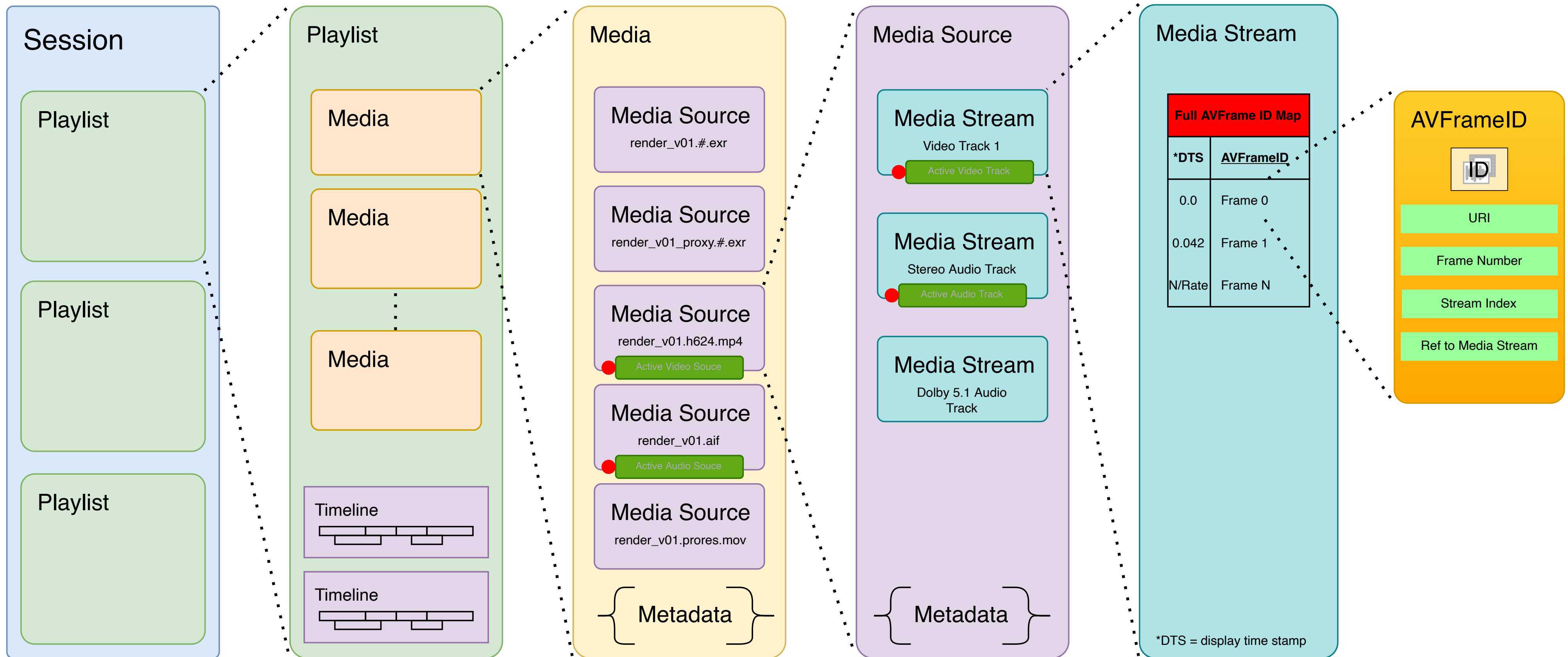
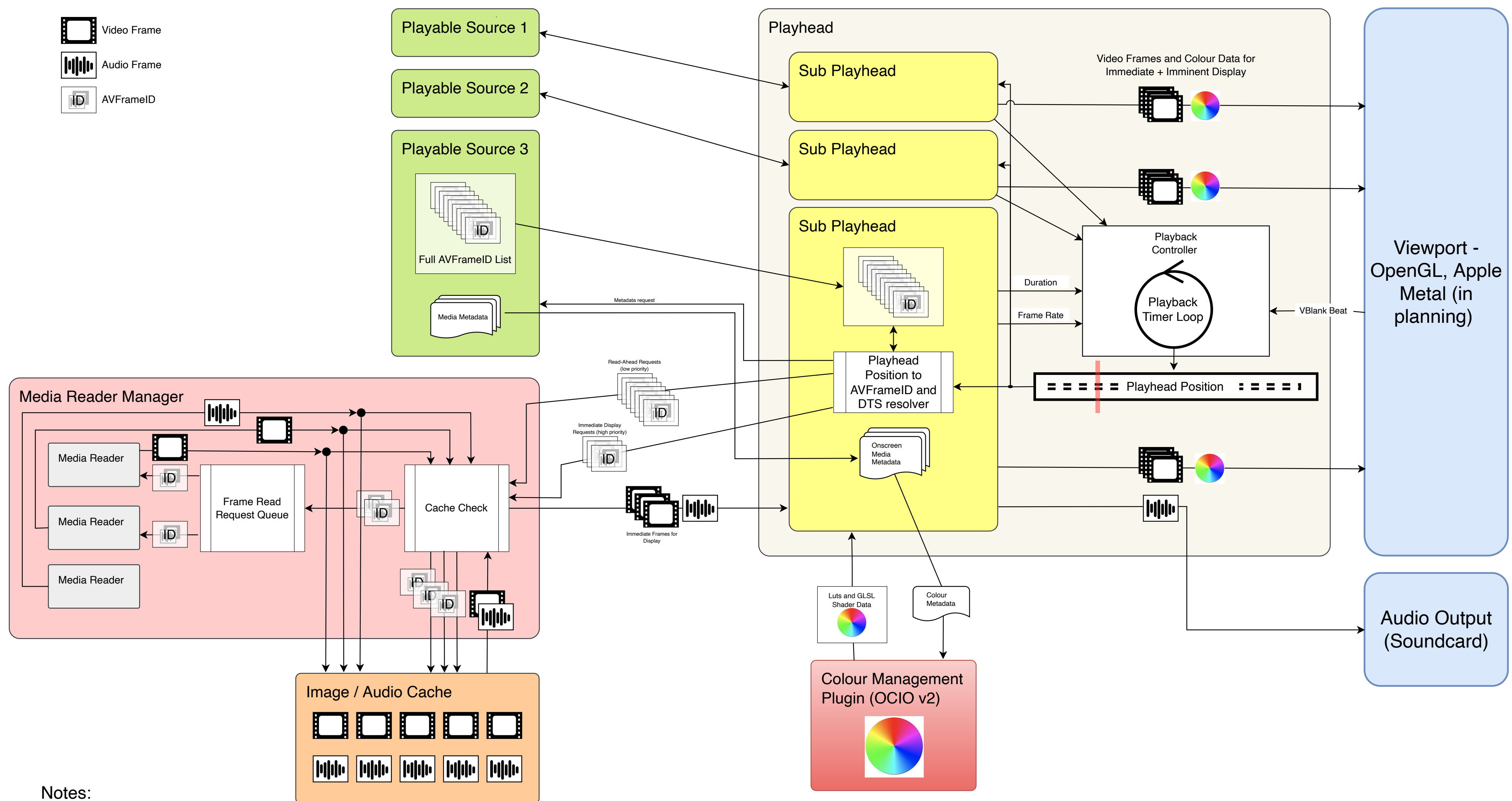
xStudio Architecture - Fig 1. Media Management Data Model



Notes:

- Classes are implemented using the Message Passing Interface (MPI) 'Actor Model' design pattern using 'C++ Actor Framework'.
- Each class instance executes code in inherently thread safe way on threadpool managed by framework. Highly concurrent.
- The purpose of the Media Source object within the hierarchy is to resolve the universal VFX practice of generating/maintaining multiple encodings of the same visual output (and associated audio). For example a CG render may be output at 4k EXR, from which one or more additional encodings would be created for various purposes, like lightweight motion compressed quicktime, a higher quality review quicktime or more compressed and down-res'ed EXR proxies. By encapsulating these within the broader 'Media' class we aim to simplify the logical access to these various (usually distinct filesystem) resources.
- The Media Stream entity further resolves a Media Source into its logical internal AV components. Specific examples are containerised encodings like MP4 that can carry multiple video and (more commonly) multiple audio tracks. Similiarly multipart EXRs are handles with a separate Media Stream for each 'part' allowing the application to enact switching between parts or displaying data relating to those parts in the UI, say.
- The Media Stream maintains a map of 'AVFrameID' which is a lightweight struct containing all the information required by a Media Reader to read and decode a single given frame of audio/video data. The passing of these structures ultimately enables the components involved in playback to request frames for decoding, cacheing and display.

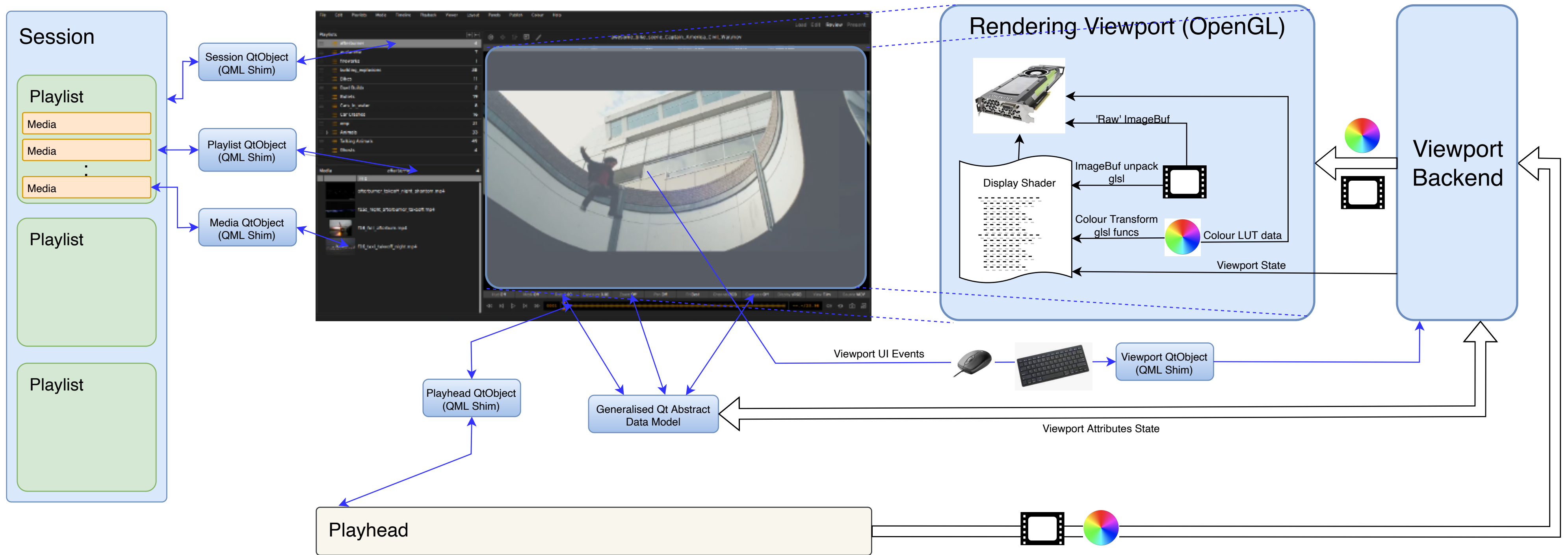
xStudio Architecture - Fig 2. Playback Engine Components and Data Flow.



Notes:

- Playheads are designed to run completely independently of UI components, with the exception of receiving a message on display refresh (framebuffer swap) to assist with video sync.
- The application can support any number of running playheads.
- Sub-playheads allow image data to stream from mulitple sources in sync., providing a means for comparing more than one source in an A/B mode or a contact sheet, for example.
- The Playhead position is maintained to a much higher granularity than source frame rate or video refresh rate. This allows sub-playheads to play media with different playback rates and maintain best possible synchronisation.
- A valid Sub-playhead source is any entity that simply a map of type <DTS, AVFrameID> through a common message handler - in other words a list of AVFrameIDs with a timestamp relative to the first frame. In practice this can be a Media item, a Media Source, a Timeline, an EditList, a Retimer (that wraps another source of AVFrameIDs) etc.
- The sub playhead delivers multiple video frames including the current frame and one or two subsequent frames to the Viewport, each stamped with an accurate display time.
- The Viewport is responsible for displaying the right frame at draw time and starting asynchronous uploads of the following frames to GPU memory where possible.
- Media Readers attach static GLSL shader code and data to the Video Frames that they generate. This GLSL code does pixel unpacking to RGB from the raw image buffer.

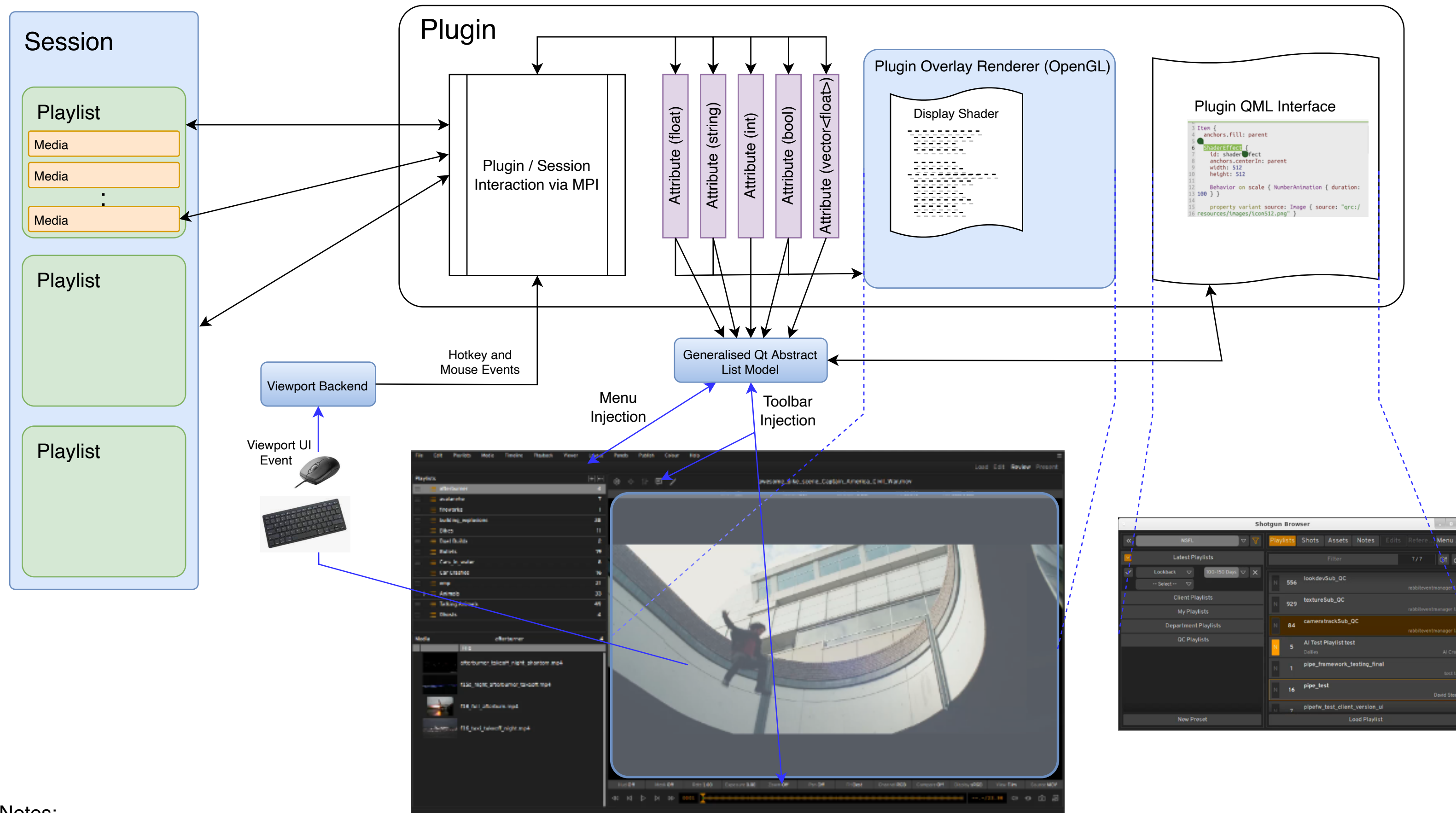
xStudio Architecture - Fig 3 Backend / UI



Notes:

- For most backend components a dedicated shim class exposes state data to Qt/QML for each component type
- QML layer visualises backend data, interacting through the shim, so no backend execution happens in the UI layer.
- Some backend components declare 'attributes' that can be exposed in the UI layer via a generalised model/delegate/view approach (e.g. toolbar buttons which are built at runtime)
- Planned refactoring will make greater use of this generalised approach, we may be able to drop the 'shim' classes altogether for a full, more flexible 'runtime constructed' GUI.
- Viewport renderer has no dependency on Qt and therefore can render into any OpenGL surface. It is hoped this approach will help with port to Apple metal.
- All user interaction is forwarded through the application's message passing framework and as such the Viewport backend class does not have to execute within the main Qt event loop.

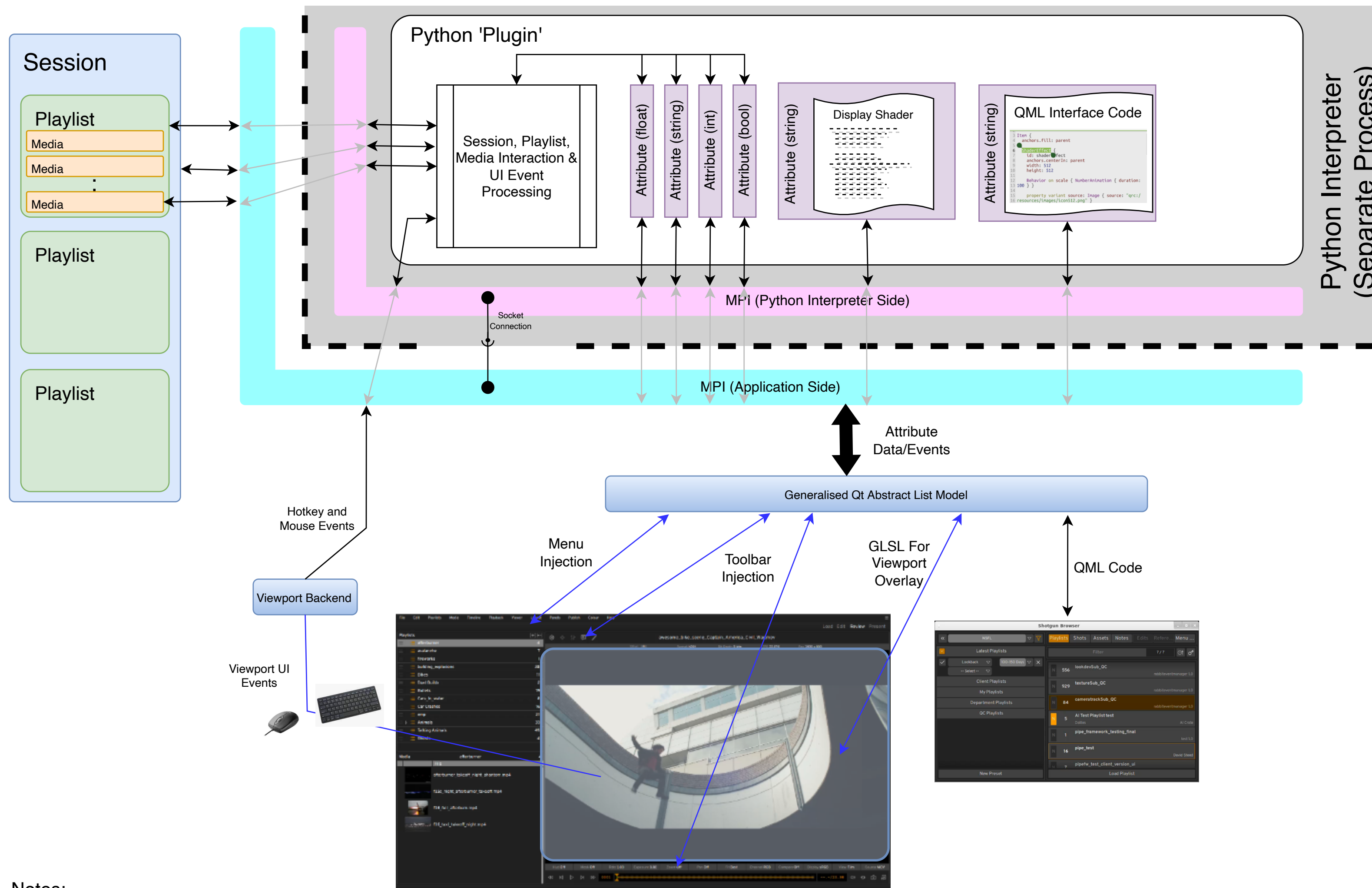
xStudio Architecture - Fig 4 C++ Plugin



Notes:

- There is no Plugin specific C++ API - plugins are authored like any core component of xStudio
- Most xStudio classes have a public message handling interface, plugins can interact freely with core components through the MPI framework.
- Core components are reachable via the public object registry
- Plugins can include OpenGL C and GLSL code to render viewport overlays
- Plugins can include QML code to add new interfaces
- Attributes can be used as a convenient way to define data that connects QML UI to C++/Python backend (and OpenGL renderer)
- Attributes can be used to add menus, menu items and toolbar widgets to the main xStudio interface

xStudio Architecture - Fig 5 Remote Control (Python)



Notes:

- xStudio's MPI framework can transparently send and receive messages over a network socket
- This feature allows Python plugins to run in a completely separate process, if desired
- Since they run in a separate process, a Python controller with a novel PySide interface could be created, for example, that would not interfere with xStudio playback performance
- The Python API binds the MPI framework functions and types for sending and receiving messages and mirrors backend core classes.
- Objects within the Session can be interacted from the Python side using their public message handlers
- Attributes can be used to add menu items, standard widgets (like toolbar buttons) or custom widgets to the xStudio interface
- Python plugins can also pass GLSL and QML code to the xStudio application for creating graphics overlays or new interfaces, as required
- GLSL uniforms can be automatically mapped to plugin attributes
- Direct OpenGL rendering hooks are not possible, however.
- C++ plugin/controllers can also be implemented in a similar way