Developing a Unified Pipeline with Character

Andrew Kaufman[*] Image Engine
John Haddon Image Engine
Ivan Imanishi Image Engine
Lucio Moser Image Engine

© 2009 TriStar Pictures. All Rights Reserved.
© 2011 Universal Pictures. All Rights Reserved.
© 2012 Universal Studios. All Rights Reserved.

Figure 1: *Prawn from* District 9 *(left), Juliet from* The Thing *(middle), and the Thug from* Battleship *(right) are a sampling of creatures created using our constantly evolving pipeline.*

Abstract

Photo-real digital characters and creatures have become an integral component of modern feature films. As audiences become more accustomed with visual effects, the demand for seamless reality is ever increasing. This in turn increases complexity of all aspects of character design, from the build phases: modelling, look development, and rigging; to the execution stages: animation, fx, and lighting. We sought to develop a vfx pipeline that was highly specialized in creature and character work, while maintaining maximum flexibility for the wide-ranging scope of work that accompanies it.

This paper explains the evolution of our creature pipeline, focusing on look development/lighting and asset management. We detail specific software design choices, highlight successful intra and inter-studio collaborations, and discuss our current development efforts towards scalable, maintainable tools for the ever increasing complexity involved in visual effects production.

Keywords: pipeline, asset management, procedural, node graph

1 Introduction

As a small to midsized studio, we don't have the resources to explore every avenue of development that creature work may require. Our tools must work within our computing and storage limits, and must be polished in fast, fluid development cycles, done in close collaboration with artists and supervisors. We aim for application independent development whenever possible, reusing as much common code as possible, and sharing development initiatives between all aspects of the pipeline. We have leveraged several common open source libraries, including Boost, Imath/OpenExr, and TBB when building our software.

By tending towards application independent tools, we are able to future proof our efforts to some extent, not relying on any one commercial (or proprietary) application. We also get the benefit of focused optimization impacting our entire pipeline. One developer can improve file IO across the facility by optimizing the object serialization code. As we adhere to a strict unit testing regiment, we can make optimizations like this without fear of disrupting the pipeline. Issues are also easier to debug, since so much flows through the same code in the end.

Such a pipeline requires highly planned, object-oriented software design, with meaningful class inheritance, interfaces, and architecture. It also requires a predictable, extensible API which can be scripted by TDs who may or may not have a background in software engineering. As such, the backbone of our pipeline is a suite of core c++ libraries, which we make available in python via a thin layer of bindings. Our higher level tools are developed in either c++ or python, depending on project specific prioritization of development speed vs computational efficiency.

We have released our cross-application framework for computation and rendering as a suite of open source libraries [Cortex 2007], with hopes of encouraging collaboration within the broader visual effects community. We are also developing an open source application framework for visual effects production [Gaffer 2011], which adds a mutlithreaded node based computation framework and a QT based user interface framework for editing and viewing node graphs. This application framework currently forms the basis of our Asset Management system, and we are actively developing it for further use throughout our pipeline, including look development, shader authoring, and lighting. In the future, we would like to use it for fur, crowds, layout, render farm management, and even creation of client outputs.

The core of Gaffer has similarities to other multithreaded node graph projects, like LibEE [Watt et al. 2012], but differs in that it aims to provide a generic node graph framework, applicable to all aspects of visual effects, rather than focused optimization for rigging and animation. In that sense, Gaffer seems similar to Crom [Cournia et al. 2012], with the key difference being the open architecture, encouraging both intra and inter-studio collaboration.

Here we present some of the design decisions that drive our creature workflow, from modelling through to lighting. For the purposes of this paper, our discussion will be in the context of a modified VFX pipeline (Fig. 2). We first describe the Renderer abstractions provided by Cortex (Sec. 2), followed with an explanation of how we use this framework in our Look Development/Lighting pipeline (Sec 3), including our fur and hair system (Sec. 3.1). Next we describe the design of our Asset Management system (Sec. 4).

[*]e-mail: {andrewk, john, ivani, lucio}@image-engine.com

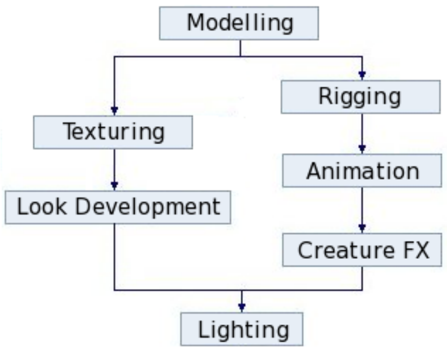

Figure 2: *A simplified VFX Pipeline for purposes of this paper.*

We follow with some rules of thumb on collaborative development (Sec. 5), and finally discuss (Sec. 6) limitations of our current pipeline, and our ongoing efforts towards a hierarchical, node based approach to production pipelines.

2 Renderer Abstraction

One of the key features of our look development and lighting pipeline is the use of delayed evaluation procedurals [Pixar 2005] to represent all geometry. As artists in these disciplines rarely need direct access to the geometry data, the procedurals help keep their working files light weight, while still providing visual representation of the geometry, along with specific interaction techniques, including shape selection and constraining external objects to bounding boxes and centroids (Fig. 3).

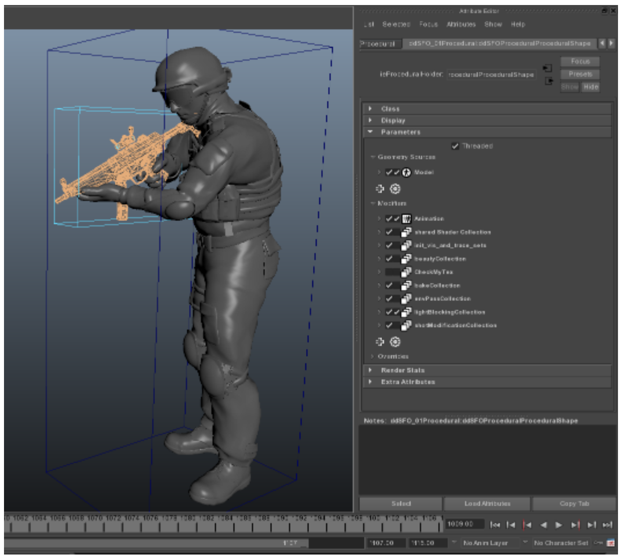

Figure 3: *A ModularProcedural loaded in Maya*

This visual feedback is a primary advantage of our renderer abstraction. We have one code path that defines our procedural. That code is rendered first by OpenGL, in an interactive 3d application like Maya, Houdini, or Gaffer. When we want a final render, we inject our unexpanded procedural definition into a file (.rib, .ass, .ifd, etc.) and the same code that was previously processed by our OpenGL Renderer is now processed by a Renderer implementation for the associated 3rd party software at render time.

The Renderer base class provides a unified means of describing scenes for rendering, regardless of the specific backend software. Its interface is modeled closely on OpenGL and Renderman, with an attribute and transform stack. We currently have derived Renderer implementations for OpenGL, Renderman, Arnold, and Mantra, with OpenGL and 3delight (which is Renderman compliant) seeing the most use in production.

In order to hand objects to the renderers, we need a generalized interface for describing things that are renderable. This is provided by the Renderable and VisibleRenderable classes (Fig. 4). VisibleRenderables are expected to be objects which are visible in final rendered images, while Renderables just change some part of the renderer state (such as an attribute).

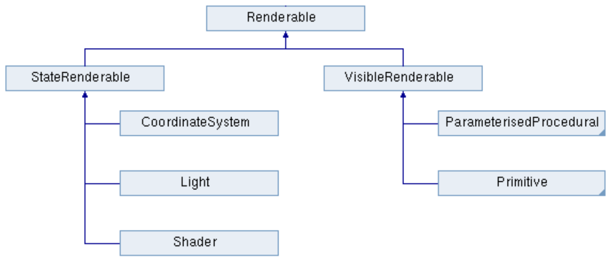

Figure 4: *Partial class hierarchy for Renderable objects*

2.1 Primitives

The most common VisibleRenderable is geometry, which is described to the renderer in the form of Primitives. Meshes, curves, particles, nurbs, and even images all derive from the Primitive base class, which means they may hold PrimitiveVariables: values which vary over the surface of the Primitive and can be used by the renderer to define various aspects of its appearance. The most common examples would be position, normals, and UVs for geometry, and color channels for images. Primitives can be passed to a renderer directly, using a simple script (Algo. 1), or they could be passed in a delayed manner via a procedural.

Algorithm 1 Render a Mesh in OpenGL from Python

```
from IECore import *
import IECoreGL
m = MeshPrimitive.createBox( Box3f( V3f( 0 ), V3f( 1 ) ) )
r = IECoreGL.Renderer()
r.setOption( "gl:mode", StringData( "immediate" ) )
r.display( "/tmp/output.exr", "exr", "rgba", )
with WorldBlock( r ) :
    r.concatTransform( M44f.createTranslated( V3f( 0, 0, -15 ) ) )
    m.render( r )
```

Each primitive type has an associated evaluator, which we use to query the topology and variable data. PrimitiveEvaluators permit spatial queries on primitives, such as determining the closest point, retrieving the position from a given UV coordinate, signed distance, intersection points, etc. The evaluators provide a Result object, which may be used for further evaluation of other variable data relating to the same query (Algo. 2). Evaluators guarantee thread safety provided each thread is using a unique Result object. Some evaluators, such as the MeshPrimitiveEvaluator, lazily compute and store bounded KD trees [Bentley 1975] using both triangle bounding boxes and uv bounds to accelerate concurrent queries.

Algorithm 2 Find the normals at the closest point on a mesh

```
from IECore import *
m = MeshPrimitive.createBox( Box3f( V3f( 0 ), V3f( 1 ) ) )
mpe = PrimitiveEvaluator.create( m )
r = mpe.createResult()
if mpe.closestPoint( V3f( 1, 2, 3 ), r ) :
    geometricNormal = r.normal().normalized()
    shadingNormal = r.vectorPrimVar( m["N"] ).normalized()
```

2.2 Manipulation on the fly

In addition to geometry evaluation and rendering, we have a suite of operators for render-time manipulation of the geometry by merging, distorting, filtering, adding or recalculating variables, etc. These all derive from a common base class, PrimitiveOp, which is responsible for defining whether or not the object is modified in place or as a copy, and preparing the input primitive for modification. The simplest example would be triangulating a mesh and adding normals (Algo. 3) before handing off to the renderer.

Algorithm 3 Triangulate a Mesh and Calculate Normals

```
from IECore import *
m = MeshPrimitive.createBox( Box3f( V3f( 0 ), V3f( 1 ) ) )
TriangulateOp()( input=m, copyInput=False )
MeshNormalsOp()( input=m, copyInput=False )
```

A more complex case could form the basis for a simple fur system. The PointDistrobutionOp will create a PointsPrimitive of blue noise [Kopf et al. 2006] covering the input MeshPrimitive. The points can be used as seeds (or follicles), from which curves are extruded. Further use of PrimitiveOps could deform the curves according to physical laws or from user input for grooming. We provide example code for such a fur system [Cowland 2011], but we won't include it here for brevity. In practice, our production fur and hair system (Sec. 3.1) is significantly more complicated, building on top of the foundations provided by Cortex.

2.3 User Parameters

Automated UI generation is made possible by our ParameterHandlers. We have a suite of registered ParameterHandlers for each host application (Maya, Houdini, Nuke, Gaffer), and common parameter types have associated creation mechanisms and UI code for hosting natively in each 3d application.

Dynamic parameter addition is accomplished through a mechanism for embedding Parameterised code snippets within an existing parameter. This allows one parameter to easily nest a set of parameters and computations, while exposing the hosted parameters publicly. Any of the exposed parameters may load other parameter/code snippet pairs, allowing artists to build dynamic trees of computation with user-exposed parameters (Fig. 5).

2.4 Automated Efficiencies

In addition to the visualization and interaction described in the beginning of Sec. 2, our procedurals provide an automated form of multithreaded evaluation. Procedurals are able to spawn sub-procedurals, and if the reentrant attribute is set, these sub-procedurals will be spawned on separate threads using TBB [Reinders 2007].

Prior to our latest round of productions, we added the concept of hashing to our base object class and are using the Murmur Hash implementation [SMHasher 2010] . This means that all objects in our libraries are hash-able, including procedurals, primitives as a whole, their topology, and each of their variables.

We have been able to leverage this hashing to perform automatic instancing at render time. Rather than having artists define instances manually, they can just pass their data along and the renderers will recognize geometry that has been seen before. We currently leverage automatic instancing in our OpenGL and Renderman layers, and are investigating adding Arnold support in the future.

Recent developments to our OpenGL layer have made use of our general purpose Least-Recently-Used (LRU) Cache [Coffman and Denning 1973] to further optimize redraw events. Any primitive that has been converted to OpenGL goes through our CachedConverter, and any subsequent requests for conversion will simply read the existing GL buffers from the cache. This is just one example use of our LRU Cache, and in practice, it is used transparently by many of our repetitive operations, including file I/O.

3 Look Development in Production

When we started building our lighting pipeline, our procedurals were built as matching pairs of python scripts and compiled shaders, hosted on a custom node in Maya. A look development TD would script the procedural and shader in unison, compiling and installing as a software engineer might do. This approach was pioneered for *District 9* and improved on *The Twilight Saga: Eclipse* and *Stargate: Universe*.

As our creatures became more intricate, this approach became infeasible. For *The Thing*, we moved towards a more dynamic workflow, allowing look development artists to interactively add, remove, and re-order pre-defined mini shaders (co-shaders) onto the procedural live in Maya (Fig. 5).

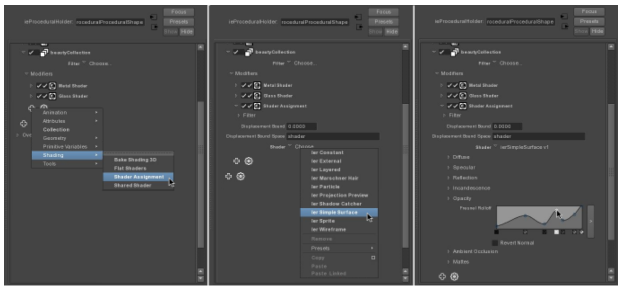
ieProceduralHolder: roceduralProceduralShape
Filter Choose...
Displacement Bound 0.0000
Displacement Bound Space shader
Shader ieSimpleSurface v1

Figure 5: *Adding a shader assignment (left), setting it to Simple Surface (middle), and changing the Fresnel rolloff (right).*

We call procedurals built in this manner ModularProcedurals, and they have three main component types in our pipeline. First is a list of geometry sources, used to load geometry from disk, generate it on the fly, or feed it in from live geometry directly in Maya. The second is a series of modifiers. They may perform generic geometric operations (Sec. 2.2), apply an animation cache, and be used for shader assignment and filtering. Finally, we allow in-shot overrides of shading parameters. In practice, we use one ModularProcedural per character, and provide a mechanism for sharing global shader collections between them.

While Lighting artists cannot manipulate the procedural geometry directly, we provide a simple transform modifier for moving individual components, as well as visibility filters, so last minute in-shot updates can be made. In addition, the context menu on a procedural can be used to convert any selected components to live Maya geometry. This is accomplished using a special type of Renderer subclass, called a CapturingRenderer. We use the CapturingRenderer in Houdini as well, in order to perform on the fly conversion of procedurals to Houdini geometry in a SOP node chain.

ModularProcedurals were key to our success on *The Thing* (Fig. 6), *Battleship*, and several smaller projects involving digital doubles, creatures, and hard surface vehicles. The asset complexity and production timelines for these shows would not have been achievable without the interactive shader construction made possible by the ModularProcedurals. For our current productions, we again took a leap forward in complexity, and ran into performance and usability issues with the ModularProcedurals. We discuss our current and future plans to address these issues in Sec. 6.

Figure 6: *Edvard-Adam from* The Thing *received a VES nomination for this transformation sequence.*

3.1 Fur and Hair

Our proprietary fur and hair system was first developed for *The Twilight Saga: Eclipse* and has become a staple tool, seeing use in almost all of our productions. The entire system exists as a modifier within the procedural, and is itself modular as well, the main stages being seed, grow, groom. The underlying system manages patches of fur independently. Each patch is a sub-procedural, rendered on a separate thread.

Seeding happens slightly differently than described in Sec. 2.2. The same blue noise code is used, applied to a patch at a time, in UV space. These seeds are extruded straight out from the mesh by a length modifier, then interactively groomed using custom paint tools built within Maya. The painting is stored directly on the procedural as either float or color maps, and controls the effects of each grooming modifier. We have a suite of common modifiers, such as frizzing, mutating, clumping, etc [Bredow 2000] which grooming artists use to perfect the look of the fur or hair.

We exploit the modular nature of our procedurals for fur clumping as well. We define clump centers and directions by simply seeding another fur system at a lower density, and clumping the main hairs towards this secondary groom. These clump curves can themselves be styled with any of the available fur modifiers, including further clumping. While we can get interesting effects with this approach, this sort of nearest neighbor clumping is limited to producing essentially conical clumps, as the target is a single curve.

More complex shapes can be realized (Fig. 7) by instead targeting the nearest isoparm on a user defined ribbon. Our "Ribbon Clumping" approach allows artists to draw clumping lines directly on the growth mesh with our paint tools. The resulting map is then thinned to a series of pixel wide lines [Cychosz 1994], which are traced to produce curves in UV space. The object space positions of these CVs then seed another fur system, and the resulting growths effectively form a ribbon that can be groomed, allowing cracks, mohawks, forks, and other irregular clump shapes to be achieved.

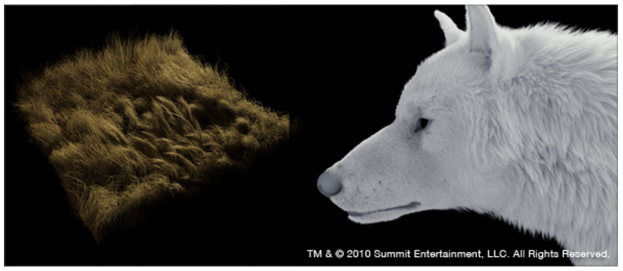

Figure 7: *Various grooming effects (left) and the final groom (right). Recursive clumping was essential in creating the final look.*

4 Asset Management

We have developed a proprietary asset management system, called Jabuka, which has gone through significant changes over the years. In it's origins, Jabuka was developed by our Asset Lead, as a mix of database and filesystem operations, and the UI was built entirely within Maya. We successfully used this system for several projects, but ran into key design issues as complexity grew.

For Jabuka's redesign we decided to take a node based approach to asset management. The hierarchical approach of our original system was too rigid, and the flexibility of Gaffer's connectable node graph provided the ideal combination of structure and freedom we desired in an asset management system. In addition, Gaffer's QT-based UI framework was ideal for portability between applications.

4.1 Connectable Entities

The core objects in Jabuka are called Entities, which are Parameterised objects, like Ops and Procedurals (Sec. 2), allowing them to leverage existing UI code for user input and basic node representation (Sec. 2.3). Assets are VersionedEntities (Fig. 8) and are used to organize a set of Components, which are themselves StorableEntities, meaning that they are versioned, store some information on the filesystem, and provide automated compatibility tracking.

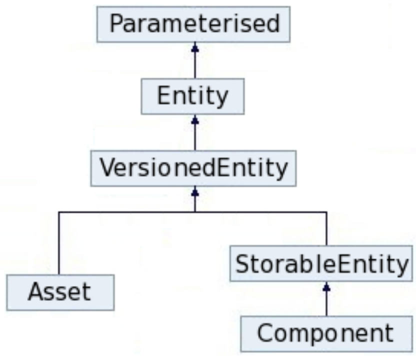

Figure 8: *Partial class hierarchy for Jabuka Entities*

All entities are represented as Gaffer nodes, and can be connected together either manually or through scripting, just as any geometric or shading node might be. There are several advantages of these connections. First, they allow for compatibility tracking and version management. Any given component can follow its input connections to determine exact versions of every other component that is required to re-create it (Fig. 9). As an example, we can look at a final composite and determine the model, textures, animation, camera, light rig, 3d renders, and HDRIs that went into it.

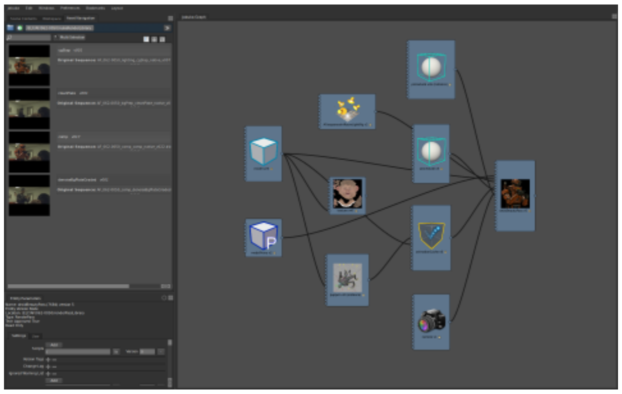

Figure 9: *Jabuka's graph view, showing a beauty render and its associated dependencies.*

This compatibility tracking is essential for disk cleanup and data management. Given overlapping production schedules, tight deadlines, and limited disk space, we need to be relatively strict about cleaning up our data. In the past, we left this to artists, who could easily forget which shots have the most data, let alone which data is safe to delete. Using connectable entities, we are able to automate the cleanup process. We set configurable rules controlling the number of per-instance versions of each entity type. When a version is a candidate for deletion, we use the connections to guarantee it will remain on disk as long as it is in use by any downstream component.

Connections can also be used to inform a component of all other components that need to be present in order for it to function correctly. This is of particular importance to our use of procedurals for lighting. In our old system, the core system was hard-coded to know that a procedural has a model, textures, and animation, and that was all. In this node based system, we use the arbitrary connections to determine what a procedural contains. It could be models, an external shader, some special FX geometry, or anything else. The core system doesn't need to know what a procedural is, just that it needs these other things. The procedural itself is then free to define how it will use them.

We utilize connections for both asset instancing and general re-use of components. In the original system, components could only be reused from direct hierarchical parents. With connections, components can be taken from anywhere in the system. A single procedural could be instanced into any number of shots, having different models, textures, and animations connected to it. The procedural definition would remain the same each time, but the exposed parameters and connections are freely altered for any instance.

4.2 Inter-departmental Flow

A key aspect of asset management is data flow between departments. We use Jabuka to manage this flow, and to help provide quality control at each stage of our work. Artists set their role in Jabuka, which is stored in the preferences. Jabuka provides the common tools as single button actions, using the role to determine the most appropriate action.

Actions are really just special Ops that act on Jabuka entities. This allows us to leverage existing code for UI display and farm submission. Any improvements we make to those underlying features, such as automated parallelization or application of user presets, will be directly available in Jabuka. We can create any number of actions associated with entity types, with no need to update Jabuka's UI for each new action, and we can easily create show, sequence, shot, or user specific actions as necessary.

When an artist performs a checkout of some component (be it model, texture, animation rig, etc), the work becomes locked for others. When they publish it back to Jabuka, a set of verifications specific to the component and role are executed. Failing verifications issue errors and warnings, as configured by our Asset Lead or CG Supervisors. Publication events trigger notification emails to supervisors, for quality control and approval, and to downstream artists, so they know to check for the latest work.

Modellers, texture artists, rigging, and look development can all work on their respective components simultaneously. When they publish new versions, the required versions of the upstream connected components will be recorded. Anyone doing downstream work, for example an animator or lighter, would get their component (as a reference), and that process would recursively get all the required upstream components. The "get" action is quite abstract, and each class of component can define it as required. Any issues with upstream components would be marked in Jabuka's navigation interface (Fig. 10) and the user can hover to find details of each issue and to reveal the possible actions.

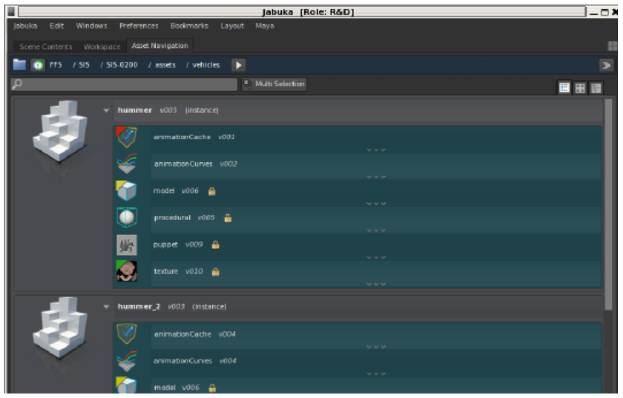

Figure 10: *The latest procedural for this asset has an invalid animation cache (red marker) and an acceptable warning with the model (yellow marker). The locks indicate read-only components.*

4.3 Extensibility

Jabuka is also highly configurable. The API is entirely exposed in python, so pipeline TDs and lead artists can automate any processes they deem necessary, on a show, sequence, shot, or user basis. Config files can be registered to component types, so that specialized behaviour can be achieved when using the default actions. This allows for show specific default values, or application specific node manipulation, without introducing transient code into Jabuka itself. Configs also shift the workload for show specific features from the core developers to people directly involved in production, who may need faster turnaround than can be achieved otherwise.

We have found this particularly useful in the case of animation renders. A CG Supervisor is able to script a config for a lightweight procedural, just loading a model and animation, ignoring textures and shaders. This allows animation preview renders to include full resolution bind geometry with final render subdivision and motion blur, at no extra effort to the animators themselves. They do not need to be familiar with the lighting pipeline, and instead simply rely on the automated config to import the appropriate components and tie them into this ad-hoc procedural at render time.

This ability to build deferred evaluation scenes programmatically also allowed us to automate our cloth simulations, greatly improving shot-turnaround for the Creature FX department (Fig. 2). When an animator initiates a cache, a series of FX caches are triggered, associated with connected downstream components, each feeding results into the next. During each stage of caching, the Maya scene from the previous stage is loaded and the shapes associated with the previous component are replaced with read-only data loaded from that component's cache. This allows for in-scene tweaks to the animation rig to be maintined for the offline caching, while still ensuring that intermediate simulations only run once, their cached results being used downstream.

This automatic layering of muscle, fat, and clothing stores intermediate results at every stage, which can then be finessed by Creature TDs. While we are interested in the possibilities of automated processes even further downstream, into Lighting and Compositing, we have not gone beyond Tech Anim in production, prefering manual quality control before handing off to Lighting.

5 Collaborative Development

Close collaboration between all parties was essential to manage the robust, yet fluid development process, and to ensure that the end tools are artist friendly and intuitive for technical and non-technical users alike. We discuss the essentials of our collaborative efforts, both in terms of internal decisions, and external partnerships.

5.1 In House

Internally, our successes have always involved small teams in constant communication, and our failures have come when interdepartmental communication was overlooked. Everybody in the development process should be aware of, if not directly participating in the efforts of their teammates. This helps expand understanding of the core issues, prevents duplication of efforts, and often inspires new ways of thinking about the problems at hand. The open nature of this working process also ensures that the feedback-develop-test cycle is smooth and accessible for those in production.

When we developed our Fur System, we had a CG Supervisor and two or three R&D programmers sitting together, talking through every step of the way. This provided constant, and consistent, hands on testing and direction from a technical artist. Even if the developers were not full time on the Fur project, they would sit in on meetings, providing input, intuition, and experience to the process. Once the system was semi-usable, our look development groomer came and sat with the team as well, to provide direct feedback on usability and desirable features.

The result of such close collaboration was a multithreaded, render-time Fur System we could put into production just 90 days after initial design/development began. Of course, development and enhancements continued on from there, and redesigns occurred as complexity needs increased, but the core system is still in place today, and happily processing, displaying, and rendering hairs on the orders of millions for multiple feature quality creatures.

5.2 Out in the Open

We decided to open-source our core libraries to encourage collaboration within the visual effects community. From feedback we have received, it would seem that very large companies have a hard time taking on external code, and often prefer internal codebases which may be overlapping in scope. On the other hand, very small companies often lack the technical resources required to engage in complex software design, or prefer to direct their resources toward shot production, relying on off-the-shelf software and artist ingenuity.

Several small to midsized companies have been able to contribute back to our open source efforts, and our mutual pipelines have been able to flourish as a result. Dr. D Studios contributed the foundations of our Houdini host layer, which we have taken on since, and expanded dramatically. Blue Bolt have contributed towards interactive rendering and look development, including the beginnings of an Alembic translation layer and an Arnold render layer. Electric Theatre Collective contributed a first pass at a Mantra render implementation.

We are aware that Cortex is quite monolithic in nature, prohibiting ease of entry into the codebase, and this may be a barrier for adoption in some cases. We plan to address this by partitioning the modules into logical groups, and deprecating legacy functionality. Learning from these suggestions, we organized Gaffer's codebase in a more modular nature from the outset.

Recently, we have begun using GitHub's public issue tracking for Gaffer and have found it to be very effective. We have documented design decisions from both artists and programmers, providing as much transparency as possible. It provides public tracking both of internal and external development branches, and also gives a nice interface for official code reviews before anything is merged into the master branch. Encouraged by this positive development experience, we have moved Cortex to GitHub as well. The main branch of both projects can be found under the ImageEngine organization.

6 Discussion

Our pipeline for detailed digital creatures has evolved over the course of several productions and will continue to evolve in the future. The most pressing issue we face to date has to do with scaling. Individual characters are becoming so complex that they require deferred evaluation themselves. This complexity is compounded by herding such characters into crowds, and embedding them in massive digital environments.

To address these issues, we are currently developing and testing a new hierarchical approach to character and environment design and storage, called a SceneInterface. We again have chosen to work in an abstracted layer, above any one file format or software application. The interface and suite of tools define workflows for hierarchical scene traversal, and any format could be plugged into them by defining an appropriate translation layer.

We have developed the translation layer for our own thread-safe SceneCache format, as well as live Maya and Houdini scenes, and we intend to modify our Alembic layer to follow this scheme as well. The resulting SceneInterfaces can be loaded in both Maya and Houdini, using deferred exploration of the hierarchy and procedural rendering of the objects, without the overhead of creating native geometry data. At any time, individual leaves or branches of the hierarchy can be converted to native geometry and manipulated by artists and TDs as necessary. The SceneInterface and associated tools are included in the Cortex libraries, and form the basis for Gaffer's native scene exploration.

Another issue we experienced with ModularProcedurals on our current productions has to do with usability. For our more complicated assets, the shader trees became massively repetitive, and difficult to introspect. There was a disconnect between how the user thought of the procedural and how it was actually represented in the renderer. While we had developed simple mechanisms for sharing shaders between assets, we were lacking a good representation of global shared information. We also had issues with performance, largely due to the fact that initially, development speed was prioritized and much of the implementation of modifiers was done in python, which is essentially single threaded. To address these issues, we have begun development of a shader authoring workflow using Gaffer. Nodes in Gaffer are almost exclusively c++, the graph is designed for multithreaded evaluation, shaders can be designed as openly as possible, and the render scene is inherently exposed, flowing through the graph itself.

6.1 Future Work

We are pushing forward on Gaffer development, currently focusing on shader authoring, interactive progressive rendering, practical lighting setups, basic compositing, and render pass management. We plan to utilize some more recent open source projects, such as OpenColorIO and OpenImageIO, deprecating the associated aspects of Cortex as appropriate. We are also very interested in investigating environment and crowd authoring and management, a native paint system, and any other ideas people find interesting. We are always open to discussion and contribution from the community, and invite anyone interested to get involved.

References

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept.), 509–517.

BREDOW, R. 2000. Fur in stuart little. In *Course 40 Notes*, ACM, SIGGRAPH 2000.

COFFMAN, JR., E. G., AND DENNING, P. J. 1973. *Operating Systems Theory*. Prentice Hall Professional Technical Reference.

CORTEX, 2007. Libraries for visual effects software development. https://github.com/ImageEngine/cortex.

COURNIA, N., SMITH, B., SPITZAK, B., VANOVER, C., RIJPKEMA, H., TOMLINSON, J., AND LITKE, N. 2012. Crom: massively parallel, CPU/GPU hybrid computation platform for visual effects. In *ACM SIGGRAPH 2012 Talks*, ACM, New York, NY, USA, SIGGRAPH '12, 43:1–43:1.

COWLAND, T., 2011. Procedrual hair system with GPU Marschner shading. http://code.google.com/p/cortex-vfx/wiki/ExamplesProceduralsHairShader.

CYCHOSZ, J. M. 1994. Graphics gems iv. Academic Press Professional, Inc., San Diego, CA, USA, ch. Efficient binary image thinning using neighborhood maps, 465–473.

GAFFER, 2011. Application framework for visual effects production. https://github.com/ImageEngine/gaffer.

KOPF, J., COHEN-OR, D., DEUSSEN, O., AND LISCHINSKI, D. 2006. Recursive Wang tiles for real-time blue noise. *ACM Trans. Graph. 25*, 3 (July), 509–518.

PIXAR. 2005. *The RenderMan Interface, Version 3.2.1*. Pixar.

REINDERS, J. 2007. *Intel threading building blocks*, first ed. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

SMHASHER, 2010. The MurmurHash family of hash functions. http://code.google.com/p/smhasher.

WATT, M., CUTLER, L. D., POWELL, A., DUNCAN, B., HUTCHINSON, M., AND OCHS, K. 2012. LibEE: a multithreaded dependency graph for character animation. In *Proceedings of the Digital Production Symposium*, ACM, New York, NY, USA, DigiPro '12, 59–66.