Presto Execution System: An Asynchronous Computation Engine for Animation

George ElKoura
Pixar Animation Studios
July 24, 2013

Introduction

We present an asynchronous computation engine suitable for use in an animation package. In particular, we describe how this engine is used in Pixar's proprietary animation system, Presto.

We will start by first describing what Presto is and concentrate on how it is used by two different disciplines. We will then dive into Presto's Execution System, its phases and the structures that make up its architecture. We then describe how several different multithreading strategies can be implemented using the architecture. We focus on Background Execution, a feature that allows users to keep working while soon-to-be-needed computations are performed in the background. We discuss the requirements and considerations from a user interaction perspective. We then end with some of the future work we'd like to pursue.

Presto

Presto is Pixar's proprietary, fully featured, animation package. Besides the main interactive application, Presto is built on top of a rich set of reusable libraries. The application supports integrated workflows for a variety of feature-film departments including rigging, layout, animation and simulation. It also provides built-in media playback and asset management tools.

For the purposes of this course, we will mainly discuss Presto's Execution System. We will use two common disciplines, rigging and animation, to illustrate how the system works. Much of what will be discussed applies to other disciplines as well. Though we will mainly talk about posing points, the execution engine itself is generic and is used for other kinds of computations as well.

One of the challenges in Presto is its integrated architecture. In a single session, the user may wish to animate or do some rigging or run a sim or all three without an explicit context switch. Some of these tasks do not lend themselves well to a multithreading environment, and yet must coexist seamlessly with all features of the application.

Rigging in Presto

Rigging is the process of modeling the behavior of the characters and props on a show. Riggers describe how, given a set of inputs, a character poses. Riggers use what is effectively a visual programming language to describe the structure and behavior of their models. In Presto, riggers do not directly author the execution structures. Instead, they visually author human-readable objects using higher-level concepts. These objects later get transformed into data structures that are designed specifically for efficient computation.

Network topology changes often during the course of regular rigging workflows. Geometry topology also changes during rigging, but is fixed once the rig is finalized. In other words, the rig, itself, does not modify the topology of the geometry.

Animation in Presto

Animation is responsible for bringing the characters to life. Animators supply input values to the rig in order to hit desired character poses.

The topology of the character's geometry does not change during the course of regular animation workflows. Network topology may change, but does so infrequently. Addition of post-rigging deformers, such as lattices, and adding new constraints are examples of animator workflows that do change the network topology.

Presto's Execution System

Presto's Execution System is a general-purpose computation engine. Given a set of inputs (e.g. animation splines) and an execution network (e.g. derived from a rig), the job of the execution system is to provide the computed result as quickly as possible. Common computations include posing points, calculating scalar fields, determining object visibility, and so on.

Much like other such systems, at its core, the execution system in Presto evaluates a data flow network. Presto's data flow network is vectorized, meaning that many uniquely identifiable elements may flow along a single connection. We don't need to get into the details of this aspect of the system to talk about multithreading, but it is worth noting as it will come up now and again.

In the following sections we'll explore the architecture of the execution system, how the parts fit together, and how they lead to a framework that is amenable to multithreaded computation.

Phases of Execution

The execution system is broken up into three major phases:

- Compilation
- Scheduling
- Evaluation

Each phase amortizes costs for the next phase. The phases are listed in the order that they run, also in the order of least-frequently run to most frequently run and from most to least expensive runtime costs.

Compilation

As mentioned above, riggers author scene objects using a rich, high level visual language. This allows riggers to be efficient at their main task. By abstracting away details of the execution system, we allow riggers to concentrate on building the rigs. However, the visual paradigms that allow for a fast rigging process may not always lend themselves to fast rig evaluation. Compilation is the phase of execution that converts the human-readable scene objects into optimized data structures that can be used for fast repeated evaluations of the rig (e.g., during animation).

The result of compilation is an execution network consisting of nodes and connections. While riggers deal in various concepts like connections, deformers, weight objects, and so on, once compilation is done, the execution system sees only one homogenous concept of execution nodes.

In addition to enabling fast evaluation, compilation provides a layer of abstraction that allows us to keep the assets separate from the data structures required by our system. Assets are time consuming and expensive to author, and we would like to have them be independent from the implementation of the execution system. That is to say, if we decide that we've been doing things all wrong in how the execution system's data structures are organized, we wouldn't have to modify a lot of assets to overhaul the system. Compilation also provides a convenient place to perform optimizations at the network level.

Full network compilation typically happens only when a character is first loaded in a session. Rigging workflows invoke an incremental recompilation code path that builds the network in pieces as the rig is developed. The results of compilation can be serialized, in which case full network compilation can be eliminated from most use cases.

Scheduling

Given a set of desired output values (e.g. the posed point positions of a character), which we call

a request, scheduling is responsible for performing the dependency analysis necessary to determine which nodes need to run, in order to satisfy the request. Scheduling serves to amortize dependency analysis costs that would otherwise have to be incurred during each network evaluation. The specifics of the analysis performed during scheduling is up to the implementation. For example, it may be beneficial for certain schemes that scheduling determine the partial ordering in which the nodes run, and for others, it might be more efficient that scheduling only determine what nodes need to run, and the ordering is left to the evaluation phase.

Scheduling is performed more often than compilation, but not as often as evaluation. Scheduling must be performed after network topology changes caused by incremental recompilation, and therefore occurs more often during rigging workflows, and rarely during animation workflows. Animation workflows may cause scheduling, for example, when adding a new constraint or creating new deformers.

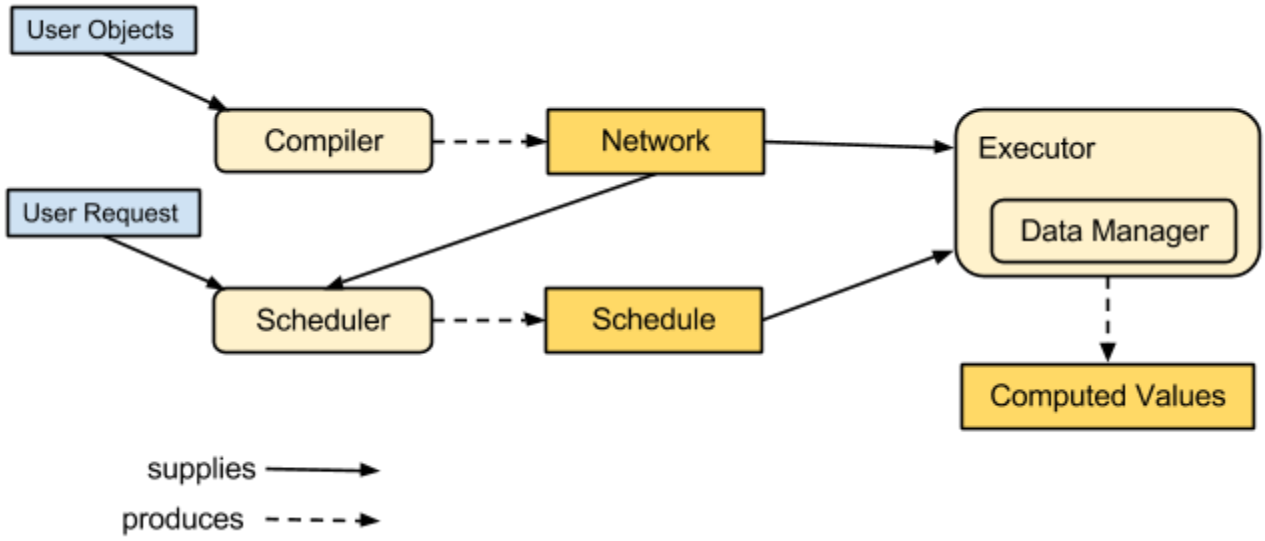
Evaluation

Evaluation is the most frequently run phase of execution. Its job is to run the nodes in the network as determined by the schedule in order to produce computed values for the requested outputs. Evaluation is run every time an input value changes, or a new output is requested, and the results are pulled on (e.g. to satisfy a viewer update).

Engine Architecture

The execution system consists of the following data structures:

- Network
- Schedulers
- Data Managers
- Executors



Network

The network is generated from user-authored scene objects and is a static representation of computations and the dependencies among them. A network is made up of nodes and connections. A node is made up of zero or more inputs and zero or more outputs (a leaf node with zero outputs is only used for tracking and broadcasting dependencies to external systems). Client code requests computed values by specifying node outputs.

This is a data flow network with added support for vectorization. *It is important to note that no stateful information is stored in the network, it embodies only the computation's static structure.*

Schedulers

Schedulers are responsible for the scheduling phase of execution and produce schedules that are held on to by the system and are used by the executors for repeated evaluations.

Schedulers can be specialized for experimenting with different kinds of multithreading schemes, or for providing more analysis that can be taken advantage of by executors.

Data Managers

Data managers are simply data containers for nodes in the network. They are used by the executors to store the computed data. Not storing data in the nodes themselves is an important architectural feature of our network. You would specialize a data manager only if you wanted to provide a faster implementation. There is no deep work going on in these objects.

Data managers store computed data as well as validity masks and other per-node or per-output data that is required by the executor.

Executors

Executors are the workhorses of the execution system. Executors orchestrate computation using the compiled network, a schedule and a data manager. They run the “inner loop” and need to run as efficiently as possible.

Executors can be arranged in a hierarchy, where the child is allowed to read (but not write) from the parent in order to avoid redundant computations. This feature is put to use in a few different ways in the system, but for the purposes of this course we will discuss how it is used for Background Execution a little later.

Executors can be specialized to supply different algorithms for running the network. You might want to specialize an executor to experiment with different multithreading techniques, for example, or to target a different platform. When we later discuss the different multithreading strategies, they are primarily implemented in different kinds of executors.

User Extensions

Since we have always intended for our system to be run in a multithreaded environment, we needed to carefully consider the responsibilities we impose on clients of the system. One of our goals was to make it as safe as possible for users to write plugin code without worrying about multithreading in most cases.

One of the problems that often complicates multithreading is having to call out to client code. Not knowing what the clients are going to do, and what resources they might acquire makes it difficult to create a resilient and robust system, let alone one with predictable performance characteristics. Our system is no different, but we do take a few extra precautions in order to minimize the chance that a user will accidentally shoot themselves in the foot. Dogged determination to shoot oneself in the foot, on the other hand, is a little more difficult to prevent. The structure of the system helps avoid common multithreading pitfalls for users in the following ways:

Dependencies Declared a Priori

The system is set up such that users declare ahead of time the inputs that their node computations will consume. They can make certain dependencies optional (meaning that if they are not available, they can still produce a reasonable answer), but they cannot add more

dependencies at run time. The static structure of the network is fixed and is built based on the dependencies that the user declares.

Client Callbacks Are Static Functions

All client callbacks of the execution system are expected to be static functions (i.e. not class methods), that are passed a single argument. They take the following form:

```
static void MyCallback(const Context &context) {
    …
}
```

The callbacks are disallowed from being methods on objects by structuring the code in such a way as to force the binding of the callback before the user can have access to any of the specific instances of the objects in question.

Clients must read their inputs and write to their outputs using only the API provided by the passed-in context. This structure discourages users from storing any state for a node that is not known to the execution system. Users are not intended to derive new node types.

Presto Singletons are Protected

Some of Presto's libraries provide API that allows users to access system functionality through singleton objects and registries. Some of these libraries are not thread safe and are not allowed to be called from user-code running inside of execution. As a safety measure for client-code, we detect and prevent the use of such singletons while running in this context. Users are of course still free to create their own singletons and access static data, but they must be responsible for the consequences of doing so unsafely.

Iterators

Access to large, vectorized, data (e.g. points) is provided through iterators that are easy to use and hide from the user the detail of where the memory is stored, or what subset of the computations their callback is dealing with. This allows us to modify memory allocation and access patterns, as well as modify our multithreading strategies, without changing client code.

And then there's Python...

Python is famously known to not play well within a multithreaded system. For this reason, we

initially disallowed the use of Python for writing execution system callbacks. However, there are some clear advantages to supporting Python:

1. Python allows for quicker prototyping and iteration.
2. Python is accessible to a wider range of users than C++.
3. Python has a rich set of packages (e.g. numpy) that users would like to leverage.

These benefits make it difficult to adopt alternatives (e.g. a different existing language, or a custom-written language).

Global Interpreter Lock

The Python interpreter is not threadsafe. It cannot be run from multiple threads simultaneously. Python provides a global lock, the Global Interpreter Lock (GIL), that clients can use to make sure that they don't enter the interpreter from multiple threads simultaneously[1]. A thread that needs to use Python must wait for its turn to use the interpreter.

Getting the locking right is tricky; it is easy to find yourself in classic deadlock situations. Consider the following user callback (though this code should be discouraged due to its use of locking to begin with, it nevertheless happens):

```
static void MyCallback(const Context &context) {
      Auto<Lock> lock(GetMyMutexFromContext(context));
      ...
      EvalMyPythonString(str); // A function that takes the GIL
      ...
}
```

Now consider the sequence of events in the threads which result in a deadlock:

MAIN THREAD	OTHER THREAD
Python command acquires GIL	Work started
Computation Requested	MyCallback runs and acquires myMutex
	MyCallback now waits for GIL
MyCallback runs and waits for myMutex	(waiting for GIL)

[1] http://www.dabeaz.com/python/UnderstandingGIL.pdf

One thing to note about this situation is that if, in the main thread, the call was made from C++, then there would be no need to hold the GIL in the main thread, and everything would be fine. If, however, it is called from Python, we get the hang. Moreover, neither subsystem knows about the other, the locks are taken in client code. The client code could be smarter about the order in which the locks are acquired, but that's not always a viable solution. In this case, the client is calling out to a function in a library, and may be unaware about it taking the GIL to begin with.

One solution in this case is that, in the main thread, we no longer need to be holding the GIL once we make a computation request in C++. Ideally, you would structure your bindings to always release the GIL upon re-entry to C++.

This is a good example of why global, system-wide, locks are a bad idea. Use lock hierarchies[2] to avoid the common deadlock patterns if you must have wide-reaching locks. Better still, prefer locks that have local, easy to reason about, scope if you must lock at all.

Performance

Anything running in Python is the only thing running in Python. This means that if your execution callbacks are all implemented in Python, you lose all the benefits of a multithreaded system.

However, Python can still be effective for writing the control logic and have the heavy lifting be performed in C++ (with the GIL released).

Memory Access Patterns

Although we tried to construct the system in as flexible a way as we could, a major guiding principle was to make sure that we paid attention to memory access patterns. How memory is accessed is extremely important for performance, and we wanted to make sure that our desire for a flexible system did not impose any detrimental patterns for memory access.

It's important that bulk data is processed in a way that is compatible with the processor's prefetcher[3]. Luckily, modern processors are clever and do a good job at prefetching memory, therefore, it is preferable to remain within the patterns that the processor can detect and pre-fetch. Hardware prefetchers typically work best with ascending access order. Though more complicated patterns can also be detected, such as descending and uniform strides, it's always best to check your hardware specifications. Arranging your data in a structure-of-arrays rather

---

[2] http://www.drdobbs.com/parallel/use-lock-hierarchies-to-avoid-deadlock/204801163

[3] http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf, section 2.1.5.4, page 2-23

than an array-of-structure often helps the prefetchers improve your application's performance.

Locality is important for performance. Most common multicore platforms available today use NUMA[4], so keeping memory and code access local to each core will improve the scalability of a multithreaded system. Memory allocators can help achieve this goal. Using an allocator written with this sort of scalability in mind, like `jemalloc`[5], is preferable to an unaware allocator.

It's important to always measure the performance of your system and monitor how changes to the code affect it. Modern hardware architectures are sufficiently complicated that intuition and educated guesses often fail to predict performance. Always test performance.

Flexibility to Experiment

We designed the system with two main unknowns: 1) we didn't know exactly what architecture it was going to run on, and 2) we didn't know how the user requirements were going to evolve. We didn't want to base our architecture on the current state of hardware and user desires. We attempted to build in as much flexibility as we could.

For example, we have written a variety of different executors to satisfy different computation models and multithreading strategies. If the target hardware architecture changes, we expect only to write a new executor, which is a relatively small piece of code. Schedulers are not often specialized, though they may be in order to add acceleration data for use with specialized executors.

Another example is if user requirements change such that the objects they deal with need to be re-thought or re-designed, much of the execution system can remain untouched. In this case, only compilation would need to be re-written. Though, admittedly, that is a lot of code.

Multithreading Strategies

Finding the right granularity for the tasks to run in parallel is an important aspect of getting the most performance from the hardware. Too many small tasks cause too much time to be spent in context switching and other thread management overhead. On the flip side, too many large tasks can lead to poor utilization.

One of the factors we have to keep in mind while choosing a granularity for our specific domain

---

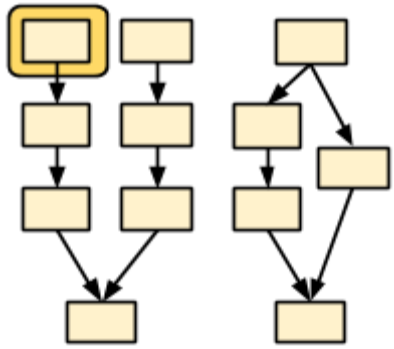
[4] http://software.intel.com/en-us/articles/optimizing-applications-for-numa

[5] https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919

is that we have a fairly small time budget for evaluating the network. We'd like to aim for running the rig and drawing the character at 24 fps. Even if we ignore the costs of drawing the character, that gives us less than 42 ms to run approximately 30,000 nodes (e.g. for a light character). We therefore have to choose a granularity for the units of work that is compatible with these target time budgets.

The architecture described above allows for easy experimentation with various multithreading strategies in order to find the right granularity for the greatest performance. In this section we'll explore only a few of the possibilities. Note that these strategies are not mutually exclusive and can be combined.

Per-Node Multithreading



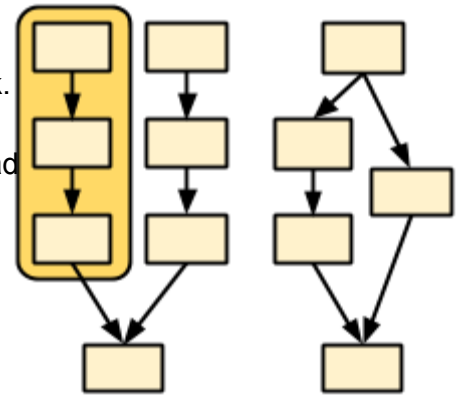
By per-node multithreading, we mean that each node independently runs its computation in a multithreaded fashion. In order to implement this scheme, nearly no specialization of the execution system data structures are needed. So long as the algorithm in the node is efficiently parallelizable, this might be a viable way to get a performance boost from multithreading.

A node doesn't typically need to synchronize or inform the system in any way that it intends to run its algorithm in multiple threads. It is also free to use any multithreading infrastructure that is deemed appropriate, for example Intel® TBB or OpenMP®, or the system's own implementation. The advantage of using the system's implementation is that it can coordinate the total number of threads in flight and can help avoid oversubscription.

In practice, the majority of our nodes run in a small amount of time and don't lend themselves to efficiently multithreading. While some particularly heavy nodes are multithreaded in this way, most of the callbacks in our system do not use this mechanism.

For most of the nodes in our system, per-node is too fine grained. We next look at a larger task unit.

Per-Branch Multithreading

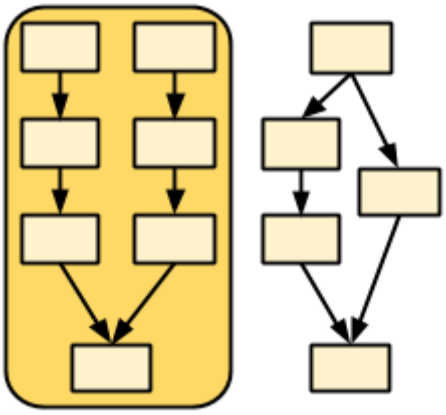


Another strategy is to launch a thread per branch in the network. Once the executor reaches a node that needs multiple inputs to be satisfied, each input branch can be launched in its own thread and can execute in parallel.

In order to implement this scheme, the executor would have to be specialized as well as the scheduler.

The amount of work that can be done in each thread here is much better, provided that your network has a lot of branches, each of which has a significant enough amount of work. This assumption does not always hold in our networks, and we find that often the largest amount of work is along a linear spine, and the branches are relatively cheap in comparison.

Per-Model Multithreading



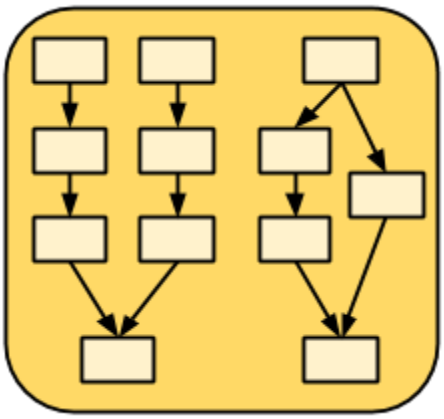
An even larger level of granularity is to run each model on its own thread. This is fairly straightforward to implement. The scheduler first finds all the disconnected subgraphs in the network, and the executor launches a thread per subgraph.

This is a very good amount of work to do per-thread, and generally works well. It runs into a couple of hurdles.

The first is that many of the scenes in our movies are animated using a small number of characters loaded at any one time. Smaller, say, than the typical number of cores found in modern hardware. This means that some cores can sit idle, which is not desirable.

Secondly, characters are rarely independent. They often have constraints among them. These constraints limit the parallelism between them, decreasing the system's potential performance.

Per-Frame Multithreading



Up until now, the strategies we've discussed don't take advantage of any domain-specific knowledge. We could employ these techniques for the evaluation of any data flow network. If we consider that our software's primary goal is animation, we find a whole new dimension along which to parallelize: time.

By allowing the system to compute multiple frames concurrently, we can significantly boost the performance of very common workflows. This approach lets us design a system where thread synchronization is not necessary at all while the concurrent tasks are running. This approach also introduces its own set of challenges that are due to its asynchronous nature.

We call this feature of our system "Background Execution", and that is the multithreading strategy that we found most successful for us and that we will discuss in more detail.

Background Execution

Background execution is the ability of our system to schedule frames to be computed in asynchronous background threads. This feature allows users to continue working while frames are computed.

The system architecture allows for easy implementation of the feature in its most naive incarnation: grab the frames that you need to compute in parallel, create an executor for each thread that you have available, and start doling out one frame per task. This approach leads to several practical issues that need to be considered when embedding in an interactive application.

User Interaction

A typical use-case is that a user will change an attribute value and then scrub on the timeline to see the effect the change has had on the animation. We would like to use the Background Execution feature to make sure that the computations of neighbouring frames have completed by the time the user scrubs to them. So as soon as the user stops modifying the value, background threads are kicked off to compute the frames that have become invalid due to the attribute change operation. How these frames are scheduled is discussed below. By the time the user scrubs over to the new frames, the idea is that the system would get cache hits for the frames and be able to return much quicker than it would have otherwise.

While the background threads are actively computing, the user is free to modify the values again, possibly invalidating the existing computations. The user's work can't be delayed by what the system is doing in the background, and we'll discuss interruption policies below as well. The main takeaway is that it's unacceptable for the system to pause or hitch due to work that the user is not expecting: avoid performance surprises.

Along the same lines, providing users with feedback that work is happening in the background can inform them of what to expect in terms of performance. For example, a scheme where you see the animation fill in in the background, helps you predict that moving to a filled in frame will be faster than moving to a frame that has not yet been processed. The feedback is also valuable for those trying to debug the system. When things are working well, the user generally doesn't have to pay too much attention and can just work undisturbed.

Frame Scheduling

In order to effectively schedule the next frames to be computed, you would need to be able to accurately predict what the user will do next. For example, if the user wants to play the frames in order after making a modification, then you would want to schedule them in ascending order. If, on the other hand, the user will frame back and forth around the dirtied frame, then you will want to schedule the frames to bloom out from the dirtied frame. But we don't know ahead of time which the user will do. We would prefer not to make this an option to the user because the user is only in a marginally better position to make this decision. The choice is not necessarily known to the user a priori either.

Some of the possible approaches are:

1. Pick a scheme and stick to it (e.g. always ascending, or always bloom out).
2. Base the decision on how the frames were dirtied. For example, if they were dirtied in a spline editor due to a tangent change, then bloom out; if it was due a change in the knot value, then use the ascending order scheme. The editor in which the change originated can also be a clue.
3. Keep a record of the user frame changes and match the dirty frame schedule to the pattern observed in past frame changes.
4. Combine #2 and #3

Interruption

The user is free to modify the state of the application while the background tasks are in flight. In particular, they are free to modify the state in such a way as to make the currently scheduled tasks invalid. One of the major problems we therefore need to deal with is interruption. It is critical that during common animator workflows, for example, changing spline values, or playing back, that the background tasks not interfere with the foreground work. Any hitch or delay can

immediately be detected by the user and can annoy users or, worse, can cause RSI. These kinds of hitches are particularly irritating because users can't predict when they will occur or how long they will last.

We don't explicitly kill threads (e.g. through `pthread_cancel` or `pthread_kill`) because that is a problematic approach[6]. For starters, most of our code is not exception-safe. We also felt that tracking down resource cleanup problems that resulted from abrupt interruption of threads would be time-consuming and constant source of bugs. So we decided to avoid it altogether.

We also didn't want to burden clients with having to check for an interruption flag. That approach is problematic as well, as some clients may either perform the checks too often or not enough.

The system therefore completely controls interruption and checks for interruption after each node has completed. Since our nodes are typically fast, that granularity seems appropriate. Unfortunately, we occasionally have nodes that do take longer to run and waiting for those nodes to complete before interruption is unacceptable for some workflows. Although we may want to support a cooperative scheme in the future, our experience so far has been that it is always better to avoid the long running computation altogether. No one node can be allowed a very long runtime if we expect to stay within our 42 ms budget.

While animators are working, we cannot afford to wait for interruption at all. Animators can change values several dozens of times per second, and cannot be interrupted. Luckily, these common, high frequency tasks, do not change the topology of the network, and therefore do not impose a requirement for our system to actually wait for the background tasks to stop before changes are made. We take advantage of this fact by not interrupting at all during these workflows. Instead, we remove the remaining tasks and set a flag telling the threads to throw away their results once the next node is done computing. With this approach, animators are no longer bothered by what is going on in the background. As mentioned earlier, animator workflows that do not modify the network, such as adjusting spline values, adjusting tangents, inserting and deleting knots, and so on, do not require us to join with the background threads. On the other hand, tasks that do alter the network topology, for example, setting new constraints or adding new deformers, do require us to stop and wait for the background threads. For these less frequent tasks, we rely on node-level granularity of interruption to be sufficient.

Constant Data

Since we would like to avoid synchronization among threads while the computations are in flight, each thread gets to manage its own network data. We quickly notice that there could potentially be a lot of computations in common among the threads. Time-independent computations will all

[6] http://www.drdobbs.com/parallel/interrupt-politely/207100682

yield the same results in all threads. These computations also tend to be slow and memory intensive. They tend to be the kinds of computations that set up large projection caches or acceleration structures that don't change with time Launching all threads to compute this same data at the same time saturates the bus and decreases the system's throughput. We would like to avoid running computations redundantly, and yet maintain the lockless nature of the execution engine.

Our solution is to launch a single thread, which we call the starter thread. The starter thread's job is to compute all the constant data in the network and make it available to all the other threads. The starter thread uses a starter executor to compute the constant data, then schedules the background frames and kicks off all the concurrent threads. For each concurrent thread, we create a new executor that is allowed to read from the starter executor. This multiple-reader scenario is supported without needing to lock since the starter executor's data is read-only while the other frames are being computed. In addition to speeding up the background threads, this scheme also significantly reduces the amount of memory needed to run multiple threads and allows our memory consumption to scale much better with the number of available cores.

The starter thread allows the main thread to be free to respond to user interaction. It also runs infrequently: once at load time and only again when time-independent data is invalidated. We further decrease the amount of time this thread needs to execute by allowing it to also locklessly read constant data (i.e. data that can't be invalidated through the model's animation interface) that is computed only once per model.

Future Work

We're excited to keep experimenting with multi-core architectures and multithreading strategies.

The system, since the start, has been designed to support a strip-mining strategy for multithreading where all the requested vectorized elements are processed in parallel in equal chunks among the available number of cores. We haven't yet taken advantage of that approach but we would like to. This strategy would complement Background Execution and would be used for improving the speed of single frame evaluations. We feel that this approach has the potential to scale well with the ever increasing number of cores available at user desktops.

A small modification to our Background Execution infrastructure would allow us to perform predictive computations. In other words, while a user is continuously changing a value on an input, we can schedule computations for input values that the system predicts the user will ask for next.

We'd like to experiment with writing a GPU Executor and a VRAM Data Manager, for example, to

run all or parts of our rigs on the GPU. Similarly, we'd like to explore the benefits of Intel's Xeon Phi™ coprocessor and see how we might adapt our architecture to get the best utilization on a very large number of cores.

We'd like to continue developing tools to help users introspect and profile their rigs. We'd also like to adopt or develop more and better tools to help us better identify performance bottlenecks and correctness issues.