s why I plan on taking what I’ve learned here and adding to it through future projects in hopes to develop a truly collaborative editor. Hopefully this research lead to a product that will not only help game studios with faster iterations but also shield developers from the dreaded “crunch time”.

Resources

1. **Shared World Shooter:** Justin Truman from Bungie (GDC 2015) - <https://www.gdcvault.com/play/1022247/Shared-World-Shooter-Destiny-s>