Abstracting rigging concepts for a future proof framework design

Jesus R Nieto
DNEG
crn@dneg.com

Charlie Banks
DNEG
cba@dneg.com

Ryan Chan
DNEG
ryc@dneg.com

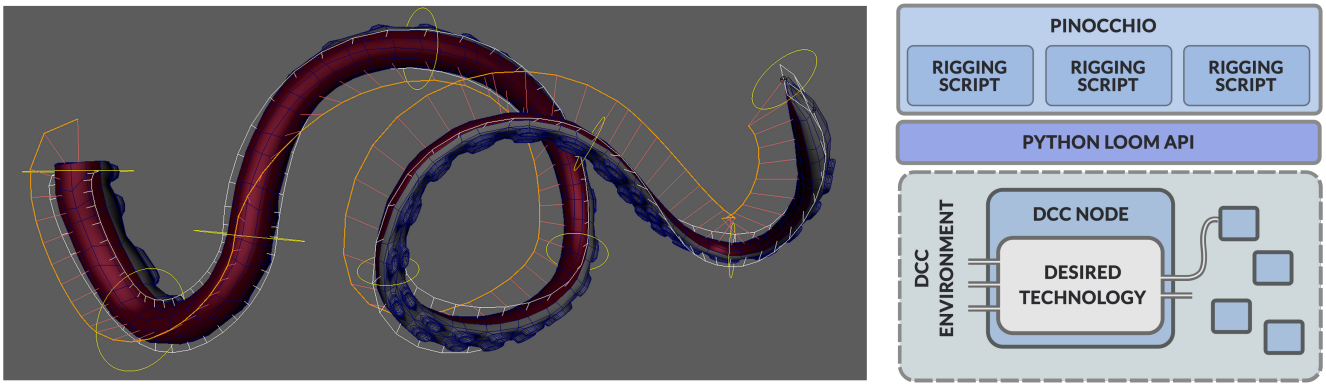

Figure 1: (left) A tentacle solver prototype built assembling simpler pieces in Loom, it's being used in a some shows at the moment; (right) Schema of the abstraction layers defined in Loom.

ABSTRACT

Several years ago DNEG set out to build Loom, a new rigging framework in a bid to improve the performance of our Maya animation rigs. This talk is an update on its development in the light of the discontinuation of its original evaluation back-end, Fabric Engine. In particular, we describe the design choices which enabled us to achieve a DCC agnostic rigging framework, allowing us to focus on development of pure rigging concepts. Also how this setback prompted us to extend the framework to properly deal with the deformation side of rigs, targeting memory efficiency, GPU/CPU memory interaction and high-end performance optimizations.

CCS CONCEPTS

• **Computing methodologies** → **Computer graphics**; **Animation**;

KEYWORDS

Rigging, Framework, Deformation, Software Engineering

DigiPro '18, August 11, 2018, Vancouver, BC, Canada

ACM ISBN 978-1-4503-5895-8/18/08…$15.00
https://doi.org/10.1145/3233085.3233088

ACM Reference Format:
Jesus R Nieto, Charlie Banks, and Ryan Chan. 2018. Abstracting rigging concepts for a future proof framework design. In *DigiPro '18: The Digital Production Symposium, August 11, 2018, Vancouver, BC, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3233085.3233088

1 INTRODUCTION

Following on from the work presented in [Nieto et al. 2016], over the past two years we have been developing a rigging framework focused on delivering performance, flexibility, agility and scalability. Although our original focus was performance, thanks to the introduction of parallel evaluation in Maya 2016, this was no longer a critical factor. Instead, we shifted our attention to agility: we wanted to be able to deploy new features incrementally and achieve fast turnaround times to address short-term show requirements, while still retaining the ability to experiment with up-and-coming technologies.

To achieve this, a key element was to design a platform-agnostic framework that allows us to develop meaningful rigging concepts and foster good rigging practice independently of the underlying implementation details of both the host DCC and the evaluation back-end. Such abstract rigging concepts allow us to describe a rig in terms of behaviours and mathematical relationships, as a series of interrelated operators modifying datasets.

Each of those operators can be represented as a node in a graph and the relations between those operators as edges in the graph. Several operators and their relationships can be grouped into a single, complex operator, culminating with an entire rig as a single operator. This concept of modularity, common to many applications, is the foundation of our rig definition. We call this framework Loom.

2 RIGGING CONCEPTS DEVELOPMENT

Based on this we can design a library of operators for use in our rigs, from fundamental mathematical tools like matrix multiplication to more advanced operators such as a tentacle solver (Figure 1). Each of these operators is defined as a node with in/out connectable attributes, which can also be interpreted as a programmatic function and its parameters, with the data types associated with them. This definition forms our first layer of abstraction. Rather than working with locators or joints, we define kinematic relations in terms of matrices, mathematical constraints and complex solvers.

Operators can be implemented independently of any DCC, using any underlying technology, and later be integrated into a particular DCC if we provide the relevant data type conversions from that DCC to our framework. This modular approach allows us to develop different operators little by little and integrate them with other elements in the DCC until the whole framework is in place. Unfortunately, the conversion of data comes at a cost as it increases evaluation time, but the modularity greatly facilitates code maintenance, reusability and automated testing.

The creation of rigs in a production environment needs to be scriptable. We use our in-house modular rigging system - Pinocchio - in order to do so, but it previously relied entirely on Maya commands, so portability was an issue. We therefore added another layer of abstraction written in Python. It provides an API for creating Loom graphs, so that rigs can be built in any DCC from the same description.

These two layers of abstraction combined have allowed us to survive to the discontinuation of Fabric Engine, our evaluation backend: while the development team works on migrating operators from Fabric to a new technology, artists can still work using the abstract rigging concepts provided by Loom. Figure 1 shows the final structure of our framework. Pinocchio will be using the python Loom API to create graphs in a similar way [Rose et al. 2013] does.

2.1 Data Flow Dependency Graph (DFDG)

Until this point we have described our previous work in [Nieto et al. 2016]. We now propose to look at the rig as a Data Flow Graph where the edges define how the data is traveling along the graph. There are three kind of nodes: data sources which inject data into the graph (eg, geometry imports...), operators that process that data, and destination nodes which render or export it. This flow of data establishes the basis of a procedural rig.

The data can be accessed or previewed at different points of the process, as a snapshot of the input/output of a given node. This is quite similar to the concepts behind Houdini's node graph but quite foreign to Maya, which makes us quite optimistic as we're forming a bridge between the different applications. A less encouraging point is that it has serious implications in the way those rigs are designed and how it affects the artists' workflow. In the next section we briefly discuss the rigging mind-set required to use these concepts.

Another important aspect is the granularity of the data as it travels along the graph, this is defined by the Data Flow Dependency Graph. From this graph we can really optimize how the large data sets are processed later at implementation level. Take for instance the geometries, the basic ingredient in any deformation graph, we might want to manage the data differently for operators that affect only points, normals or any other element defined in the geometry.

At this point we can further break down the definition of a complex data structure into two types of components: immutable (static data) and mutable (dynamic data). Given an object, for instance a polygon mesh, its static components will be centralised and shared across nodes in the graph (topology and uvs), while its dynamic data will be flowing through the graph and represent different states of the object as versioned copies of the dynamic data (point positions). This approach enables a whole range of optimizations on the use of memory and performance.

2.2 Tackling the Invisible Transform

Not all riggers, whether junior or senior, are familiar with mathematical concepts. This is problematic when approaching rigging with Loom as, for instance, it favours the use of matrices instead of Maya transforms. It can be hard for an artist to visualise a matrix in their scene when there is no Maya shape to represent it in the viewport.

The workflow architecture of Maya differs to ones such as Nuke and Houdini precisely in this aspect. Maya is built with the mindset that we want to see and have access to all items in the scene at any given time, this is provided to us by having different types of shape nodes to represent certain objects (locators, meshes, etc). In software like Houdini, these objects are purely represented by a flow of data from node to node, and Houdini provides us with the node editor to allow artists to select which part of the graph they want to visualise in the scene. When they are finished with their work, all they want to do is select the final output node and visualise the final result in the scene, they don't care about the intermediate steps any more.

With Loom, we are attempting to move Maya away from the "see everything" workflow and more towards the "see only what you want to see" workflow, but without the right toolset it can be difficult for riggers to adjust to this new design. We developed a node graph editor as the first step in bridging the gap that riggers need to cross in order to adopt this new way of working.

3 DEFORMATION FRAMEWORK

In our previous work we applied Loom mainly to solve the kinematics side of the rig. As a following step we applied the same concepts to deformation but unfortunately Loom fell short, mainly when Maya provided a strong GPU deformation pipeline in 2016 [Autodesk 2016].

There are several factors that makes Maya fast when deforming on the GPU, but much of its success comes from the fact that its heavy geometry data is already on the GPU when the render step kicks in. Communication between CPU and GPU is a serious bottle neck, therefore a good mechanism to make it efficient and easy to use is a big requirement for this kind of frameworks.

We would like to share our conclusions on this aspect and how we achieved better performance than Maya on GPU but only using CPU and why we consider this to be our main solution so far, despite of the promising hopes GPU processing may suggest. Freeing ourselves from Fabric Engine made this exploration possible and allowed us to put on place a robust deformation framework based

on the concepts described in the previous sections, achieving better performance and more efficient use of memory than our previous version of Loom and Maya.

Figure 2 shows a comparison of a series of tests ran on different implementations skincluster deformer. We chose this deformer because of its simplicity and fundamental use in rigging. We implemented it in OpenCL [Stone et al. 2010] (not particularly optimized), CUDA [NVIDIA 2012] and C++ using a combination of TBB [Pheatt 2008] and ispc [Pharr and Mark 2012]. These tests were run on an Intel Xeon E5 3.1GHz machine (32 logical CPUs) with Nvidia Quadro K4000 graphics card.

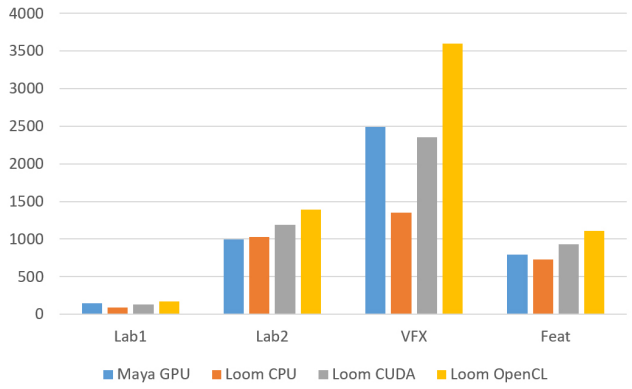

Figure 2: Tests ran on the different variants of a skincluster node. The Y axis shows the time in microseconds that the node takes to evaluate on Maya on each frame.

We ran four tests trying to show different scenarios that we might find in our productions: Two laboratory cases: Lab1, on a small mesh (1.5K verts) with 2 joints, and Lab2, on a big mesh (90K verts) with 300 joints, as well as two real cases: a VFX model (348 K verts with 146 joints) and Feature Animation model (37K verts with 156 joints). We avoided showing Maya-CPU times in the comparison because they are significantly higher, in the vfx case it's even an order of magnitude higher.

4 CONCLUSION

Beyond its feature set, performance and user adoption, the success of a particular tool can also be measured by its ability to weather adverse events. Our approach turned out to be a life-saver when one of its most fundamental components was discontinued. In addition, it improves flexibility, allowing us to develop and evolve the system in incremental steps without affecting the end user. It also allows riggers to produce scripts directly derived from pure rigging concepts rather than DCC specifics. Compared to our previous workflows, they are cleaner, more modular, build quicker and result in graphs that run faster in parallel evaluation environments.

However it does require riggers to assimilate a series of non-trivial concepts - we are working on ways to make that learning process smoother.

REFERENCES

Autodesk. 2016. Using Parallel Maya. (2016). http://download.autodesk.com/us/company/files/2016_Extension2/UsingParallelMaya.pdf

Jesus R. Nieto, Theo Facey, and Sylvain Brugnot. 2016. A Flexible Rigging Framework for VFX and Feature Animation. In *ACM SIGGRAPH 2016 Talks (SIGGRAPH '16)*. ACM, New York, NY, USA, Article 46, 1 pages. https://doi.org/10.1145/2897839.2927463

NVIDIA. 2012. Kepler GK110 Whitepaper. (2012). http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

M. Pharr and W. R. Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. 1–13. https://doi.org/10.1109/InPar.2012.6339601

Chuck Pheatt. 2008. Intel®; Threading Building Blocks. *J. Comput. Sci. Coll.* 23, 4 (April 2008), 298–298. http://dl.acm.org/citation.cfm?id=1352079.1352134

Rachel Rose, Mike Jutan, and John Doublestein. 2013. BlockParty 2: Visual Procedural Rigging for Film, TV, and Games. In *ACM SIGGRAPH 2013 Talks (SIGGRAPH '13)*. ACM, New York, NY, USA, Article 8, 1 pages. https://doi.org/10.1145/2504459.2504469

J. E. Stone, D. Gohara, and G. Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (May 2010), 66–73. https://doi.org/10.1109/MCSE.2010.69