Multithreading USD and Qt: Adding Concurrency to Filament

Jonathan Penner
Animal Logic
Vancouver, BC, CA
jonathan.penner@animallogic.ca

Jakub Jeziorski
Animal Logic
Vancouver, BC, CA
jakub.jeziorski@animallogic.ca

Kevin Russell
Animal Logic
Sydney, NSW, Australia
kevinr@al.com.au

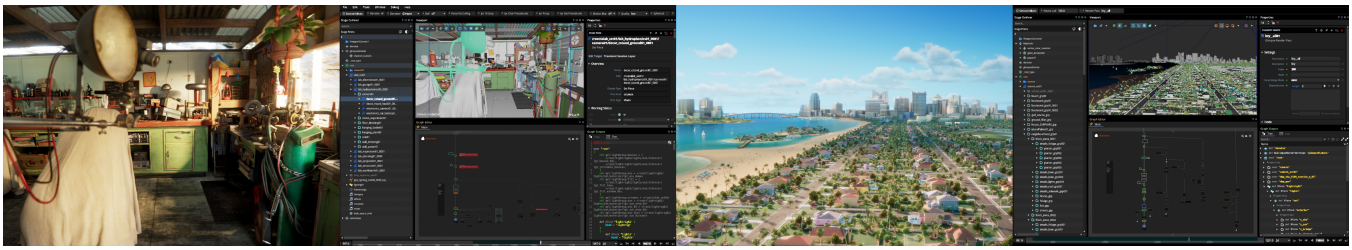

Figure 1: From left to right, an ALab promotional video shot final comp, the ALab promotional video shot in Filament, an opening shot from Leo final comp, the Leo opening shot in Filament.

ABSTRACT

As production scene complexity and CPU core count increase, the performance of software used to interact with the scenes may not scale accordingly. Filament is Animal Logic's in-house, USD-based, PyQt lighting DCC, and a key area for improving Filament was increasing performance and responsiveness when working with large production scenes. USD, Qt, and Python all have their own multithreading considerations, requiring coordination between all three to work well. Filament was updated to parallelize USD stage processing to reduce processing time, as well as adopt asynchrony to keep the main GUI thread unblocked, greatly improving artist experience. These updates demonstrate a model for multithreading USD stage access to improve other applications working with USD.

CCS CONCEPTS

• **Computing methodologies** → **Concurrent computing methodologies**; *Graphics systems and interfaces.*

KEYWORDS

USD, Qt, Python, Multithreading, Parallel Computing, UI

ACM Reference Format:
Jonathan Penner, Jakub Jeziorski, and Kevin Russell. 2024. Multithreading USD and Qt: Adding Concurrency to Filament. In *Proceedings of Digital Production Symposium 2024 (DigiPro '24).* ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

DigiPro '24, July 27, 2024, Denver, CO

ACM ISBN XXXX-XXXX-X/24/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Since 2018 Animal Logic has been using its in-house DCC Filament [Agland et al. 2020] for lighting feature animation and VFX productions, such as Peter Rabbit 2, DC League of Super Pets, and Leo. It is primarily written in Python with PyQt, along with supporting C++ code for heavy-lifting processing tasks. Artists author node graphs to make a series of changes to a USD stage, such as creating lights and setting up render passes, similar to applications like Katana. These node graphs are "executed" by Grip, a C++ library with Python bindings. At this point views, models, and other Filament code connected to the stage (referred to as "stage clients"), update to display the new state of the stage.

While the feature set in Filament grew to accommodate each production's needs, its single-threaded USD stage processing meant artist-initiated changes would make the user interface unresponsive while the changes were processed. As a result, this processing would often be slow to complete. During the production of Leo, Filament's stage processing was parallelized to improve performance, and later made asynchronous to keep the user interface responsive. This posed some challenges, as USD, Qt, and Python each interact with multiple CPU threads differently.

2 CONCURRENT USD

The internals of USD, such as its stage composition, are heavily parallelized. Its threading model allows for multiple threads to read from the stage, or have one thread writing to the stage, meaning threads cannot read from the stage at the same time as another thread is writing to it. Since traditional threading primitives like locks and mutexes are not exposed through USD's API, a different approach is needed for managing concurrent stage access.

NVIDIA's *Omniverse* [2021] approaches concurrency with its *Fabric* library, which provides a thread-safe scene representation generated from underlying USD data. This approach works well since USD itself is isolated from the concurrency; however it is only available within *Omniverse*.

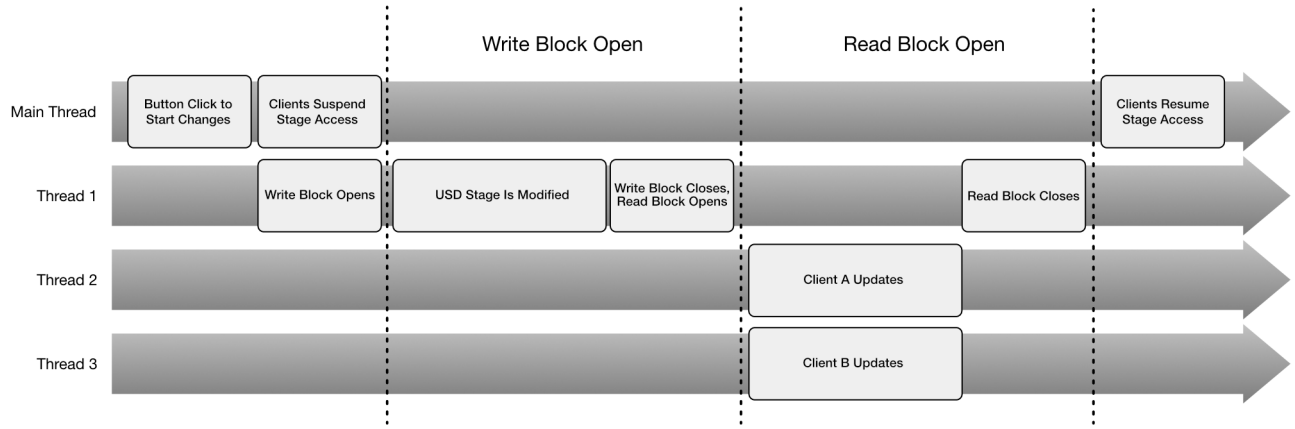

Figure 2: Stage Access in *AL_USDChangeBlock*

We developed an alternative approach with an internal C++ library called *AL_USDChangeBlock* to coordinate different stage clients across multiple threads. *AL_USDChangeBlock* handles stage access by defining two types of blocks of time: "write blocks" and "read blocks". While a write block is open, no threads are allowed to read data from the stage. Conversely, while a read block is open, threads are allowed to read data from the stage, but no thread is allowed to modify it.

This simple approach means multiple threads can access the stage without violating USD's threading rules. If a stage client wants to modify the stage, it first checks if a write or read block is already open, and if not, opens a new write block, performs its changes, and then closes the write block. *AL_USDChangeBlock* aggregates all the USD `ObjectsChanged` change notices that were sent during the write block to create a minimal change list of all the affected prim paths.

AL_USDChangeBlock then opens a read block to let stage clients update themselves based on that minimal change list with the new state of the stage. This prevents other clients from further modifying the stage during the updates. The read block closes, at which point clients are free to modify the stage again in a new write block. Custom `TfNotices` are registered for write and read block open and close events and dispatched using USD's synchronous `TfNotice` framework, letting clients not only check for stage access on-demand but also respond as the events happen.

To allow for parallelism, stage clients written in C++ can register as "readers" with *AL_USDChangeBlock* and provide an update callback function. When a read block opens, each registered reader's update callback is called in separate threads with the minimal change list, and the read block closes when each callback has finished. This means updating multiple stage clients is only as slow as the slowest client instead of the sum of each of them. Since the read block safeguards the stage from new modifications, making this process asynchronous is simple by handling the read block in another background thread, leaving the main GUI thread responsive.

3 CONNECTING QT

Qt's main restriction on multithreading is that any operation modifying visual items must run on the main GUI thread. This poses challenges for widgets that need to respond to `TfNotices` that may be sent from other threads. Fortunately, Qt's signal and slot mechanism is designed with threading in mind. If an object emitting a signal is on the same thread as an object connected to it, the corresponding slot will be called synchronously. If the objects are on different threads, Qt will automatically post an event to the receiver thread's event loop to call the slot when the thread is next processing events.

In *Filament*, widgets and `QObject` stage clients use intermediate objects that register with the `TfNotices`. When a notice is sent, such as a write block open event, the intermediate object forwards the contents of the notice to the widgets using a Qt signal, letting Qt automatically handle thread switching using its event loops. This leaves each widget to safely update on the main thread even when the stage updates come from other threads.

While write and read blocks are open, *Filament* is put into a "suspended" state, preventing modifications to the stage but remaining responsive. Visual items that modify the stage (such as viewport manipulators, certain context menu actions, and attribute editor fields) are disabled.

The following is an example of the entire process. *Filament*'s outliner view is a PyQt QTreeView and QAbstractItemModel, with its model populated by a C++ scene index.

- When a write block opens, the outliner's intermediate `QObject` forwards the notice to the view and model through signals.
- The view displays a loading indicator and disables any visible stage-modifying context menus, while the model stores any persistent indexes and stops reading new data from the stage.
- Once the write block closes and the read block opens, the C++ scene index — which is registered as an *AL_USDChangeBlock* "reader" — will update its data in parallel on background threads.

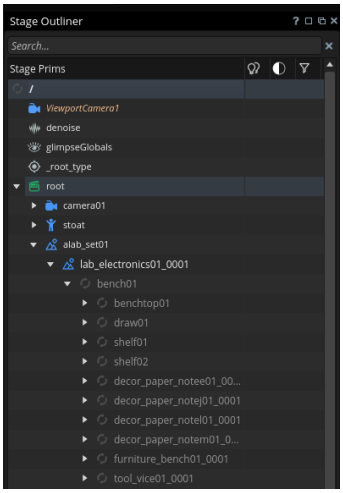

Figure 3: Filament's outliner displaying partially loaded prims while a stage update is in progress.

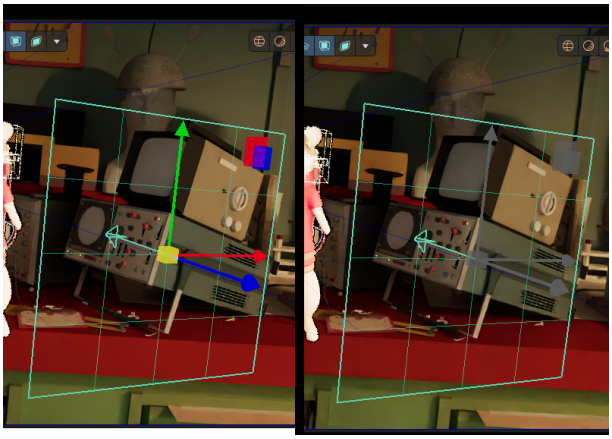

Figure 4: Filament's viewport manipulators enabled (left) and disabled (right) while a stage update is in progress.

- Once the index has finished updating, the outliner's model refreshes persistent indexes and emits a `layoutChanged` signal to the tree view, and the view displays the new stage data and hides the loading indicator.
- When the read block closes, the view re-enables any previously disabled context menus.

This process lets stage modifications and updates happen on background threads without causing incorrect concurrent USD stage access.

4 UNLOCKING PYTHON

Python is limited to single-threaded execution by its Global Interpreter Lock (GIL). While key aspects of *Filament* written in C++ (such as its outliner scene index) process updates in background threads, actions that are initiated from Python still block the main thread. *Filament* uses *Grip*'s Python bindings to modify the stage and create artists' lighting setups, but executing those changes would still run on the main thread.

A goal of adding concurrency was to make *Grip* execution asynchronous on a background thread. As a starting point, the calling Python code to start execution in *Filament* was moved to a `QObject` on a separate background `QThread`. Even though technically this ran *Grip* execution on a separate thread, the GIL prevented all other Python threads from running until execution completed, thereby freezing the user interface.

Fortunately, since *Grip* execution internally doesn't access Python objects, it can safely call Python's C API to unlock the GIL. With the calling Python code on a separate thread and the GIL unlocked, the main thread is free to process Python code and events while *Grip* execution runs, making the user interface stay responsive. Changes to the *Grip* graph during updates are queued to be executed once the current set of updates have finished, meaning artists don't need to wait for updates to finish to continue making changes to the graph. This execution also happens inside a write block, keeping the USD stage protected from concurrent access.

5 RESULTS

When concurrency was originally added to *Filament*, we observed an average 4x speedup in USD stage update time. Some scenes on Leo were up to 7x faster, with the update time in one case going from roughly 60 seconds down to 8 seconds.

Since concurrency was added we've made other optimizations and re-architected some key aspects. The previous performance difference, with and without the parallelism, isn't reproducible, but overall *Filament* runs even faster than before, in addition to the updates happening on background threads. The primary benefit of our concurrency approach is asynchrony.

Objectively, *Filament* is much more responsive than before. There is a substantial improvement in the time the main thread is blocked during updates with and without concurrency. Since graph changes during updates get queued, artists are still able to change their graph and work while updates are happening, as well as use views like the outliner and viewport.

Table 1: Leo - Typical Production Shot - 2.2 Million Prims. Without concurrency is entirely blocked.

Metric	With Concurrency Time Blocked	With Concurrency Total Time	No Concurrency Total Time
Average	0.371s	11.744s	15.604s
Min	0.239s	10.468s	11.931s
Max	0.371s	12.878s	18.544s

Table 1 shows that an average production shot on *Leo* with 2.2 million prims, the main thread is blocked for only a few hundred milliseconds during the rest of the *Filament* updates. Since stage clients update in parallel, the total time for updates to finish is faster too. The total update time is also less variable than when the updates are processed without concurrency.

Table 2 shows results for a production shot with twice the number of prims. The performance gain is less than the typical case,

Table 2: Leo - Heavy Production Shot - 4.4 Million Prims. Without concurrency is entirely blocked.

Metric	With Concurrency Time Blocked	With Concurrency Total Time	No Concurrency Total Time
Average	1.370s	21.981s	24.586s
Min	1.015s	19.909s	21.633s
Max	1.658s	23.209s	33.312s

Table 3: ALab Promotional Video - Typical Shot - 242k Prims. Without concurrency is entirely blocked.

Metric	With Concurrency Time Blocked	With Concurrency Total Time	No Concurrency Total Time
Average	0.2090s	0.549s	0.590s
Min	0.178s	0.515s	0.549s
Max	0.234s	0.599s	0.631s

but still has the great benefit of having the user interface still be responsive as the updates are being processed.

Table 3 shows that when the scene is small enough, updates with concurrency are sometimes slower than processing the updates without concurrency. When multiple clients update with their own parallelism there is noticeable overhead of task and thread switching when they run simultaneously. However, if task management were more efficient, and the time to process theoretically went down to about 0.4s, the perceptual difference between 0.4s and 0.5s would not be noticeable.

Subjectively, the reception from the lighting artists has been extremely positive. Artists are freer to work at a faster pace without needing to wait for *Filament* to finish all processing before being able to make new changes. Before concurrency was added and the main thread blocked, artists would sometimes assume the application had crashed and force-quit their session, unnecessarily losing work. Now they have more confidence that *Filament* is running correctly; they don't waste time relaunching the application and reopening their scene.

Most significantly, artists have said that they are happier working in *Filament* and enjoy the process more than they did in *Filament* before concurrency. They have fewer obstacles and slowdowns that impede their creative flow so they can focus on the art of lighting. As developers, we can get focused on numbers and performance statistics, when in reality the underlying goal is to let artists be artists.

6 DISCUSSION

The approach of concurrency in *Filament* requires all stage clients to cooperate with any open write or read blocks. It's up to the clients to follow the rules instead of preventing bad access at the API level, like the approach in NVIDIA's *Fabric* API, or other concurrency strategies such as actor models. That said, the approach is straightforward, both conceptually and practically, in the development of *AL_USDChangeBlock* and retrofitting *Filament*'s different stage clients to support concurrency.

Taking any synchronous system and making it asynchronous introduces new challenges for developers. Modern programming language features such as async/await aim to ease the mental burden of managing task queues or asynchronous callback functions. In the future we would like to explore integrating async/await to let clients request a write or read block and "await" for any current changes to complete, letting running threads yield execution instead of blocking entirely.

7 CONCLUSION

Internally at Animal Logic, *Filament*'s concurrency exceeded expectations for application responsiveness, raising our standards to a much higher level. Artists spend less time waiting to see their changes and have a better experience overall working in *Filament*, making it easier to work with the increasingly large scenes involved in productions. Our hope is that the lessons and techniques adopted by *Filament* can be used similarly to improve user experience in other applications within and outside of Animal Logic.

ACKNOWLEDGMENTS

We would like to thank Steve Agland for encouraging this idea and pursuit of making *Filament* more responsive; Curtis Black, Basile Fabroni, and Andy Chan for their guidance in developing *AL_USDChangeBlock*; and Rodrigo Janz, Herbert Heinsche, Etienne Marc, Joshua Nunn, and the rest of the lighting team for testing these substantial changes.

REFERENCES

Steve Agland, Jakub Jeziorski, Manuel Macha, Simon Bunker, Francesco Sansoni, and Eoin Murphy. 2020. Grip and Filament: A USD-Based Lighting Workflow. In *ACM SIGGRAPH 2020 Talks* (Virtual Event, USA) *(SIGGRAPH '20)*. Association for Computing Machinery, New York, NY, USA, Article 33, 2 pages. https://doi.org/10.1145/3388767.3407350

NVIDIA. 2021. *Omniverse Website.* Retrieved May 1, 2024 from https://www.nvidia.com/en-us/omniverse/