Blender Live Link Prototype Architecture v0

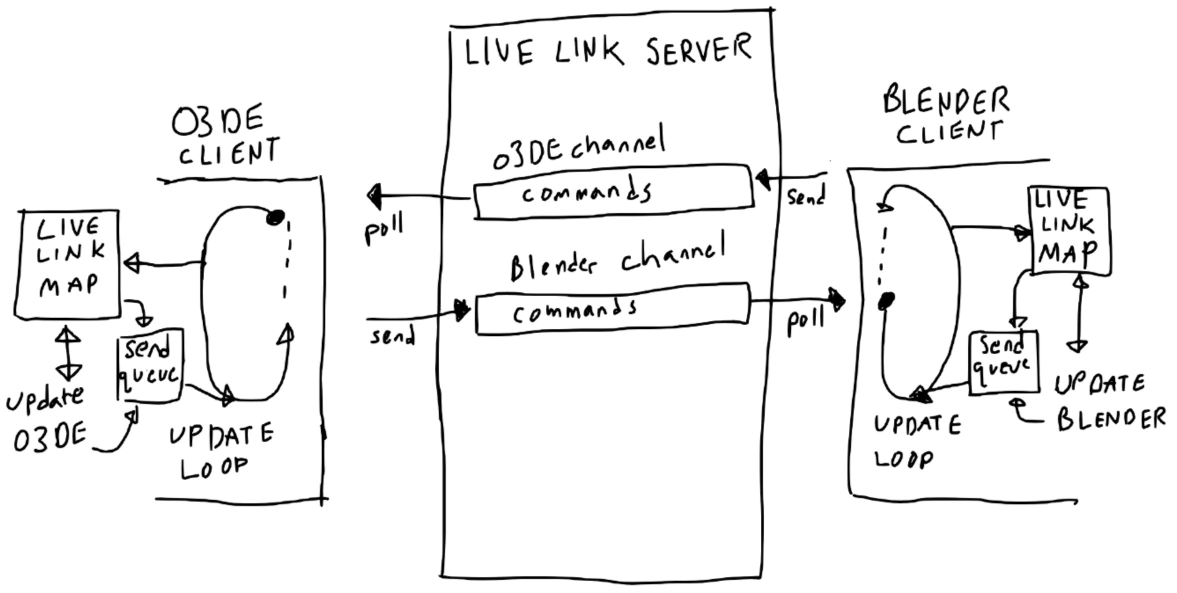
Purpose

This document outlines the architecture and implementation for the prototype live linking server between O3DE and Blender. It will also discuss the strengths and weaknesses of the approach used, and possible extensions or revisions for full integration with DCCsi. A followup review is recommended to determine how to best integrate multiple DCC tools in a live linking ecosystem.

This prototype is scoped to two workflows: whitebox modeling, and entity positioning.

Youtube video: https://www.youtube.com/watch?v=lONmqaW7EJE
Github Link: https://github.com/tkothadev/o3de-dccsi-sandbox/tree/tkothadev/blender-live-link

Overview

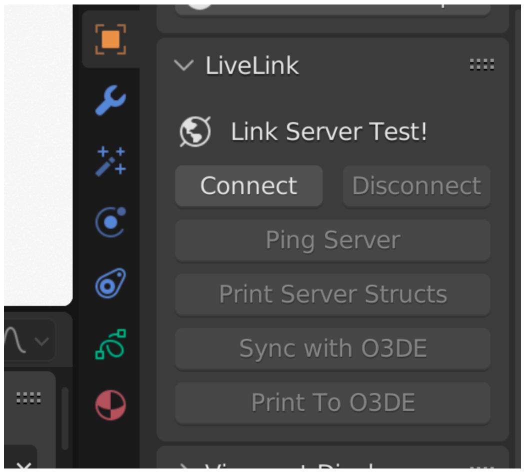


The Live Link Server is the central piece enabling communication for all applications. It is a separate python script that runs as its own process, with open ports matching the number of expected apps to connect to it (i.e. O3DE and Blender).

The server holds a list of channels, which are queues (in this case just regular python lists) mapped to the name of an application. The convention is that the app matching the name of the channel pops the channel for messages, while every other app pushes messages to the channel. This rule enables duplex messaging between apps, since each app gets a dedicated channel.

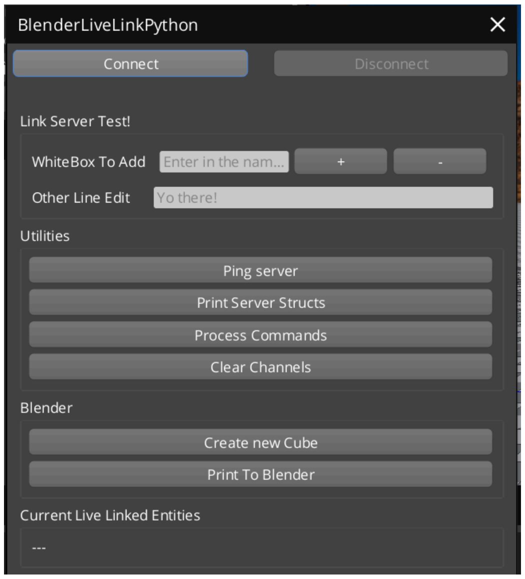
The python package `multiprocessing` was used to create a network client capable of sending python lists across the network. These lists facilitated the command structure. The structure was simply `[op_code, args...]`. Once the command is in the channel queue, the recipient client is responsible for taking it off the queue, and parsing the command. Based on that parsing, it runs the relevant function mapped to that command with supplied arguments, or ignores it if none is found.

Each client must first formally connect to the server, via a known IP address and Port. As of this prototype, that was hardcoded (incidentally, this also meant that Live Linking worked across computers in a local network).



BLENDER PANEL

The UI is simpler here, since we use blender’s existing features to facilitate live linking.



O3DE PANEL

The UI has slightly more than the Blender Panel, because it contains some explicit commands (such as creating new objects, or removing objects from the list).
In the future, this should be improved such that the interface between the two apps are symmetric.

Live Link Mapping

On each client, there is a data structure called the live link mapping. This mapping is a dictionary which tracks a list of entities designated for live linking, and all attributes that need to be synced for a given workflow.

For example, with a positioning workflow, only the position, rotation, and scale attributes are tracked for each entity.
For a whitebox workflow, the transform attributes are tracked, as well as the object mesh.

Each attribute, except for Mesh, had a designated `previousValue`. If this did not match the attribute’s current value, that was known as a discrepancy. If a discrepancy is found during a client’s update loop, the discrepancy is corrected for the current value in the client, and the update for that attribute is sent to the network. All other clients receiving this update must apply the update to their own representation, and prevent any follow up commands, otherwise a feedback loop occurs.

All commands from each client are not sent immediately. First they are stored in a send queue. Then at a designated time in the client’s update loop, first we process a command from the server’s queue, then all commands stored in the client’s send queue are sent to the server, in an appropriate channel. This prevents messages from getting mixed up (which happens if we try to communicate with the server from multiple spots in the client code).

This syncing mechanism is what makes bi-directional editing possible. The reason this works in a peer-to-peer fashion is because typically one user is operating both systems. This means changes often happen one at a time from either end of the network. It is very rare for changes to happen simultaneously, and any discrepancy introduced is quickly swallowed by the next update.

The Live Link Mapping also either maintains, or is connected to, the list of currently linked entities. In O3DE, this is a list maintained with the Live Link Mapping, while in Blender, it exists as a distinct collection called “O3DE”. This list makes it possible to select a portion of the scene for live linking in a flexible and intuitive manner. All that has to be done is drag and drop an entity into the collection and it will be synced. This allows us to have a staging area for synced entities, and anytime we don’t want items to be synced, they can just be moved out of the collection.

Mesh Syncing

Meshes are the main synchronization edge case. Currently this only works from Blender to O3DE, but the principle is ideally the same the other way around, it’s just a matter of implementing it.

Instead of syncing every time something changes in the mesh, we instead listen for a specific event to occur: We wait for the user to exit Edit mode from Blender. From there we check what mesh was edited, gather its vertices and faces, and send that across the network to O3DE. O3DE will then reconstruct the whitebox entity with the mesh it received.

This is a very crude method, but is effective for smaller meshes for rapid prototyping. Although it is not immediate, seeing the updates after just exiting edit mode still feels pretty good from a UX standpoint. However this will fall apart if dense meshes with complex geometry is being edited (exact threshold has not been determined yet).

The biggest room for improvement is to find a way to send smaller mesh changes across the network, such that both clients converge on the correct model change. That is a topic for further research. Later on we want to also support changes in material and texture, and live linking such rendering setups from Blender to O3DE (possibly a full translation of EEVEE to Atom).

Strengths

The command structure is very simple, and enables us to handle a diverse range of use cases, provided the schema is fleshed out. It’s also possible to cleanly handle different DCC tools, by segregating commands by channel name, each name being a distinct DCC tool. This lets us handle DCC specific edge cases consistently. By extension, the Live Link server is very simple as well, and can operate across multiple machines. The server itself can be DCC agnostic.

The Live Link mapping allows us to react to each entity as needed for the given workflow. The staging area provides a natural interface for syncing, and makes feedback very quick and responsive (especially once network delays are improved).

Overall, the main strength is that the architecture shows promise for scalability.

Improvements

Currently the setup is restricted to python API only. If we want live linking to work in game runtime scenarios, we will need a language agnostic scheme so that any client can communicate with the server.

Also, the client logic for handling attributes is currently hard-coded. As such I would like to spend time codifying the concept of ‘workflow’, such that it is easy to define new workflows. In this manner, positioning, whiteboxing, animating, and texturing can be distinct workflow objects, with filtering for specific entities for syncing, and customized logic to facilitate proper UX in each workflow. The exact mapping from O3DE entities/components to the DCC attributes needs to be fleshed out as well.

From an architectural standpoint, this is the timeline I’m looking at for implementing workflows with live linking:

1. Positioning and Whiteboxing - This will focus on solidifying the fundamental Live Link architecture, and workflow model, and make sure the basic structure is solid. During this phase we can also create useful stubs for DCC tools other than Blender.
2. Complex Mesh editing - Improve mesh syncing such that all changes are immediately seen, even if the mesh is dense or

complex. This will expand the workflows to allow more than just whitebox modeling.

3. Texture/Material editing - ability to sync textures and materials. Bonus points if we can link the Blender material node system to Atom renderer.
4. Animation editing - The ability to sync, and create animation clips from inside blender, and use them immediately in O3DE