5-2014

Efficient Control of Assets in a Modern Production Pipeline

Timothy Curtis
Clemson University, tim2curtis@gmail.com

Efficient Control of Assets in a Modern Production Pipeline

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

---

by
Timothy Curtis
May 2014

Accepted by:
Dr. Jerry Tessendorf, Committee Chair
Dr. Donald House
Dr. Timothy Davis

Abstract

Managing large collections of assets created in today's CG productions is no easy task. This thesis examines basic production hierarchies and identifies problems that arise without proper workflow and asset control. The possibility of hundreds of assets being created in several workflows each week creates the potential for extraordinary time waste and user error without a system to track and manage the files being produced. It is becoming increasingly necessary to utilize methods during production that enforce naming and storage standards to prevent catastrophic data loss. This thesis presents an implementation of a customized database used to track all of the assets created and used within a production. This thesis also describes a package of tools and modules developed to handle asset creation during production, communicate with the database about asset information, and streamline the transfer of assets between workflows in a safe and consistent manner. The tools and database are constantly used by students in the Digital Production Arts program and have drastically reduced the amount of time spent on repetitive tasks when moving assets through the production hierarchy.

Acknowledgments

I would like to express my gratitude to my advisor, Dr. Jerry Tessendorf, for constant guidance, feedback and support throughout the course of the project. His efforts and contributions made this thesis possible.

I would also like to offer thanks to Dr. Donald House for urging me to apply to the Digital Production Arts program, and supporting me during coursework. I also offer my thanks to Dr. Timothy Davis for mentoring me during student productions and educating me outside of them.

Several Digital Production Arts alumni and students have also offered their support during this process, so I would like to thank Josh Tomlinson and Wil Whaley for sharing their pipeline expertise, as well as all the students involved in the Digital Production Arts program. Their collective feedback and support drove the creative process for the project.

Lastly, I would like to thank my mother, Pamela Curtis, and father, John Curtis, for their constant support and words of encouragement throughout my time at Clemson.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction and Concepts

A computer graphics (CG) production is generally the combination of many artists' work to create one final entity, regardless of whether it is a video game, film, TV episode or commercial. The role of each artist throughout the production is pivotal in meeting deadlines and ensuring the success and exceptional visual quality of the product. The work in a CG production is separated into several different departments, and responsibilities are divided among artists. Typically an artist will only contribute to one, maybe two, distinct areas in a production. This creates the opportunity for groups of artists to focus on different components of the production and create highly detailed and functional assets throughout the production process.

In order to successfully complete a CG production, artists must collaborate and build upon each other's work. The assets artists create in their respective departments will all contribute to the final product. Considering the detail and magnitude of today's CG productions, a large amount of information must be exchanged between artists. The protocols and processes for transferring items between artists defines the production pipeline. The structure of the pipeline plays a large role in determining the quality of the resulting film, as inefficient methods used to manage the movement of assets through the production hierarchy can result in data loss, increased effort to properly locate assets, and less iterations during the production.

This thesis will present an organized and efficient system to manage assets. First, core concepts behind the implementation will be discussed. Next, the pre-existing framework for the system, as well as common pipeline concerns, will be addressed. Similar implementations at other studios and companies will be explored. Lastly, design decisions and the implementation of the asset

management system are presented. In that chapter, each tool discussed is a new addition to the existing pipeline framework to establish safe and consistent ways to handle asset operations. Results of introducing the asset management system into the production pipeline are shown after.

1.1 Common Terms

For reference, the terms used frequently in this thesis are introduced:

Workflow This term describes the stage of production a particular item is currently in. Artists work within one workflow on a production to produce components that other workflows will rely on. For example, work produced in the modeling, surfacing, and rigging workflows will be necessary to create a functionally and visually acceptable character. A full set of production workflows is presented in Section 1.2.

Asset This term broadly describes an entity that is created during the production process. Assets may be Maya files, rendered images, textures, and other types of files that are created by any software used during production. Furthermore, assets will often act as components used to create larger and more complex assets later in the production process.

Publish This term refers to the action of declaring an asset as ready and available for subscription. A published asset is considered stable and non-volatile, and should be safe to use within any suitable workflow in the production. This concept is described further in Chapter 4.

Subscribe This term refers to the action of using a published asset. Typically, this action follows publishing operations to make use of the new assets. Furthermore, when an artist creates a subscription, the subscribed asset is referenced and used in the subscribing workflow until an artist changes, updates, or removes that subscription. This concept is described further in Chapter 4.

1.2 The Production Pipeline

The production pipeline can be loosely described in two ways. First, more concretely, the production pipeline is the set of processes used to transform a concept into a final product. Figure 1.1 presents a simplified view of the entire production workflow.

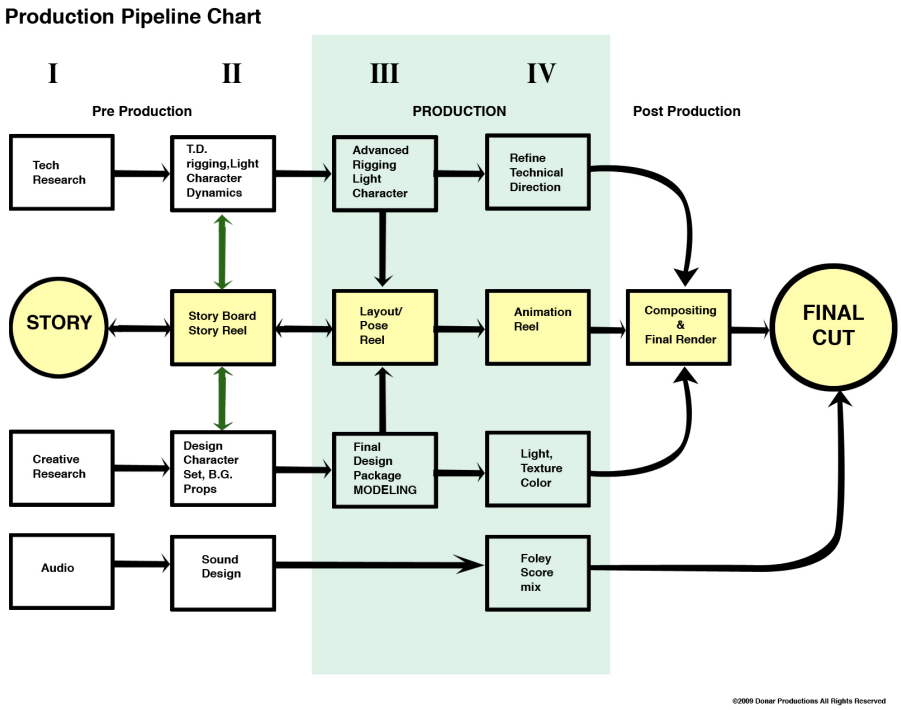


Figure 1.1: A view of the production pipeline and typical workflow [6].

The beginnings of the pipeline lie in the script and story development phases, where the details of the production are proposed and decided upon. This is the bulk of the first major production stage generally referred to as Pre-Production. Visual development for environments, characters, and props begins in this stage, as well as initial research for both character and visual effects if those areas are expected to be vital to the story.

The second major stage is Production, in which modeling, surfacing, rigging, layout, animation, lighting, effects (FX), and compositing work occurs. For reference, these workflows are briefly explained:

Modeling 3D meshes are created to reflect designs from Visual Development. Character meshes describe physical proportions, anatomy and structure, and environment meshes describe the terrain and areas characters will interact with.

Surfacing Character and environment surfaces are identified, and distinct materials and textures are created to adorn the respective meshes. Surfacing depends on modeling assets, and effectively finalizes the form of these assets by suggesting material and surface properties that

weren't realized in the modeling workflow.

Rigging Movement and control systems are created for any meshes and objects that will require movement in the production. Character skeletons are crafted to adhere to the intended anatomy of the character, and various skinning methods are applied to assure proper mesh deformation. Rigging depends on modeling initially, and then surfacing to assure that mesh deformation does not create problems with the assigned materials and textures.

Layout The story is realized in a 3D world. The environments and characters are roughly placed in accordance with the story. Camera properties, movements and cuts are decided to create a unique and engaging view into the story and animation.

Animation Character meshes are brought to life through key-framing control systems provided by rigging, and occasionally through motion capture of actors or artists for very specific or complex movements. Personalities of characters are fully realized through movement, mannerisms, and timing.

FX Can be separated into two areas: Character FX and Visual FX. Character FX refers to any supplementary actions needed to finalize character animation, such as cloth deformation for clothes, hair systems, or muscle, skin and flesh simulation for fine detail. Visual FX generally refers to environmental actions like explosions, fire, water and cloud simulations, fracturing of geometry, electricity, particle simulations, and more.

Lighting The 3D world is filled with light to interact with work created in all of the above workflows. Lighting can focus or remove attention from anywhere in the scene, and can interact with character and environmental surfaces in visually appealing ways.

Compositing The final production workflow that combines assets produced in the lighting and FX workflows into flattened image sequences. Elements of each rendered sequence are properly merged together to create the final images of the film. Image finaling and color corrective operations occur at this stage to properly integrate elements into the overall image sequence.

Dependency relationships are formed through the movement of assets through the production workflows. Assets are generally transferred downstream, that is, to workflows that depend upon assets generated from other workflows before work may begin. But CG productions are created iteratively, often with tens or hundreds of versions being created for each asset. In any given shot

or scene in a production, artists can make changes to assets created in earlier workflows with little effect on downstream departments. In this way, elements of production can move up and down the hierarchy based on the needs of the shot, and multiple workflows can produce multiple items for a shot simultaneously.

The last major stage of production is referred to as Post-Production. Any extra image finaling processes occur at this stage. Foley, dialogue, and sound FX are matched to the animation delivered from Production, and the final cut of the film is rendered with all of the latest assets. Figure 1.2 presents a view of the pipeline that is more complex and refers to many different workflow departments that exist today. Generally, many of these departments exist in studios that integrate CG material into live action footage. Full feature animations do not typically include all of these departments, but the figure shows the complexity that modern production pipelines can expect.

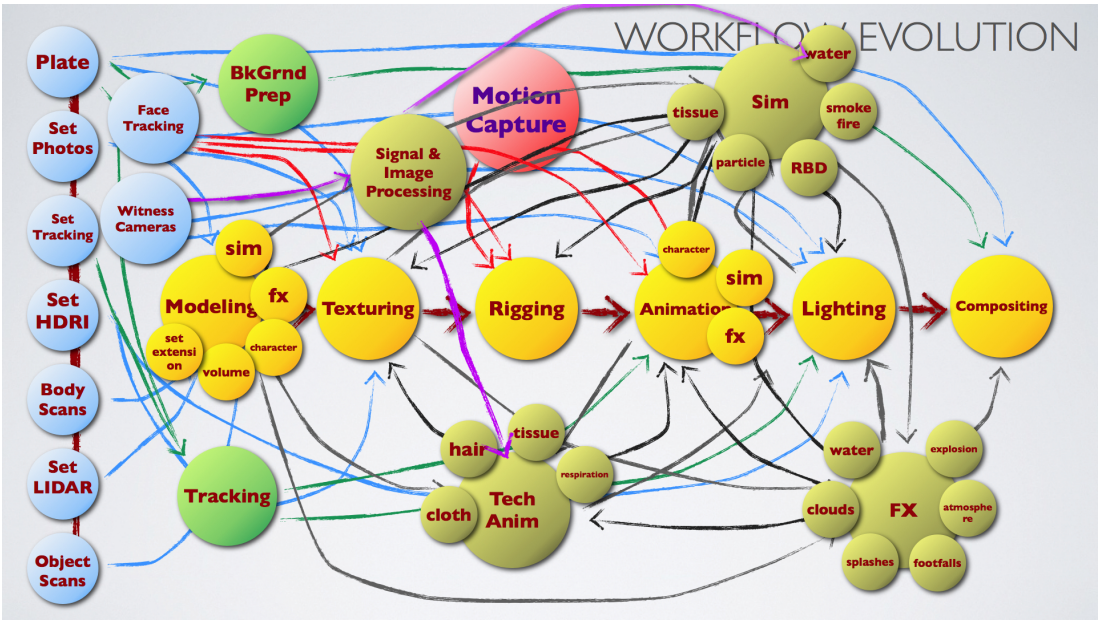


Figure 1.2: A view of the production pipeline integrating a multitude of workflow departments that can exist in modern studio environments. Courtesy of J. Tessendorf [15].

The other aspect of the pipeline is the explicit framework established to enforce organization and structure within the production. "Abstractly, a pipeline is both an established set of decisions and a collection of tools developed before production and managed throughout so that the focus for the artist is on the creative and technical challenges not the nuts and bolts of production" [14]. The set of decisions and collection of tools are typically used to describe naming conventions and storage locations for files generated during production so artists may simply produce them and let the underlying pipeline framework figure out the rest. Typically the pipeline file structure mirrors the workflow dependency relationships examined at the working level to form a logical workflow hierarchy. The tools that are used to enforce pipeline rules can be a combination of command line scripts, custom interfaces, file templates and versioning control schemes to ease the burden of file management for the artist. "Efficiency is gained by designing a set of procedures that best utilize the talents and specializations of the people and resources available without hindering them under the weight of an overbearing system" [11].

In discussing the pipeline further, this thesis will only focus on productions that are entirely CG, and will not discuss productions in which CG elements are combined with live-action film.

1.3 Databases

Databases can be used in the production pipeline to assist with various storage and management problems. A database is a collection of data stored with a governing organizational scheme. Casually, a database might refer to a set of spreadsheets, log files, or even papers filed in a cabinet. For the purposes of this thesis, database will refer to the collection of data stored and managed on a computer or set of computers.

Databases are created to store large quantities of data, and retrieve pieces of data with specialized requests. "Databases are set up so that one set of software programs provides all users with access to all of the data" [2]. Furthermore, the logical structure of the database is decided by a database model. The database model describes the format for the database's information, as well as how the data can be accessed and updated by users. Various models exist that describe different storage and access techniques for databases, but this thesis will focus solely on the Relational Model.

The Relational Model describes a structure in which data is stored in tuples, where a tuple is an ordered set of attributes. An attribute represents a single piece of information of the tuple

object. This structure can be realized in a table, with the rows being the tuples, and the columns being the attributes [5]. Figure 1.3 provides a table representation of relations.

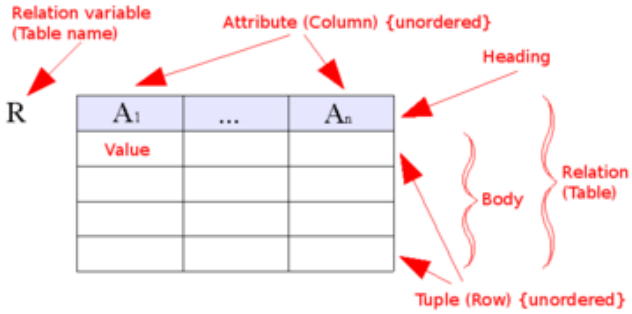


Figure 1.3: Table representation of a relation [3].

A relation is a set of unordered, unique tuples that share the same attributes, otherwise known as a table. Therefore, duplicate rows in a relation may not exist unless a customized database model explicitly allows it. Furthermore, "one or more attributes has to be chosen as the primary key, which must be distinct among all tuples in the relation" [10]. The significance of the Relational Model lies in the power to declare exactly what needs to be stored or queried. With table headers being a collection of the attributes, software or users can access attributes in any number of rows, which can be controlled through various sorting or filtering requests. Common operations on rows and attributes are insert, update, remove and query, which are generally part of a structured query language (SQL).

Relationships may also exist between tables, and this concept is expressed through the use of foreign keys. An attribute in one tuple may be used as the attribute of another tuple, and often this relationship occurs between two tables as opposed to within one. For example, consider two tables, ***Store*** and ***Product*** that describe company store locations, and the items that each store sells. A particular company may sell fifty unique products, but each of the rows in the Product table would have an identifier to let the user know which Store that item belongs in. This relationship is also known as a foreign key constraint, emphasizing that that one relation is dependent upon the attribute of another relation. Database models can utilize this concept to construct complex database structures and table hierarchies, and establish a scheme of control and dependency between the tables.

1.4 Asset Management

Pipeline and database attributes are key components in designing a system specifically to handle the creation and usage of assets in production. Managing assets in the physical or digital form has been an arduous process since mass manufacturing of products was conceived. In CG productions, every asset is unique, and storing, tracking, and re-using these assets has often been a time-consuming and repetitive process. The term Asset Management is broadly defined as a system to control the production, location, metadata, and retrieval of assets, and these systems may vary widely from company to company. Asset management systems are the backbone of a production pipeline, as assets are the items that are transported between workflows to build the final product.

Artists in downstream workflows will depend on assets created upstream, and the ability to quickly and efficiently locate the assets they need is critical. Co-Founder and CTO of Southpaw Technology, Remko Noteboom, offers his advice about the searchability of assets:

> Fundamental to any workflow system is the ability to search for assets. However, no Google-style, tag-based indexing algorithm will ever be sophisticated enough to construct an engine, a building or any complex digital asset. In order to achieve this, a different organizational structure for assets must be used, one in which everything has its place and is organized around a strict set a rules based upon the design of the workflow. This requires the assets to be organized not according to tags and indexing, but according to its "place" [11].

Asset management systems can use a database backend to properly store assets with an array of metadata and important information. The type of database impacts the performance of the production. Allowing the backend to handle data directly separates the asset management tools from the data. Consolidating the data management in this way reduces the overall complexity of the system because supporting tools need only to know how to request data from one source, not how to locate the data manually.

In most cases assets depend on other assets for correct functionality. Tracking these dependencies is just as critical as tracking the assets themselves. Dependencies are typically uni-directional, and more specifically, dependent upon assets in upstream workflows. In other cases, dependencies might be bi-directional, where it is important to define what an asset is using, as well as which assets are dependent on it. In any case, ensuring that this information is available to the users and automated tasks is important, in order to control the integrity of data transfer and relationships between workflows.

Chapter 2

Background

Since animated productions became popular nearly a century ago with the rise of Walt Disney Animation, handling assets during production has exponentially increased in complexity. "Prior to the use of computers, cel animation work was linear in nature. A traditional animator could change animation repeatedly during the animation stage of production" [4]. Iterations in such a production were expensive as all the assets had to be recreated when redrawing a sequence. While this process restricted changes later in production, the workflow was very direct. Complexity in the pipeline was not generally an issue because artists only needed to keep track of the assets currently being created as if they were part of a one-directional assembly line.

The major drawback to a linear workflow is the absolute dependence on previous assets. If a character design changed, months of drawn animation needed to be redone. Design changes once animation begins is normally a poor decision, but artists in today's production pipelines can better afford to make those decisions due to the non-linear workflow of a CG production. Multiple workflows for a single shot or item can begin simultaneously, which means lighting for an environment can start while rigs are still being finalized for animation. Furthermore, altering asset dependency when production reaches the animation and lighting stages is not completely destructive. This means that if the model changes for a particular character, animation for that character can remain the same if the rigging workflow produces new assets that contain re-skinned characters and can follow the same animation curves.

Creating an efficient and flexible pipeline that can adapt to the needs of any production environment is nearly impossible. The needs of every studio in terms of hardware and software are

too diverse. There is a common set of issues that each studio must confront to ensure their files are handled safely and that artists have the ability to transfer assets to other workflows. Developing a logical file structure, version control schemes, and an asset management system are all key to keeping working files safe and ensuring the success of the production.

2.1 Clemson's Production Pipeline

Clemson's Digital Production Arts (DPA) program made decisions about structure, naming conventions and tools that supported a rigid and logical file system structure, and the Asset Management system presented by this thesis focuses on the toolset developed for this pipeline specifically.

2.1.1 Production Hierarchy Overview

The DPA program follows a very strict production hierarchy. Figure 2.1 presents a visual representation of all the elements of the hierarchy, and the elements are explained below:

Show Top level of the hierarchy. Everything produced will be under this name.

Stage Defines the major working stages of the show. The six major stages are:

lookdev - Visual development for characters and environments.

board - Story development and layout.

prod - Elements of production, where most work is done.

sched - Schedule for the show deadlines and supporting scripts.

ref - Reference material for elements in the show.

review - Weekly and/or daily reviewed material for critique.

Shot A piece of the story that takes place from a distinct camera viewpoint, in which camera cuts usually decide a shot change. Alternatively, custom shots can be created that hold files not related to the story, but instead files that may be used within many other shots. Two custom shots are created by default for each show:

share - Holds any element in the show that will be used in two or more shots.

master - Holds official versions of published assets.

Workflow Defines working stages of a shot or shared asset. The major workflows were presented in Chapter 1.

At this point, the hierarchy diverges into two separate paths for assets and workflow items. Workflow items generally consist of iterative working files that have not been published, and are not treated as assets that will move to other workflows. These items are still version controlled to preserve previous work if a roll-back is needed, but no system is responsible for tracking these files. On the other hand, the assets are a collection of the published items and files that will be handed to other workflows. Assets are also version controlled, and there are typically tracking systems in place to locate these files for other workflows. Furthermore, assets have extra information associated with them in the file structure hierarchy to distinguish various types or resolutions of assets that may have been created under the same asset name.

Workflow Item Path:

Workflow Item A short name that describes the working element.

Tool Any tools installed in the DPA studio that can be used produce assets. Common examples are *maya, houdini, nuke, mari.*

Version Versions of the working element.

Asset Path:

Asset Name of the asset, generally similar to the workflow item from which it was produced.

Version Versions of the asset.

Resolution Resolutions differ between modeling, surfacing, and rendering assets. Models are typically distinguished as high or low resolution, surfacing texture files as 1K, 2K, 4K, 8K or higher, and rendered images as 1920x1080, or another custom image width and height.

Type Common examples are *exr, tiff, mov, ma* (for Maya ASCII), *hipnc* (for Houdini).

Every file that is created in the production will fall into one of the leaf elements of the production hierarchy, version or type, with the exception of some configuration files that exist at the show and shot levels to describe specific user and production variables.

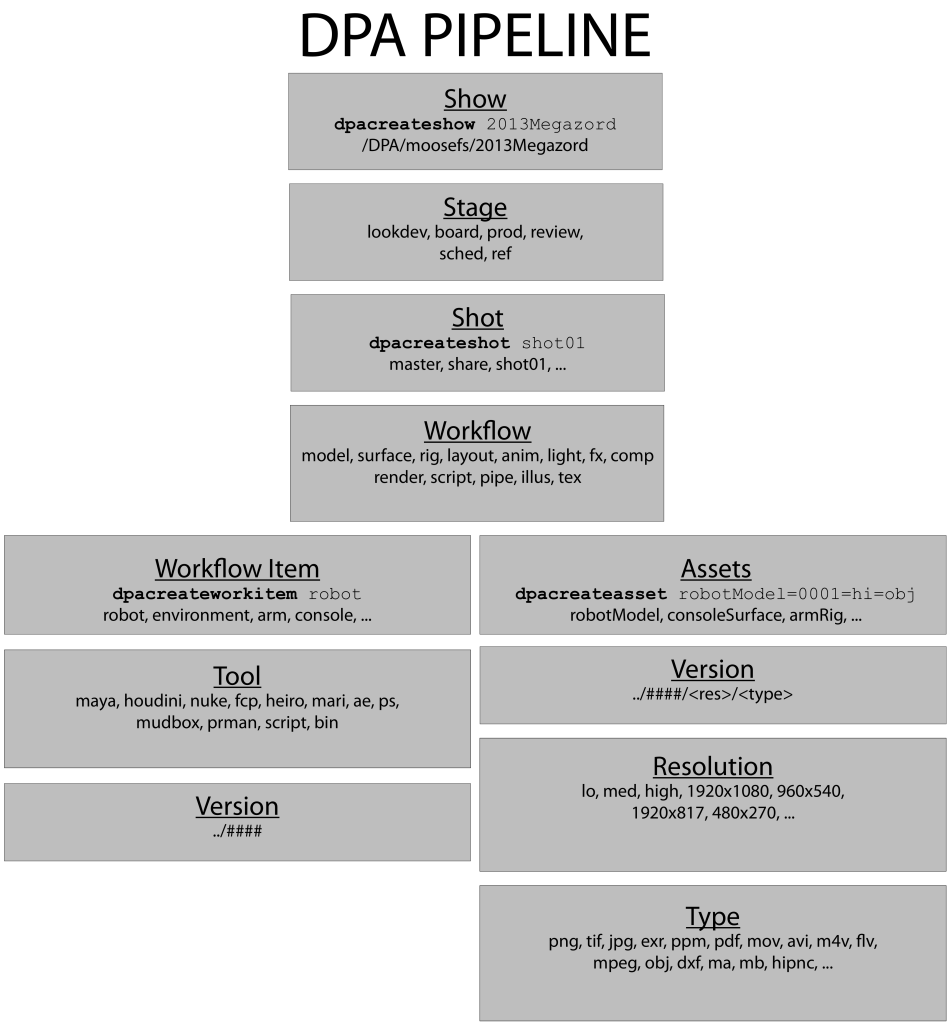


Figure 2.1: Overview of the production hierarchy in DPA, with sample script commands [1].

2.1.2 Pipeline Foundations

In addition to determining the file structure for any production, the DPA program developed a collection of tools to support and enforce the file structure. The pipeline at Clemson was created around the idea of tracking the user's location in the production hierarchy whenever the user started to work on a file. Both naming and location conventions could be enforced if pipeline tools could determine where a user was in the show. Therefore, the action of *setting* to a location was decided as the best option. This action would alter environment variables that are visible to the pipeline

toolset and create a context that would inherently drive naming and location conventions.

The driving script behind the pipeline was titled **dpaset**. This script would alter custom environment variables in the user's profile to establish context within a show. A large python library of utility scripts were created to support this basic functionality, and provide more control over other aspects of the pipeline. File structure conventions, as well as naming conventions, could easily be enforced if tools automatically refer to the environment variables and infer naming principles based on shot, workflow, etc. Furthermore, the version control scheme for *workflow items* is enforced efficiently since the user's current working version is tracked through environment variables. Version control for assets is not explicitly defined through the same environment variables, but tools detect the latest version and increment to avoid overwriting previous assets. Additional tools and interfaces extend the use of the contextual identification to repetitive tasks.

Some important tools supporting the file structure are *dpacreateshow, dpacreateshot, dpacreateworkitem,* and *dpacreateassset*. The first two tools are responsible for creating a templated directory structure for a show and shot respectively. The latter two tools create the items and assets within workflows, the file structure for workflow items and assets, and install templated data if appropriate. These tools enforce the file system context expected by dpaset. Other versioning tools handle workflow items after the initial creation.

2.2 Common Problems and Concerns

Just as a pipeline can increase the productivity of a team of artists, it may also act as a barrier. Unsolved issues and new obstacles almost always weave their way into the pipeline, and if they aren't handled correctly, new hindrances can halt the flow of assets until corrected. A confusing file structure or naming convention can introduce problems for artists attempting to locate certain items or assets. Negligence of pipeline or file structure scalability can cause locating assets to be extremely difficult for some productions attempting to create custom resolutions or files. Alternatively, the pipeline framework should be comprised of modules that are as flexible as possible, so bugs or errors can be handled gracefully, and any errors that leak through do not result in data loss.

2.2.1 Pipeline Structure

Some problems may arise in the artist's discontent with a rigid file structure. The workflow hierarchy is absolute in the DPA program, and disallows the creation of custom production stages and workflows unless those changes are introduced throughout the entire studio.

Very rarely should flexibility be a problem for an artist during production. Problems such as custom image resolutions or unknown file formats can very easily confuse the pipeline toolset and create files in unknown locations, or fail during asset creation entirely. Oftentimes overrides can be defined at a show-specific level that can support the creation of some lower-level, customized items. Ultimately, the producers for the show and pipeline developers decide on the file system structure to use, and enforcing that structure is pivotal in maintaining control of working files and assets.

Alternatively, the pipeline framework supporting the file structure of the show might cause uneasiness with some artists. While there isn't a foolproof way to ensure artists' complete acceptance of pipeline tools, developing the tools in a modular way that allows them to easily be extended or updated, as well as transparent to the user, creates a flexible working environment where changes to the pipeline framework are rarely noticed by the artists.

2.2.2 Asset Management

If the concept of asset management is neglected in a modern production pipeline, artists can see an exponential increase in difficulty of locating assets they depend on for work, and will often have to spend a burdensome amount of time on tasks that produce, use, and locate those assets. An asset management system was a critical component of the pipeline that the DPA program was missing, causing the publishing and referencing operations in Autodesk Maya to be very time consuming.

Two of the most important aspects of any asset management system is the ability to both automate certain tasks and avoid burdening the artists with extra information. "Any operation that requires an artist to carry out the same tasks over and over again is a prime candidate for automation via the asset management system" [7]. Version control must inherently be implemented with asset management, as hundreds of versions of assets are created in a production and maintaining control of those versions helps to prevent data loss. Common automated operations include publishing a file to deliver to other workflows, exporting mesh from modeling files, or submitting jobs to a rendering farm.

These operations should not seem like obstacles for the user. "Many artists perceive asset management as introducing obstacles into their workflow, or forcing them to work in certain ways" [7], so reducing or completely hiding the overhead of the operations is beneficial. This task can be gracefully handled by creating custom interfaces for the artists that behave just as normal interfaces within their software, and present the artist with expected fields and information. Some of the information can be automatically populated based on workflow and asset context, and any extraneous work can be executed in the background to reduce the presence of unnecessary information or processes.

Chapter 3

Related Work

Many studios and third-party software vendors have created their own implementations of a production pipeline and methods to control and distribute digital content. The scope of these software packages varies, depending upon the popularity and community backing of the products. The two software packages that will be addressed in this thesis are *OpenPipeline*, which is an open-source, MEL-based approach to workflow management, and *Shotgun*, which is a software package developed commercially that has elaborate user interfaces and methods for production tracking, version control, and file management. Additionally, the production pipeline at *Rhythm & Hues* is relevant, as it has several similarities to the pipeline at Clemson.

3.1 OpenPipeline

OpenPipeline is an open-source, community-driven framework that was initially conceived at Eyebeam Art and Technology Center, and later primarily developed at the Pratt Institute [12]. The project started around integrating a pipeline framework into a production centered around Autodesk Maya. Most of the logic and structure is formed around MEL commands that are executed within Maya, and is free of any database and external tool interaction. Simple commands such as saving and opening are replaced with OpenPipeline operations that will handle their items and assets in a consistent manner, and ensure those items are properly version controlled.

Robert O'Neill, the creator of OpenPipeline, offers a concise explanation of OpenPipeline features:

> OpenPipeline offers asset management, revision control, user notes, and scene population and management capability. The asset management feature allows you to name, store, retrieve, and associate information with data. If, for example, you create an asset called bob of type character, it will automatically be saved to /lib/character/bob. This file structure easily archives and delivers the asset, removing the guesswork of where it should be saved in the directory structure. Revision control keeps track of the edits to a file and incrementally saves the file, instead of overwriting it with each save [12].

Within OpenPipeline, artists have the opportunity to work on files independently of the pipeline in "workshops", and choose the appropriate time for assets to be mastered, or published, and made available to other workflows in the production [13]. This process offers a layer of protection to downstream workflows so assets are not constantly produced that may be incomplete or volatile. With an implicit form of asset control and a strict versioning system for workshop files, OpenPipeline offers artists a simple and unobstructed pipeline framework for studios primarily using Maya, and relies only on its self-controlled file structure conventions.

3.2 Shotgun Software

Shotgun offers a much more extensive approach to managing the production workflow. Shotgun, at its core, is a web-based production tracking system that presents users with a detailed view of ongoing tasks in a current production. Producers and supervisors are given reports and broad views into the production activity, while artists manage personalized task lists and messages to direct their daily routine. Shotgun's production management introduces a new, powerful method of keeping productions on track by managing workflow information and streamlining communication between all levels of production.

While the production tracking side of Shotgun handles much of the scheduling and task management, Shotgun is also supported by an easily accessible Python API. Developers at any studio can customize the way Shotgun integrates into their production process by manipulating this API to adapt to their software. Jim Geduldick explains his view about Shotgun's Open API:

> Because of Shotgun's open technology, you can really tailor it to your studio's needs. Open Python API, event triggers, asset management integration and being able to extend the UI with tab widgets are just some of the tools at hand to get your pipeline where it needs to be. One thing that comes to mind with the above listed feature set is automation. Shotgun allows you to automate very tedious tasks with some of its scripting capabilities [8].

Shotgun benefits users with its new views into production data and its flexibility for customization and addons, and also supports this data with an asset management system called Shotgun Pipeline Toolkit (SGTK). SGTK, formerly known as Tank, "is fully integrated with the Shotgun production tracking platform, providing tools to keep the file system organized and an app framework that provides a wide range of independent apps for specific functions such as file publishing, versioning, and scene building" [16]. The app framework that SGTK uses was built with flexibility in mind, as their framework is highly customizable and new apps for asset management operations can be created quickly for in-house software. With this type of dynamic file management, Shotgun has consolidated many aspects of the production pipeline into one flexible and robust system.

3.3 Rhythm & Hues

The production pipeline at Rhythm & Hues (R&H) operates similarly to the pipeline at Clemson. Artists complete most of their work within *work areas*, iterating through controlled versions of their working files until they are ready to publish an asset. During the publish, metadata is added to the assets to track the dependency information and work area that created it. Other operations extend the functionality of the asset dependency tracking, such as dynamic scene building, where groups of assets can be resolved and referenced into a work area.

The actual pipeline framework is abstracted in a way so specific shows can implement their own asset hierarchy based on the needs of the production. The flexibility of the system is well-explained below:

> High level tools and interfaces are layered on top of the framework and operate on the abstract components rather than the concrete types. The result is a system where users at every stage of the pipeline use the same toolset and speak the same production language. As new production requirements are added, support teams add new concrete component implementations or plugins that work within the pipeline immediately making the pipeline extremely flexible from top to bottom [9].

Behind the framework lies a database capable of tracking and delivering all the assets produced at R&H. Because R&H consists of several locations, assets need to be shared amongst locations, and the database employs several methods to manage the relationships between studios, as well as synchronize assets between locations to ensure data consistency.

Chapter 4

Design and Implementation

Asset Management is a concept that was not addressed completely in the pre-existing pipeline in the DPA program. This chapter will cover all of the decisions and tools that construct the new asset management system. Each tool discussed is a new addition to the production pipeline, and was developed to properly and concretely address pipeline features that would be responsible for handling assets and accompanying operations in a deliberate and logical manner.

To integrate the asset management system in the DPA pipeline, a combination of backend data management and frontend control modules was developed to separate the functionality of storing and using assets in production. All tools were created to interact naturally with the existing pipeline framework described briefly in Section 2.1.2. These tools were designed primarily to create a better working environment for the artist by automating repetitive, technical tasks and allowing artists to spend more time in the creative process.

4.1 Design Decisions

An asset management system can be designed in many ways depending on the type of storage and use the studio expects. Implementations can range from simple file storage and log files existing alongside assets on disk, to complex databases that track not only the assets, but dependencies, use, and post-production operations (i.e. archiving, deletion).

A MySQL database approach was chosen as the backend for the asset management system, supported by Python modules on the command line and PyQt graphical user interfaces (GUI) within

the studio applications. This design enhances the asset production process and allows the artists to make use of various tools to create, publish, locate, browse and subscribe to assets.

PyQt was selected as the interface library in an attempt to create a suite of pipeline interfaces that could communicate with different software packages. This method is superior to simply relying on application-specific APIs because it enables the use of common operations across workflows and platforms. This reduces the number of tools needed to complete publishing, subscribing, and other common operations by introducing one standardized and compatible set of tools in the pipeline.

4.1.1 Backend Data Management

A database was chosen precisely for the basic functionality of a database: to organize data. Assets in the recent-past pipeline at Clemson were only organized if artists consistently created assets in the same way, which presented the potential for instability and data loss as each production grew larger. The DPA program needed a system that was configurable, extendable, and has already been practiced by other studios.

MySQL was chosen for the database language and model because the notion of relations discussed discussed in Section 1.3 provided the type of file storage the DPA program needed. Artists create thousands of assets, and while each individual asset is unique, the necessary attributes to store assets properly are not. The table scheme within the relational database model can store millions of rows that share the same attributes, and assets fall into that category. Additionally, the use of foreign key constraints could enforce a hierarchy in the table schema. Users could discern the location of an asset by following the path of foreign key constraints until it reached the root table, which in this case is the show. Each asset would have a place in the pipeline, just as each asset has a file location on disk. Furthermore, separating the tables into a hierarchy of relations that have different, but specific, attributes reduces the amount of data redundancy in the backend and may lead to faster operations on the database. See Figure 4.1 for the database model visualization.

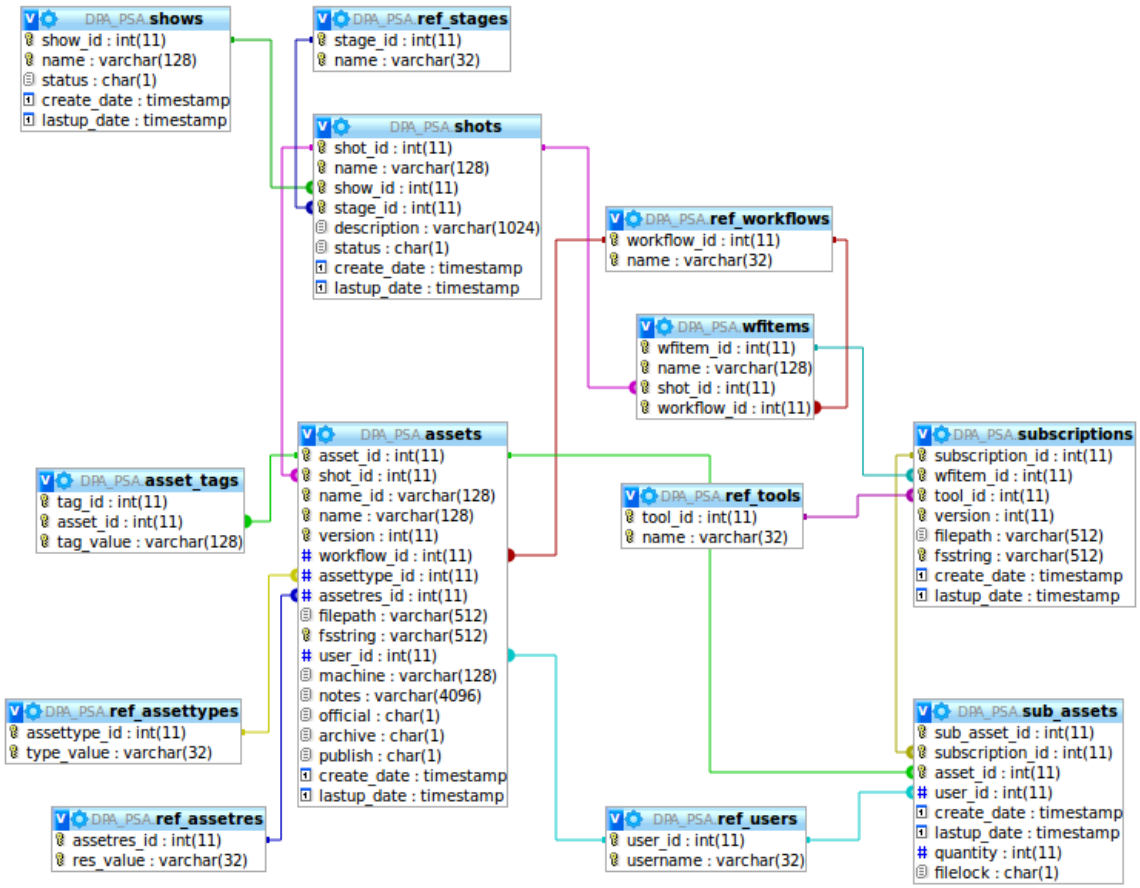


Figure 4.1: MySQL Database Model for the Asset Management System.

4.1.2 Publishing

The act of publishing is the primary way of moving assets through the production hierarchy. Many of the assets in the DPA program are created in Autodesk Maya, so connecting the pipeline to the Maya Python/MEL API was a logical choice.

Earlier versions of the Publishing tool were developed to address fundamental pipeline concerns, such as naming conventions and correct storage locations. Previously, the mastering process for a particular workflow item was a foreign concept, and this tool allowed users to explicitly declare assets as ready for referencing. Additionally, the earlier versions of the tool properly version controlled published assets, but presented artists with the latest version of the asset through symbolic link manipulation, which would update workflows depending on that asset automatically.

After the database was introduced, the pipeline was able to track and manage assets much more effectively, which removed the need to update assets automatically through the use of symbolic links. Artists needed the opportunity to explicitly publish assets, tag them with keywords and notes as metadata, and version control those assets without affecting the workflow items in dependent departments. Various notes and tags could be automatically inferred from the artists working context to reduce the amount of information explicitly required from the artist. Therefore, the publishing process became a deliberate user action with the capability to add as much information as desired, but never the necessity to do so.

4.1.3 Subscribing

Subscribing was a newer concept to establish the process of creating and tracking asset dependencies. Just as assets are produced from Autodesk Maya via publishing, they are also referenced into workflows by use of the Maya Referencing Tools. Using a new method to redefine this referencing process allowed greater control over tracking these asset dependencies.

Earlier versions of the Subscription tool were developed simply to buttress the existing Maya Referencing Tools. These versions were less concerned with the asset tracking, and instead focused on the core issues of resolving naming and location conventions correctly. After completion of the database model, it was clear that new tools needed to be developed to support new asset tracking methods to extend greater searching power to the artists and enforce dependency correctness.

A new subscription concept was developed to give the artist explicit control over the assets in their working files. Subscriptions are a combination of two parts: *Subscription Areas* and *Subscriptions*. Subscription Areas refer to the location of working file. This location allows the user to create distinct asset dependencies as they iterate on that working file, meaning that older versions will remain unaffected by future changes and retain stability. Subscriptions are actual asset dependencies that are created within a Subscription Area to include that asset's data in the current working file. For example, creating a model Subscription in a rigging Subscription Area means referencing the published model file into the rigging work file within Maya.

Database asset and dependency tracking techniques extended the subscription process to support new conflict identification and resolution methods if the working file's subscriptions were different from those last recorded in the database. In the case of file corruption or deletion, new files could be created in a location the database recorded subscriptions for, and references could

be automatically loaded into the file based on those records. Most importantly, the artist would have control over every decision in the new subscription process, and increased asset searching and referencing abilities would cut down on time spent referencing assets from upstream workflows.

4.1.4 Rendering

Rendering tools to export lighting assets from Maya had previously been developed in an effort to maximize the use of the renderfarm in the DPA program. However, rendering complex final images from Nuke had not been addressed. Artists would manually have to create asset directories, version them correctly, and assign those file paths from within Nuke. This process needed automation via the Nuke Python API, which allows users to declare custom nodes and panels for specific studio needs. Using this, a DPA-specific Write Node was developed to interact with the current pipeline, and a custom panel was created to split rendering jobs into four parts and distribute the rendering tasks to the renderfarm. This node drastically reduced the rendering times for Nuke jobs and allowed artists to iterate in the final stages of productions much faster.

4.1.5 Command Line Control

While GUIs would act as the primary means of connecting the asset management system with the artists, command line tools quickly fetch and display data from the backend. Publish and Subscribe operations are also available on the command line, allowing other pipeline tools to incorporate those operations to track generated assets. Furthermore, publishing was not the only action that resulted in tracking an asset. A generic asset registration tool was written early in the development process to track assets regardless of how they were generated, an action that was vital for tracking and archiving assets important to the show that were not published through the Maya and Nuke workflows.

4.2 Database Implementation

The database was constructed to closely resemble the production hierarchy in the DPA program. Foreign key constraints were created to form a tree that could be traversed in the same way a user could move deeper into a filesystem. Some tables were identified as reference tables, meaning that these tables would only hold static information. *Workflows, Tools, Asset Types* and

Asset Resolutions were tables of information that would remain the same throughout the course of a production, and these tables are seen in Figure 4.1 with a **ref_** prefix. Separating these tables from the absolute hierarchy created less complex query chains to reach low level information.

The two primary entities in the database model are the ***Assets*** table and ***Subscriptions*** table. The Asset table handles all information in regards to when and where an asset was created, the version of the asset, whether the asset was published or marked as the official versions, and various pipeline identifiers. Alternatively, the Subscriptions table, in combination with the ***Sub Assets*** table, store information about asset dependencies. Subscriptions are created within workflow items to track locations of asset dependencies, and the Sub Assets track which assets belong to a particular subscription, as well as who subscribed to an asset. Software GUIs as well as command line tools were developed to interact primarily with these three tables.

The entire script used to create the database tables can be found in the Appendix. For reference, consult Listing 4.1 for the fields and commands required to create the Assets table.

```
-- -----------------------------------------------------
-- Table DPA_PSA.assets
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.assets (
  asset_id              INT NOT NULL AUTO_INCREMENT,
  shot_id               INT NOT NULL,
  name_id               VARCHAR(128) NOT NULL,
  name                  VARCHAR(128) NOT NULL,
  version               INT NOT NULL,
  workflow_id           INT NULL,
  assettype_id          INT NULL,
  assetres_id           INT NULL,
  filepath              VARCHAR(512) NULL,
  fsstring              VARCHAR(512) NULL,
  user_id               INT NULL,
  machine               VARCHAR(128) NULL,
  notes                 VARCHAR(4096) NULL,
  official              CHAR(1) NULL,
  archive               CHAR(1) NULL,
  publish               CHAR(1) NULL,
  create_date           TIMESTAMP NULL DEFAULT NULL,
  lastup_date           TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (asset_id),
  UNIQUE INDEX (shot_id, name_id),
  UNIQUE INDEX (shot_id, name, version),
  UNIQUE INDEX (fsstring),
  CONSTRAINT asset_shot_id
    FOREIGN KEY ( shot_id )
```

```
    REFERENCES DPA_PSA.shots ( shot_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT asset_workflow_id
    FOREIGN KEY ( workflow_id )
    REFERENCES DPA_PSA.ref_workflows ( workflow_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT assettype_id
    FOREIGN KEY ( assettype_id )
    REFERENCES DPA_PSA.ref_assettypes ( assettype_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT assetres_id
    FOREIGN KEY ( assetres_id )
    REFERENCES DPA_PSA.ref_assetres ( assetres_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT asset_user_id
    FOREIGN KEY ( user_id )
    REFERENCES DPA_PSA.ref_users ( user_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;
```

Listing 4.1: Asset table creation command.

4.3 Python Toolset Implementation

In order to interact with the MySQL database, a large backend module was developed to consolidate the database connection and communication within one object. Front end tools did not manage a connection with the database, nor did they form SQL statements to send to the database. The backend module is capable of handling requests from all frontend modules by using flexible search functions, and return data in dictionaries that the frontend modules all know how to handle.

The backend module used the MySQLdb python library as the foundation for managing the connection to the database and executing queries. The DictCursor class in the MySQLdb library was used to return data from queries in a dictionary rather than a collection of tuples, providing extra flexibility when sending information to frontend tools and reducing error that could arise from attempting to infer information based on tuple values rather than key-value pairs. Listing 4.2 showcases the use of MySQLdb in the backend to retrieve certain assets based on a specific query. Table 4.1 holds the assets those commands would retrieve.

```
db = MySQLdb.connect(
    host=HOST,
    user=USER,
    passwd=PASS,
    db=BASE,
    cursorclass=MySQLdb.cursors.DictCursor
)
cursor = db.cursor()
cursor.execute("""SELECT name, version, user_id, create_date
                  FROM assets WHERE shot_id=4 AND version=8""")
assets = cursor.fetchall()
cursor.close()
return assets
```

Listing 4.2: Execution sequence using MySQLdb to retrieve assets.

name	version	user_id	create_date
modelCoralBranchPublish	8	7	2014-03-10 20:52:48
surfaceAnemoneCondyPublish	8	2	2014-03-12 14:09:23
modelGrass01Publish	8	2	2014-04-05 11:47:59
surfaceSandFloorPublish	8	2	2014-04-05 12:39:36
surfaceRockShaderPublish	8	2	2014-04-06 10:29:44

Table 4.1: Assets returned from MySQLdb query.

Furthermore, the backend was supported by two modules responsible for explicitly determining how assets and subscriptions are handled on disk. These two modules defined item-specific fields, the rules and naming conventions for creating fields, and the formatting guidelines necessary for proper metadata storage in the database as well as how to print to a terminal. When registering an asset in the database, the respective handler was always invoked to properly format the asset data and provide the backend with an additional layer of definitions to ensure data correctness for the artists.

4.3.1 Command Line Functionality

Other functional scripts were created to quickly add assets or subscriptions to the database based on filepath flags specified on the command line at execution time. Additionally, command line tools could be executed to create an interactive shell session with the user. The user can browse both assets and subscriptions of the show specified at start-up, and has the ability to add notes to

assets if they are aware of a problem with an asset not recognized by the artist. Figure 4.2 showcases the command line interactive module, with the valid selection of options printed for the user.

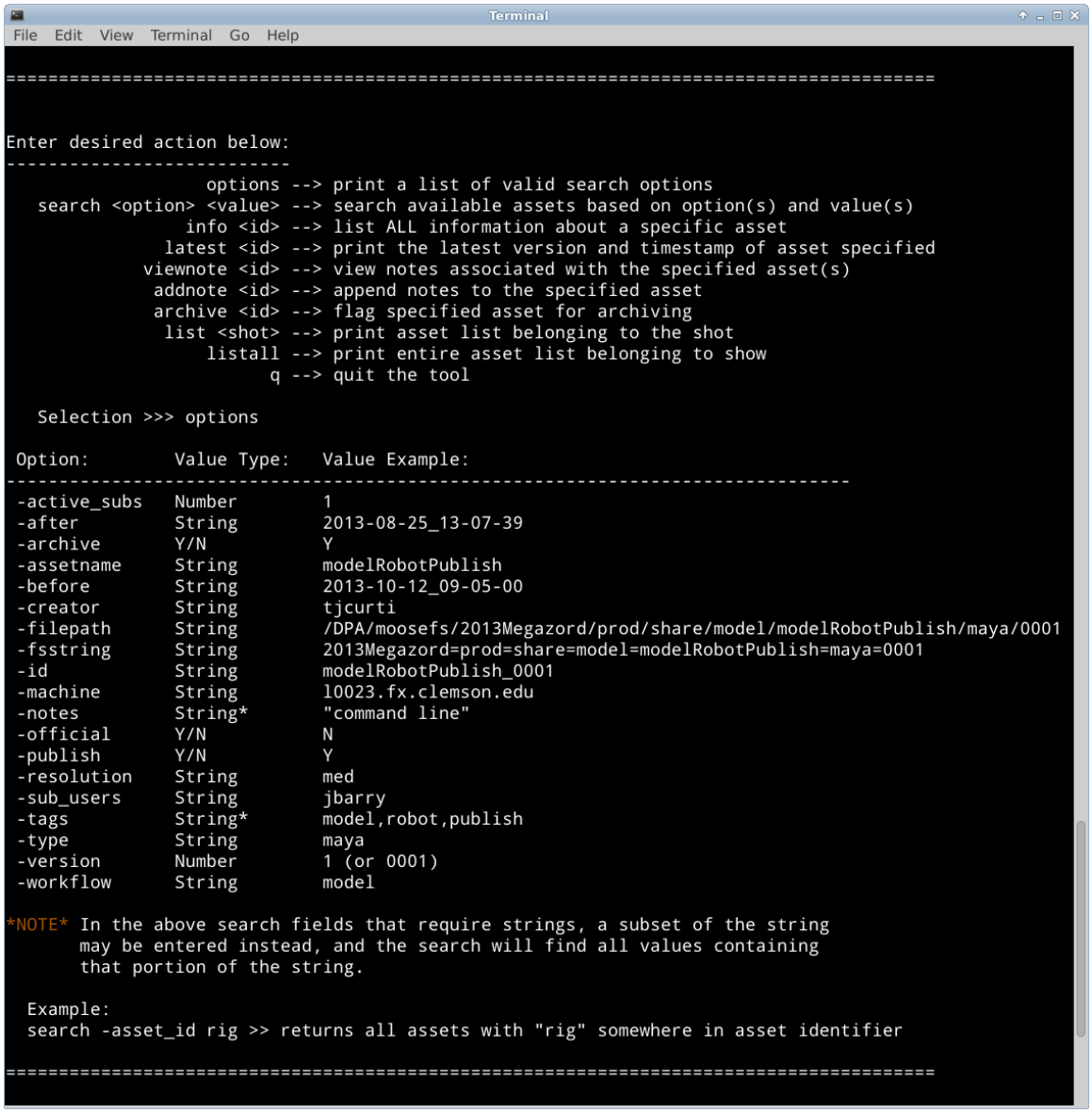


Figure 4.2: Command line asset browsing options.

4.3.2 PyQt Interfaces

A powerful tool that aided in the production of the custom software interfaces for the Maya was QT Designer. Using QT Designer, GUIs were created for the publish and subscribe actions. Each interface presents all necessary information directly to the user in an easily understandable window, and simple actions such as double clicking and pressing return would activate commands based on the fields these actions occurred in. Furthermore, these GUIs were designed with fields and objects that were explicitly compatible with the Maya Interface API. In this way, the GUIs could be executed as stand-alone applications, or loaded as Maya GUIs.

4.3.3 Maya API Integration

With compatible fields pre-determined, GUIs are loaded dynamically into Maya in a custom tab that housed proprietary plugins for the DPA program. Three separate interfaces were created for primary asset operations: publish, subscribe, and remove subscriptions. Each operation is encapsulated within a module that handles data transfer with the backend module, and displays the correct data for the artist. Furthermore, each operation is loaded into Maya through an intermediate interface that handles the dynamic signals generated from key or mouse clicks within the interface. The intermediate module also ensures that multiple interfaces cannot be created, therefore removing the potential for conflicting sets of information for the artist.

The publishing tool presents an interface to master a workflow item and create a subscribable entity in the filesystem and database. The publisher allows the artist to append meaningful metadata to the published asset, such as tags and notes, so that subscribers may have access to important information they may not otherwise have with files simply residing on disk. Additionally, after several iterations, or reaching the state in which the artist is ready to deliver a final version of the asset, the asset can also be deemed official, adding an extra tag to alert other artists of the recommended version of the asset they should subscribe to. Figure 4.3 shows the publish tool within Maya.

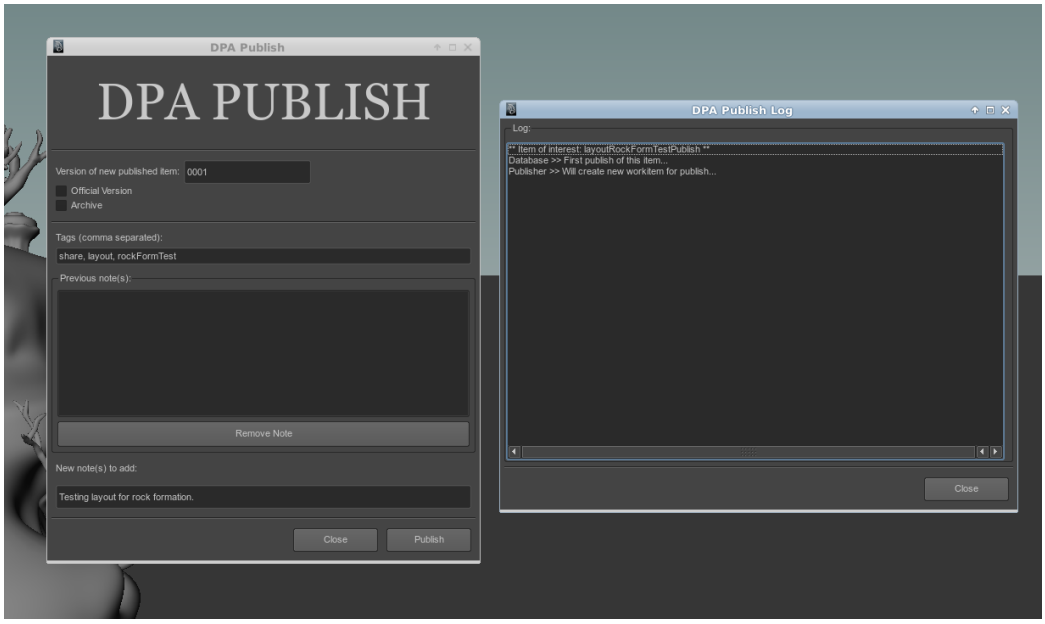


Figure 4.3: Publishing interface within Autodesk Maya.

The subscription tool gives the user several fields that filter search results when locating the assets they need for their work items. Some of these fields are automatically populated with upstream workflow information if the artist is in a downstream workflow, such as rigging or animation. Additionally, a checkbox can filter all search results by the latest version, which is generally the version most artists choose. Double-clicking an asset in the list will bring up a separate window so the artist can view any notes the publisher has attached to them.

Each time the subscription tool is launched, the current subscriptions for that workflow item are loaded and cross-referenced against the assets that exist within the file. If for some reason the working file is out of sync with the subscriptions recorded in the database, the subscription tool begins a rigorous resolution process that presents the user with options to correct the discrepancies in the file and ensure a safe working environment. Figure 4.4 shows the subscribe tool within Maya.

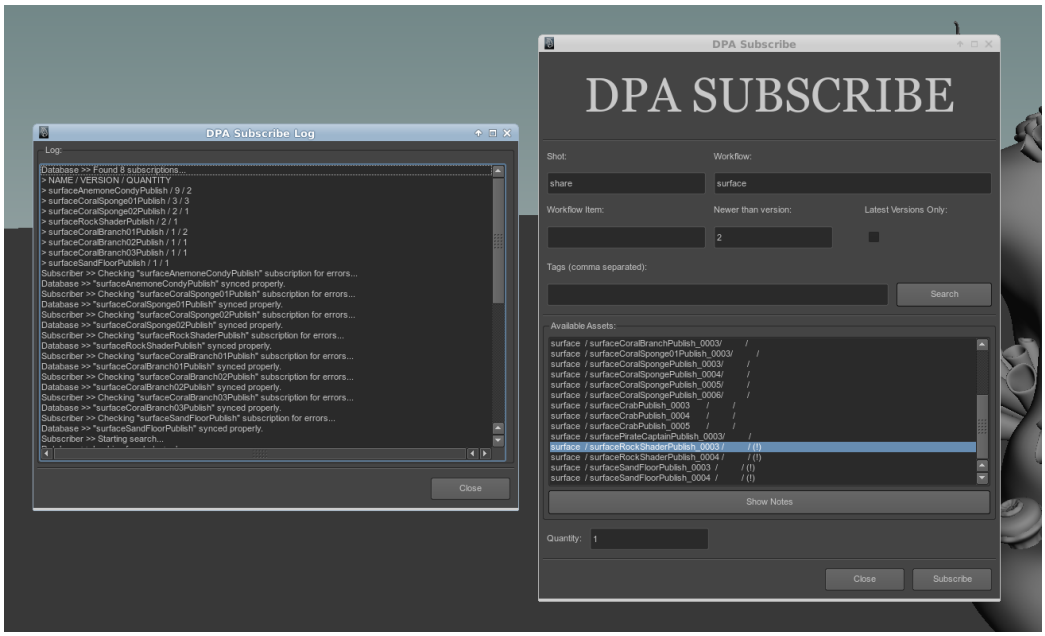


Figure 4.4: Subscription interface within Autodesk Maya.

The subscription removal tool is less complex than the previous two, and allows the artist to select any component of a referenced asset and remove, or decrement, the subscription to the asset. Many situations could occur where users subscribe to a large quantity of the same asset for scene layouts, animation, etc., and the ability to remove specific references and sync the subscriptions properly is essential for the artist to quickly iterate in those workflows. Figure 4.5 shows the subscription removal tool within Maya.

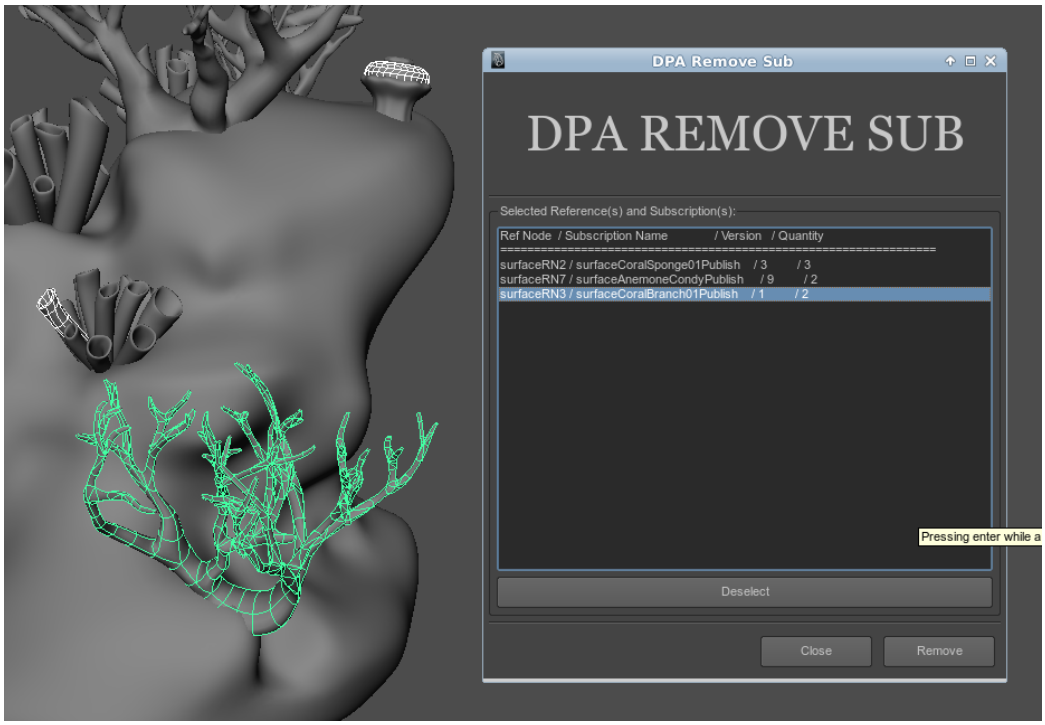


Figure 4.5: Subscription Removal interface within Autodesk Maya.

In addition to the primary interfaces that hold key information for each operation, log windows are also available. While an artist may not desire to read all the information that is printed to these windows, errors and warnings are displayed with prominent tags to alert the user of potential problems or code bugs. This includes specific information as to how the artist can correct the problem, or a specific message that can be relayed to the developer if the error was an unforeseen occurrence.

4.3.4 Nuke Tool Development

Aside from publishing and subscribing, rendering final images from Nuke is a process that requires distribution to the DPA rendering farm, CheesyQ. Two tools control the assets produced from Nuke. The first is a custom Nuke Write Node that infers asset name and file path fields from pre-existing assets in a tab within the Write Node. At render time, the node will source these fields to build the proper file path to save the rendered images. The second tool handles splitting and distributing the rendering jobs to the render farm.

The Write Node is integrated directly into the Node Graph Menu so the artist may create

the node as simply as any other without manually loading a script. Furthermore, methods are preloaded into the rendering python fields to execute specific operations before and after rendering frames to ensure the correct file directories exist and that user permissions are set correctly upon completion. Figure 4.6 shows the DPA Write Node within Nuke.

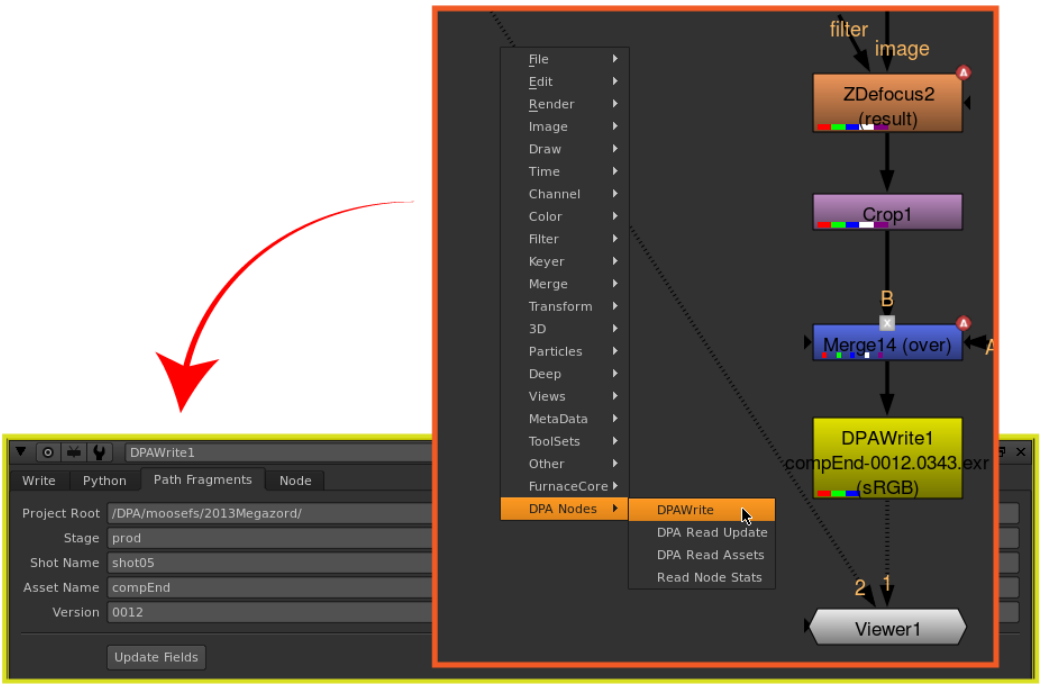


Figure 4.6: Custom Write Node within Nuke.

The distributed rendering panel is integrated into the Nuke Rendering Menu. This panel presents the artist with pre-determined rendering information, as well as the option to select how many jobs to split the render into, up to four. Jobs are immediately queued in the render farm after executing this tool, unless specifically told otherwise within the rendering panel. Each of these rendering jobs consists of a Bash script that sets the proper environment variables for the machine that picks the job up, and then invokes a python script that launches the Nuke render from the command line. Figure 4.7 shows the Render Panel within Nuke.

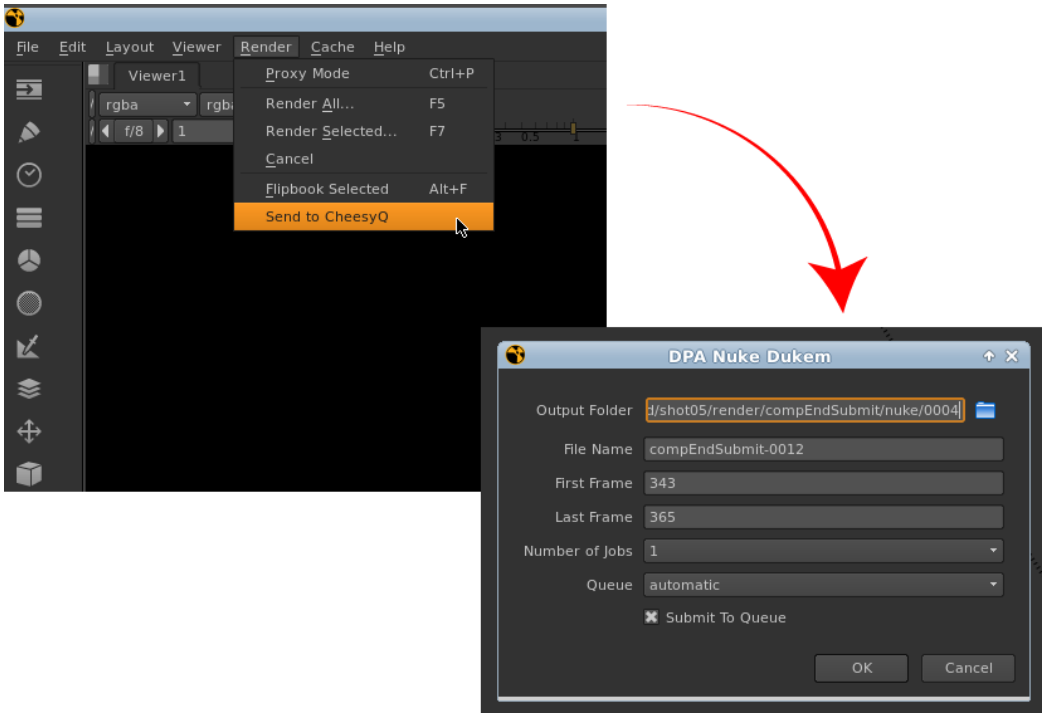


Figure 4.7: Renderfarm (CheesyQ) integration with custom Nuke Panel.

4.4 Summary of Methods

These GUIs comprise most of the frontend for the new asset management suite developed for the DPA program. Each GUI interacts directly with the MySQL database and the pipeline framework to ensure consistency between the file system and database tables. Command line tools further supplement the artists' abilities to create, view and control assets through browsing and archiving, as well as the potential for incorporating publish and subscription tasks into other tools. The database behind these tools tracks all the asset metadata and the relationships between assets to ease the burden of knowledge for the artists. Furthermore, the database, coupled with the robust MySQLdb backend, creates the opportunity to extend asset management functionality through new tools or plugins that can immediately interact with database information.

Chapter 5

Results

There are several different ways in which the presented Asset Management system has benefited the artists in the DPA program. Overall, the system automates several very repetitive tasks, and eliminates user error from manual execution. By doing so, the artists can create more iterations of their working files and assets, spending more time creating high quality visuals and less time deciphering pipeline nuances.

Figure 5.1 shows the increasing number of published items, work item versions, and asset versions created during the past six productions in the DPA program.

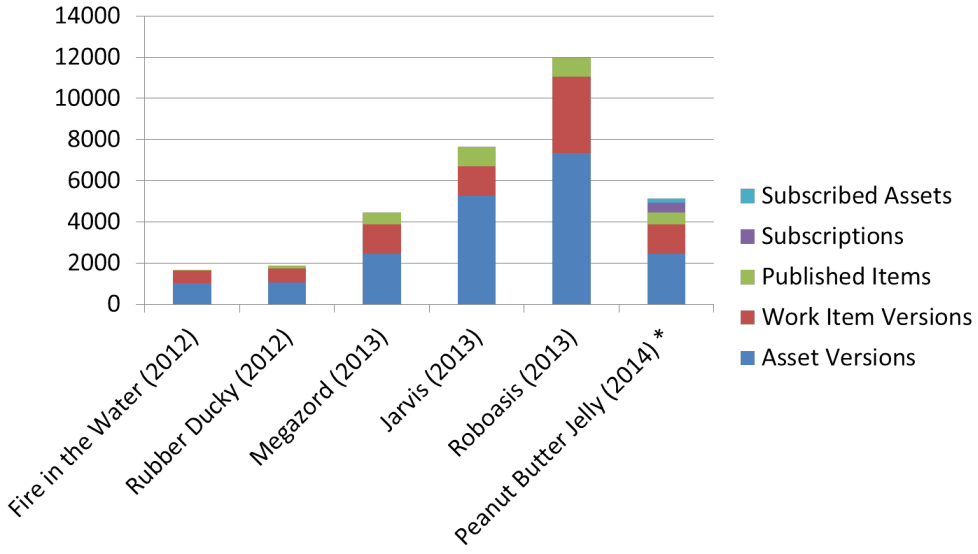


Figure 5.1: File versions created in past six productions in DPA.

Figures 5.2 and 5.3 show the average number of assets and workflow items created in the above six productions with respect to how many artists were on the team, how many shots the production consisted of, and how many workflows the production went through. [1]

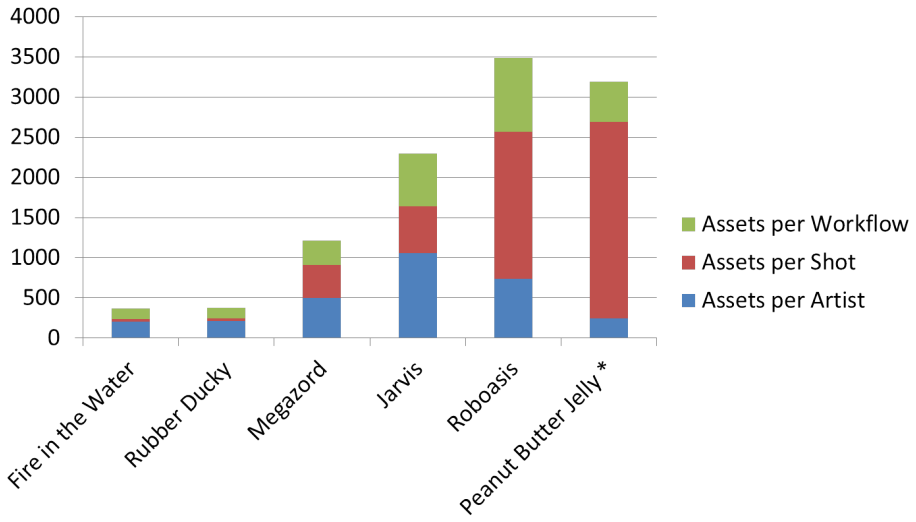


Figure 5.2: Average number of assets created during productions.

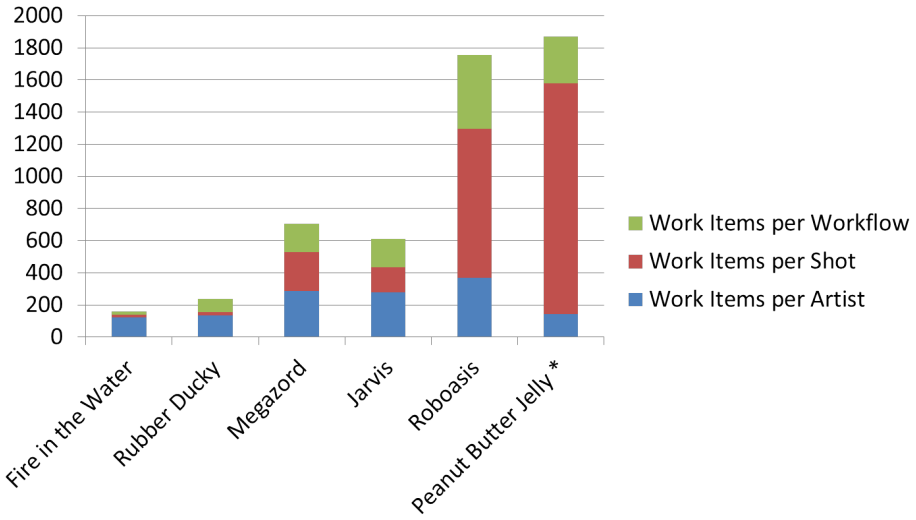


Figure 5.3: Average number of work items created during productions.

[1]The Peanut Butter Jelly production is currently still in beginning stages, and has not yet moved into shot-work or the animation workflow. Most work has been completed in the *share* shot in preparation for layout, in addition to some R&D into visual FX.

Artists in recent productions have been iterating within workflows and creating assets at a much higher rate than earlier productions. Consequently, artists in recent productions have spent more time polishing their work to produce higher quality imagery. The introduction of the production pipeline and asset management system has had a very positive effect on the final imagery and films produced in student productions. Section 5.1 discusses exact numbers and averages of the above graphs, and explains these results in more detail.

5.1 Asset Management in DPA Productions

Before asset management was recognized in the pipeline, artists were responsible for creating new versions of their assets, publishing their files, and referencing the necessary files to complete their production. They could not rely on any systems to assist with tracking these files, and any lack of communication between the artists could result in use of out-of-date assets, or manual location of the assets they needed. Both *Fire in the Water* and *Rubber Ducky* were completed in 2012 without any formal pipeline software to support them. Table 5.1 shows the total number of files produced for those films, consisting of working file versions, asset versions, and the number of published items.[2]

	Asset Versions	**Work Item Versions**	**Published Items**
Fire in the Water	1029	604	42
Rubber Ducky	1064	670	138

Table 5.1: Number of file versions created in 2012 productions.

The published items in these productions were files located at a master level, and overwritten when an artist was ready to provide others with a new version of their working file. Therefore, no version control for published items existed at this stage. The length of the of the films, with *Fire in the Water* at about 2 minutes and 30 seconds, and *Rubber Ducky* at about 3 minutes, contributed to lower number of iterations per shot, as well as the lack of a system to automate repetitive processes. There were five artists on each team, and each production was composed of 33 shots. Table 5.2 shows production averages in regards to file versions created in each show.

[2]The numbers shown do not reflect incremental saves within versions. Proper pipeline usage denotes version controlling assets in an easily distinguishable manner

	Fire in the Water	Rubber Ducky
Assets per Artist	205.8	212.8
Assets per Shot	31.2	32.2
Assets per Workflow	128.6	133.0
Work Items per Artist	120.8	134.0
Work Items per Shot	18.3	20.3
Work Items per Workflow	75.5	83.8

Table 5.2: Average number of file versions in 2012 productions.

The first use of asset management tools consisted only of early versions of Maya and Nuke interfaces to control the publishing and rendering operations. Despite the lack of the database at this point, the number of iterations artists were able to achieve through streamlining these asset operations is much higher than previous productions. The *Megazord*, *Jarvis* and *Roboasis* production teams all completed short films in 2013 using the early versions of the asset management tools. Table 5.3 shows the number of versions produced between assets, work items, and published files. At this point, published files were considered version controlled assets, and the number of published versions is separate from the number of the recorded asset versions.

	Asset Versions	Work Item Versions	Publish Versions
Megazord	2449	1437	564
Jarvis	5283	1400	982
Roboasis	7344	3697	935

Table 5.3: Number of file versions created in 2013 productions.

Both the *Megazord* and *Jarvis* teams had five artists, and each production was composed of six and nine shots, respectively. The *Roboasis* team had ten artists, and the production was composed of four shots. The lengths of these productions were far less than previous ones, with *Megazord* and *Jarvis* films at 15 seconds each, and the *Roboasis* film at 7 seconds. This results in much more work being done per shot, but more specifically, a functional pipeline and asset management system allows for each artist to iterate in their respective workflows much more. Table 5.4 breaks down the average number of versions for each production.

	Megazord	Jarvis	Roboasis
Assets per Artist	498.8	1056.6	734.4
Assets per Shot	408.2	587.0	1836.0
Assets per Workflow	306.1	654.8	918.0
Work Items per Artist	287.4	280.0	369.7
Work Items per Shot	239.5	155.6	924.3
Work Items per Workflow	179.6	175.0	462.1

Table 5.4: Average number of file versions in 2013 productions.

With a formal pipeline, proper version control methods, and the beginnings of an asset management system, artists in the 2013 productions were able to create many more assets and workflow items. Furthermore, artists could now roll back references to published assets if newly published assets contained errors. The increased number of artist iterations contributed to increased visual quality in the short films.

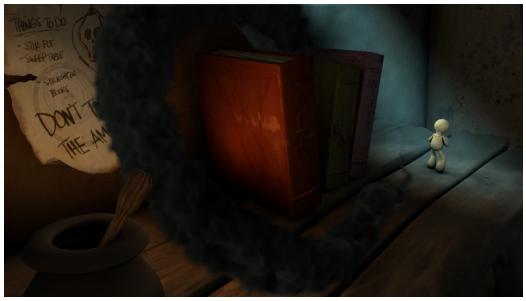
The database and supporting toolset are currently in heavy use by the *Peanut Butter Jelly* (PBJ) production in DPA. Database integration with updated tools present the artists with much more meaningful data for the publish and subscribe operations, and have reduced these operations to a series of mouse and key clicks, with occasional text editing for metadata or subscription filtering. Table 5.5 shows the current show statistics for the PBJ production.

Asset Versions	Work Item Versions	Published Items	Subscriptions	Subscription Areas
2449	1437	564	469	227

Table 5.5: Number of file versions created in the PBJ production.

Roughly 10 artists are actively working on the PBJ production, most on a volunteer basis. The number of asset and work item versions they have produced is comparable to the final Megazord number, but the PBJ production has not yet moved into the animation workflow. Artists can view asset dependencies through subscriptions, a process previously unavailable. The completed asset management system coupled with new versions of the frontend GUIs allows these artists to publish, tag, and track their assets with ease, offering users more information about production assets as well as further automated support during publish and subscribe activity.

Figure 5.4 shows production stills from all of the above mentioned productions in the DPA program. Each year has seen an increasing amount of functionality from the DPA pipeline, which has contributed to increased visual quality in more recent productions.



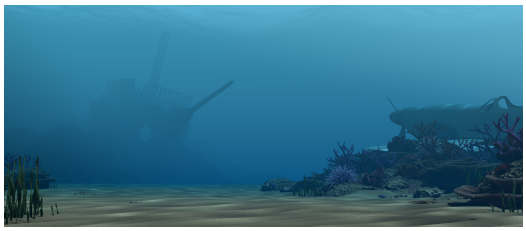
(a) Fire in the Water (*Fire in the Water, 2012*)

(b) Rubber Ducky (*The Water is Always Bluer, 2012*)

(c) Megazord (*QA-ARM-A, 2013*)

(d) Jarvis (*Robo Repair, 2013*)

(e) Roboasis (*Alien Oasis, 2013*)



(f) Peanut Butter Jelly (*Peanut Butter Jelly, 2014*)

Figure 5.4: Production stills from the past six productions in DPA.

5.2 Command Line Browsing

Command line tools allow browsing assets and subscriptions quickly and deliberately. Despite a wide variety of search options that can result in a complex database query, displaying results of of the search is much quicker than a file system crawl for the same information.

As an example, consider searching for all *rig* published items within the *PBJ* production. Figure 5.5 displays the results using command line browsing tools.

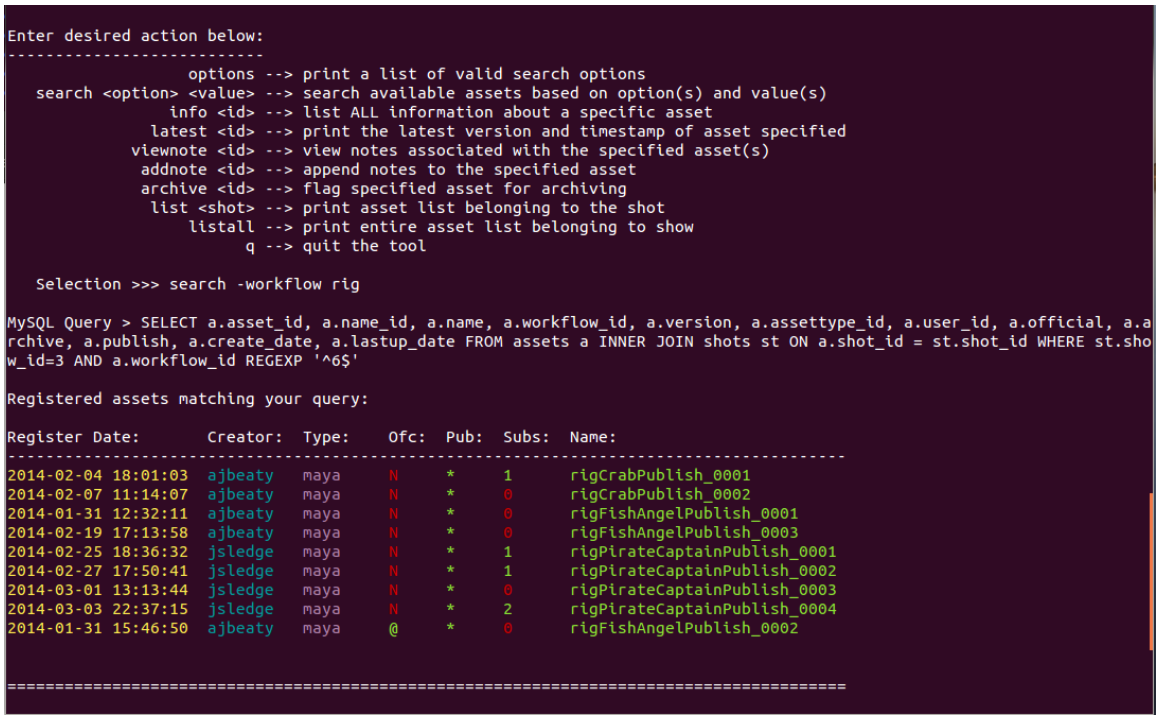


Figure 5.5: Sample asset search using command line browser.

The asset browsing tool took **0.0013 seconds** to fetch the necessary data from the database, whereas a file crawl for the same information, using Python, took **1.3 seconds**. Furthermore, the data fetched from the database holds additional asset-specific information, such as tags that can describe category or asset groupings, notes that artists appended to relay vital information, and the status of the asset. The ability to view asset location and metadata without the use of GUI adds a new, user-friendly view into the production data.

Chapter 6

Conclusions and Discussion

The Asset Management system developed with the MySQL database backend and flexible Python frontend has succeeded in facilitating asset creation, storage, and retrieval, and streamlined many repetitive tasks for artists. Modern pipelines demand efficient control of assets in productions to ensure the success of the film, as productions may very well fail without the information that a rigorous management system provides. The introduction of this system in Clemson's DPA program has given artists the opportunity to iterate more within their workflows and produce more functional and appealing assets as a result.

The MySQL database has shown no signs of performance barriers or data loss throughout its use in current student productions at Clemson. The structural hierarchy of the database tables allows for very specific queries to be executed without the baggage of unnecessary rows or information being returned from larger, more cumbersome tables. Furthermore, all fundamental operations for the publish and subscribe processes are compatible with the current database specifications.

The PyQt interfaces developed to interact with users have fundamentally changed the way assets are distributed and used throughout a production in Clemson's DPA program. A wide variety of new information is given to the artist to control, yet it is never necessary for artists to add more information than the interfaces automatically provide.

The command line extensions to the asset management system provide another view into the database with the ability to browse large amounts of assets or search for specific ones. Additionally, common asset operations such as publishing or subscribing are available through the command line, so users have the ability to integrate these processes into other related pipeline scripts.

6.1 Recommendations for Future Work

One of the first improvements that could be made to the Asset Management system is a new process or tool that automatically cross-references an artist's current subscriptions against the latest versions of those subscribed assets. This way, artists would immediately have knowledge of any out-of-date assets, and choose to update those assets at their earliest convenience. This system would be a great addition to ensure the success of the production through automated delivery of vital information.

Furthermore, the system is currently implemented for Maya and Nuke. While most of the assets with downstream dependency are produced within Maya in the DPA program, it is needed to extend the asset management functionality to Houdini, Mari, etc. The PyQt interfaces can be used in other software packages just as they are used in Maya to streamline publishing and subscribing operations. Alternatively, socket programming methods could be employed to move these interfaces into a stand-alone environment, and could communicate with any software requesting its services.

6.2 Closing Remarks

Managing the generation and use of assets throughout a production not only benefits the pipeline framework with a new set of automated rules and modules to rely on, and greatly benefits the artists with customized interfaces that provide information and processes that were never traditionally available through third-party software. The system described by this thesis has introduced a new set of powerful tools backed by a database capable of delivering information in real time. Avenues for improvements to flexibility and control exist, but the current system has proved to be a great boost in productivity for artists by allowing them to delegate pipeline concerns to a tool set very capable of handling them. By spending more time iterating through workflows rather than battling file system obscurities, artists have been able to focus on creative processes, ultimately creating better visual imagery.

Appendix: Complete MySQL Database Model

This SQL script can be executed to create all of the tables used in the database implementation, and create all of the foreign key constraints between the necessary tables.

```
SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0;
SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0;
SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='TRADITIONAL';

CREATE SCHEMA IF NOT EXISTS DPA_PSA
  DEFAULT CHARACTER SET utf8 COLLATE utf8_general_ci;
USE DPA_PSA;


-- -----------------------------------------------------
-- Table DPA_PSA.ref_assettypes
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.ref_assettypes (
  assettype_id          INT NOT NULL AUTO_INCREMENT,
  type_value            VARCHAR(32) NOT NULL,
  PRIMARY KEY (assettype_id),
  UNIQUE INDEX (type_value))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.ref_assetres
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.ref_assetres (
  assetres_id           INT NOT NULL AUTO_INCREMENT,
  res_value             VARCHAR(32) NOT NULL,
  PRIMARY KEY (assetres_id),
  UNIQUE INDEX (res_value))
```

```
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.ref_users
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.ref_users (
  user_id                 INT NOT NULL AUTO_INCREMENT,
  username                VARCHAR(32) NOT NULL,
  PRIMARY KEY (user_id),
  UNIQUE INDEX (username))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.shows
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.shows (
  show_id                 INT NOT NULL AUTO_INCREMENT,
  name                    VARCHAR(128) NOT NULL,
  status                  CHAR(1) NULL,
  create_date             TIMESTAMP NULL DEFAULT NULL,
  lastup_date             TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                            ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (show_id),
  UNIQUE INDEX (name))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.ref_stages
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.ref_stages (
  stage_id                INT NOT NULL AUTO_INCREMENT,
  name                    VARCHAR(32) NOT NULL,
  PRIMARY KEY (stage_id),
  UNIQUE INDEX (name))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.shots
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.shots (
  shot_id                 INT NOT NULL AUTO_INCREMENT,
  name                    VARCHAR(128) NOT NULL,
  show_id                 INT NOT NULL,
  stage_id                INT NOT NULL,
  description             VARCHAR(1024) NULL,
  status                  CHAR(1) NULL,
  create_date             TIMESTAMP NULL DEFAULT NULL,
```

```
  lastup_date           TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                           ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (shot_id),
  UNIQUE INDEX (name, show_id, stage_id),
  CONSTRAINT show_id
    FOREIGN KEY ( show_id )
    REFERENCES DPA_PSA.shows ( show_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT stage_id
    FOREIGN KEY ( stage_id )
    REFERENCES DPA_PSA.ref_stages ( stage_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.ref_workflows
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.ref_workflows (
  workflow_id           INT NOT NULL AUTO_INCREMENT,
  name                  VARCHAR(32) NOT NULL,
  PRIMARY KEY (workflow_id),
  UNIQUE INDEX (name))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.assets
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.assets (
  asset_id              INT NOT NULL AUTO_INCREMENT,
  shot_id               INT NOT NULL,
  name_id               VARCHAR(128) NOT NULL,
  name                  VARCHAR(128) NOT NULL,
  version               INT NOT NULL,
  workflow_id           INT NULL,
  assettype_id          INT NULL,
  assetres_id           INT NULL,
  filepath              VARCHAR(512) NULL,
  fsstring              VARCHAR(512) NULL,
  user_id               INT NULL,
  machine               VARCHAR(128) NULL,
  notes                 VARCHAR(4096) NULL,
  official              CHAR(1) NULL,
  archive               CHAR(1) NULL,
  publish               CHAR(1) NULL,
  create_date           TIMESTAMP NULL DEFAULT NULL,
  lastup_date           TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                           ON UPDATE CURRENT_TIMESTAMP,
```

```
  PRIMARY KEY (asset_id),
  UNIQUE INDEX (shot_id, name_id),
  UNIQUE INDEX (shot_id, name, version),
  UNIQUE INDEX (fsstring),
  CONSTRAINT asset_shot_id
    FOREIGN KEY ( shot_id )
    REFERENCES DPA_PSA.shots ( shot_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT asset_workflow_id
    FOREIGN KEY ( workflow_id )
    REFERENCES DPA_PSA.ref_workflows ( workflow_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT assettype_id
    FOREIGN KEY ( assettype_id )
    REFERENCES DPA_PSA.ref_assettypes ( assettype_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT assetres_id
    FOREIGN KEY ( assetres_id )
    REFERENCES DPA_PSA.ref_assetres ( assetres_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT asset_user_id
    FOREIGN KEY ( user_id )
    REFERENCES DPA_PSA.ref_users ( user_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;

-- -----------------------------------------------------
-- Table DPA_PSA.wfitems
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.wfitems (
  wfitem_id               INT NOT NULL AUTO_INCREMENT,
  name                    VARCHAR(128) NOT NULL,
  shot_id                 INT NOT NULL,
  workflow_id             INT NOT NULL,
  PRIMARY KEY (wfitem_id),
  UNIQUE INDEX (name, shot_id, workflow_id),
  CONSTRAINT wfitem_shot_id
    FOREIGN KEY ( shot_id )
    REFERENCES DPA_PSA.shots ( shot_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT wfitem_workflow_id
    FOREIGN KEY ( workflow_id )
    REFERENCES DPA_PSA.ref_workflows ( workflow_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
```

```
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.asset_tags
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.asset_tags (
  tag_id                INT NOT NULL AUTO_INCREMENT,
  asset_id              INT NOT NULL,
  tag_value             VARCHAR(128) NOT NULL,
  PRIMARY KEY (tag_id),
  UNIQUE INDEX (asset_id, tag_value),
  CONSTRAINT tag_asset_id
    FOREIGN KEY ( asset_id )
    REFERENCES DPA_PSA.assets ( asset_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.ref_tools
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.ref_tools (
  tool_id               INT NOT NULL AUTO_INCREMENT,
  name                  VARCHAR(32) NOT NULL,
  PRIMARY KEY (tool_id),
  UNIQUE INDEX (name))
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.subscriptions
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.subscriptions (
  subscription_id       INT NOT NULL AUTO_INCREMENT,
  wfitem_id             INT NOT NULL,
  tool_id               INT NOT NULL,
  version               INT NOT NULL,
  filepath              VARCHAR(512) NULL,
  fsstring              VARCHAR(512) NULL,
  create_date           TIMESTAMP NULL DEFAULT NULL,
  lastup_date           TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                          ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (subscription_id),
  UNIQUE INDEX (wfitem_id, tool_id, version),
  UNIQUE INDEX (fsstring),
  CONSTRAINT wfitem_id
    FOREIGN KEY ( wfitem_id )
    REFERENCES DPA_PSA.wfitems ( wfitem_id )
    ON DELETE NO ACTION
```

```
    ON UPDATE NO ACTION,
  CONSTRAINT tool_id
    FOREIGN KEY ( tool_id )
    REFERENCES DPA_PSA.ref_tools ( tool_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


-- -----------------------------------------------------
-- Table DPA_PSA.sub_assets
-- -----------------------------------------------------
CREATE TABLE IF NOT EXISTS DPA_PSA.sub_assets (
  sub_asset_id          INT NOT NULL AUTO_INCREMENT,
  subscription_id       INT NULL,
  asset_id              INT NULL,
  user_id               INT NULL,
  create_date           TIMESTAMP NULL DEFAULT NULL,
  lastup_date           TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
                                          ON UPDATE CURRENT_TIMESTAMP,
  quantity              INT NOT NULL DEFAULT 1,
  filelock              CHAR(1),
  PRIMARY KEY (sub_asset_id),
  UNIQUE INDEX (subscription_id, asset_id),
  CONSTRAINT subscription_id
    FOREIGN KEY ( subscription_id )
    REFERENCES DPA_PSA.subscriptions ( subscription_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT sub_asset_id
    FOREIGN KEY ( asset_id )
    REFERENCES DPA_PSA.assets ( asset_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION,
  CONSTRAINT sub_user_id
    FOREIGN KEY ( user_id )
    REFERENCES DPA_PSA.ref_users ( user_id )
    ON DELETE NO ACTION
    ON UPDATE NO ACTION)
ENGINE = InnoDB;


SET SQL_MODE=@OLD_SQL_MODE;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS;
```

Bibliography

[1] Dpa tools, 2013. http://wiki.fx.clemson.edu/mediawiki/index.php/DPA_Tools.

[2] Database, 2014. http://en.wikipedia.org/wiki/Database.

[3] Relational model, 2014. http://en.wikipedia.org/wiki/Relational_model.

[4] D. E. Bettis. Digital production pipelines: examining structures and methods in the computer effects industry. Master's thesis, Texas A&M University, May 2005.

[5] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[6] D. Donar. Production pipeline chart, 2009. http://dpa860.wordpress.com/2009/09/10/production-pipeline-chart/.

[7] R. Dunlop. *Production Pipeline Fundamentals for Films and Games*. Focal Press, Burlington, MA, USA, 2014.

[8] J. Geduldick. Review: Shotgun software. *Post Magazine*, May 2010. http://www.postmagazine.com/Publications/Post-Magazine/2010/May-1-2010/REVIEW-SHOTGUN-SOFTWARE.aspx.

[9] C. Johnson, J. Tobiska, J. Tomlinson, N. Van den Bosch, and W. Whaley. A framework for globalized visual effects production pipelines, *SIGGRAPH Talk Submission*, 2014.

[10] A. Lith and J. Mattsson. Investigating storage solutions for large data. Master's thesis, Chalmers University of Technology, June 2010.

[11] R. Noteboom. Configurable workflow systems. Technical report, Southpaw Technology, 2013. http://digitalassetmanagementnews.org/features/configurable-workflow-systems/.

[12] R. O'Neill. Building the perfect production pipeline. *Computer Graphics World*, 32(8), August 2009.

[13] R. ONeill, P. Mavroidis, and M. Ho. openpipeline: Teaching and implementing animation production pipelines in an academic setting. In *ACM SIGGRAPH 2007 Educators Program*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[14] R. ONeill, P. Mavroidis, M. Ho, and G. Elshoff. Openpipeline specification, 2007. http://openpipeline.cc/pipeline-specification/.

[15] J. Tessendorf. Private communications, 2014.

[16] J. Thacker. Shotgun software unveils tank asset manager, August 2012. http://www.cgchannel.com/2012/08/shotgun-software-unveils-tank-asset-manager/.