# Pipeline Development Roadmap

James M. Callahan

9th February 2004

## Contents



# 1 Overview

This document outlines the future course of Pipeline development over the next couple months. It contains solutions to the problems encountered during use of the beta version of Pipeline on the ATI demo project at RhinoFX. This will serve as a guide to the tasks remaining before Pipeline is ready for its initial commercial release.

# 2 Performance

## 2.1 Database

The SQL database will be replaced with a custom server program written in Java which will be able to represent the tree-like data structures inherent to Pipeline in a vastly more efficient manner. Much of the time currently spent by users waiting on status updates while using *plui(1)* and *pipeline(1)* is related to the large numbers of queries required to traverse tree-like data strucutures in SQL. Also, some of the critical queries where themselves very costly and didn't scale well for large numbers of users or nodes.

The replacement database will be composed of a set of Java objects which can be saved and restored from disk in a human readable text format. The database server will provide a high level interface to manipulating these Java objects and manage all thread and transaction related issues internally.

## 2.2 File System

Whenever a user requests an status update, e needs to examine potentially large numbers of files in order to compute a correct status for the associated

nodes. The information required for these files usually is fairly lightweight and consists of determining file existence, file size and last modified time stamps. In some cases, an expensive checksum may need to be generated for files which have been modified. In both cases though, this file system activity is taking place over the network via NFS. One of the problems with this approach is that the latency associated with checking file status over NFS becomes significant when hundreds of files are involved. Even more problematic is the cost of moving entire files over the network merely to generate checksums. Lastly, even though a complex tree of nodes may involve hundred of associated files, it is highly unlikely that all of the files have actually changed since the last status update. Currently, every file is checked with every update regardless if it has changed or not.

A possible solution to the network latency and transmission cost issues would be to use a custom file status server program to manage file changes. This file status server program would be run on the file server, this way all files related to PIPELINE would be accessed locally and bypassing NFS altogether. If the file server hardware was incapable of running the file status server program (NetApp), it could be run on a dedicated machine with very fast access to the file server. This server program could check large numbers of files with much lower network activity and latency. If checksums need to be generated, the source files would not have to be transmitted over the network.

The problem of redundently checking files which haven't changed can be solved by using a feature of the Linux kernel called DNotify. This feature allows programs to register interest in certain directories and be notified (via signals) when files are created, deleted and modified in these directories. The file status server could maintain an internal table of information related to files associated with active nodes and only update this table when notified by the kernel that a change has occurred.

### 2.3 Lightweight Clients

When a user requests a status update, PIPELINE currently must gather and analyze a large amount of database and file system data to compute a small amount of state information for each node. The source data for this computation must be moved over the network. In addition, a large percentage of this source data is usually accessed multiple times and possibly by more than one client program between changes to the source data.

Status updates could be made more efficient if they where computed lazily on the server-side in response to file system change notifications, node operations and queue activity. Only those nodes which are actively being monitored by Pipeline clients programs would need to be updated. When new node states are generated on the server, only the changed state information would need to be sent to clients.

Another benefit of this approach would be the elimination of the need for users to manually request the state of nodes. Instead, the client programs would simply always display the latest known state information without repeated user status requests. This should improve useability and eliminate some user confusion related to viewing stale node state information.

## 3 Nodes

### 3.1 Multiple Working Areas

Currently, each user has a working directory where all files associated with their working copies of nodes reside. This prevents changes to working area files from being seen by other users and allows each user to have their own version of a particular node. However, each user can only have a single copy of a given file in their working area at one time. This is usually not a problem, but there are some cases where this limitation causes difficulty. For example, a user might be rendering a Maya scene, but wants to be able to work on textures or model files being referenced by the running render.

The solution is to allow user to have more than one working area. Users should be allowed to create, destroy and change the active working area on demand. Each working area will be treated as if it was a seperate user. Changes made in one working area

can be communicated to other working areas using the usual check-in/out mechanism.

Currently, PIPELINE provides an environmental variable called "WORKING" which contains the path to the current user's working directory. To support multiple working areas, this variable will be replaced with a pair of variables "PLWORK" and "PLVIEW" which when used together (i.e. "$PLWORK/$PLVIEW"). When editors or regeneration actions are launched by PIPELINE, these variables will be set to point to the proper working area directory for the node involved.

## 3.2 Node References

Nodes can currently only be connected based on dependency relationships. Changes to upstream nodes cause all downstream nodes to become stale and require regeneration using the job queue. This works well for the core functionality of Pipeline, but is overly strict or undesirable to represent all kinds of node relationships.

Node references provide a way to connection nodes which have a relationship which isn't a dependency. Instead of having the dependency OPTION and OFFSET parameters, references will have a TYPE parameter. Users can create new reference types on demand to document the nature of the relationship.

When displaying trees of nodes, each type of reference can be optionally displayed or ignored. Reference links will be displayed differently than dependency links and will be labeled with the reference type name.

Another difference between dependencies and references is that nodes on the upstream side of a reference link will not affect the state of the downstream node and will not be regenerated by "make" operations initiated on the downstream side of the link.

## 3.3 Non-Stale Property Modifications

Whenever any node property is changed, it causes node's with regeneration actions to become stale and therefore require regeneration before they can be checked-in. In most cases this is desirable, since the property changes are not reflected in the files associated with the modified node. However, there are some node properties (editor, job requirements, etc...) which cannot affect the contents of the associated files and therefore should not cause the node to become stale when they are changed. However, the node would still become modified when one of these properties is changed to that these changed are properly handled by the check-in/out system.

## 3.4 Node Tree Display and Layout

Node trees are currently displayed by starting at some picked node and traversing the upstream dependency links until all leaf nodes are reached. Any nodes reachable by more than one path are displayed multiple times. There is a single viewing area which can display more than one tree can at a time.

This will be replaced by multiple tabbed viewing areas in which each view contains a single tree of nodes. Two new layout styles will be added to the current upstream layout style. The downstream layout will display the inverse tree of nodes which depend on a single upstream node. The bidirectional layout will display both upstream and downstream trees centered on a single node.

Any particular viewing area should be easily toggled between these three layout styles. Each style uses a single key node as the starting point for the layout. When the layout style is changed, the key node will remain the same. When a node is selected for display (left side of *plui(1)*) they replace the key node for the currently selected viewing area tab. In addition, any currently viewed node can be made the key node for a particular viewing area tab. Tabs can be added and removed on demand much like the tabs in Mozilla.

Only the currently viewed tab will be considered when communicating with the database and file servers to update node status. In other words, there would be no overhead to having many tabs since only one of them is active at any one time.

One key difference in terms of status update is that nodes downstream of the key node will not have their status evaluated. These downstream side nodes will be rendered gray and without status symbols since their status is undefined. All node operations (check-

in/out, make, etc..) will also be disabled for these nodes. The reason for this restriction becomes clear when you consider the huge number of nodes which would need to be considered when the key node is a commonly used dependency such as a model node. Computing the status of the entire tree would then become equivalent to computing the status of all node trees which reference the key node. If status information is desired for one of the downstream side nodes, it can be obtained by changing the key node to one further downstream.

### 3.5 Node Deletion

Once a node has been checked-in, it cannot ever be removed. This is usually exactly what is desired, since the permenancy of repository versions is one of the core functions of PIPELINE. However, sometimes a user creates and checks-in a node which is badly named to placed in a awkward location. Ideally, these kind of mistakes should be noticed before checking-in the node, but sometimes mistakes are made.

PIPELINE should provide a way to destroy a node and all of its checked-in repository versions and associated files to deal with problem nodes. All references to the deleted node would also be removed. This operation should only be accessable to priviledged users, such as system administrators or supervisors, since irreparable damage could be done by deleting the wrong node.

### 3.6 File Deletion

Pipeline should provide an operation which would remove a specific file or all files assocated with a node. This is sometimes desirable when the regeneration action has touched the file, but then crashed or otherwise failed to finish writing the file properly. Currently, Pipeline would consider the file to up-to-date and will refuse to regenerate it. This file deletion operation provides a way work around problems of this kind.

## 4 Job Queue

### 4.1 Incremental Output

Each line of output from standard out/error of running jobs will be captured as it is output from the job. Users will be able to monitor this output while jobs are running by opening a dialog for the job. Each line of output will be transmitted via a network connection from the job monitoring daemon to each PIPELINE client with an open job output dialog. Upon job exit, all output will be stored in the database as usual.

### 4.2 Job Failure from Output

Some programs, noteably Maya, fail to return a nonzero exit status for some failure conditions. What is needed is a flexible way to parse the output for signs that something has gone wrong and possibly even kill the job. For example, missing texture files will not cause Maya to fail but the results of allowing the rendering to continue are less than ideal. It would be good to be able to optionally request that Pipeline treat a missing texture as a failure and thereby avoid further downstream processing.

A solution would be to add a method to the regeneration action plugin API which would be called after each line of output from the action's suprocess and return a boolean value for whether processing should continue. The default implementation would always return true. Actions could provide controls over the output parsing as action parameters.

## 5 Customization

### 5.1 Tool API

PIPELINE models the production process in a fairly abstract manner. Features common to any conceivable production are implemented as core functionality. However, it is often the case that a particular production or studio has specialized needs which may not be generally desirable to include as a core feature of PIPELINE.

The current version supports three mechanisms for extending PIPELINE: editor plugins, action plugins and batch scripts. PIPELINE comes with a rich set of editor and action plugins to support the common industry programs and these can be extended by writing a custom Java classes. Batch scripts allow a set of Pipeline operations to be scripted and executed on the command-line, but the limitation of communicating with command-line arguments makes doing more complicated operations difficult.

To support more arbitrarily complex operations and avoid awkward command-line syntax, a new Java based API will be provided wich gives access to nearly all PIPELINE functionality. This API can be used in two ways. Standalone Java programs can use the API to make new kinds of PIPELINE clients. Tool plugins can be written which when loaded by *plui(1)* will provide additional high-level functionality to the user. When loaded as a plugin, the tool will be able to create its own dialogs for getting collecting input from the user.

Tools could be written using this API to implement functionality such as setting up a commonly used heirachy of nodes, performing a repetitive operation on a large number of nodes or communicating with third party software.

### 5.2 Editor and Action Plugins

The current set of core editors and actions is incomplete. Notably support for Softimage, RenderMan and Mental Ray needs to be added. The fcheck(1) editor plugin is less than ideal and other image playback programs need to be supported. Some common desktop text editors such as kedit(1) need to be added.

There is also a problem with the way editors are launched which causes them to be killed when plui(1) exits. This is usually not desirable and can sometimes be very damaging when the user has not saved their work.

### 5.3 User Preferences

User preferences should be extended to support custom colors for all UI elements, time/date formatting and hot keys for all PIPELINE operations and UI navigation menu items.

## 6 System Administration

### 6.1 Backup and Restore

Backup of the repsitory files should be more selective than simply backing up the entire repository directory. Once a particular version of a file has been backed up, there is no need to process it every backup cycle. PIPELINE should provide support for determining when a repository file was created and produce an incremental listing of files which need to be backed up. The database should also store a list of backup tapes or other media which contain a particular repository version. Also, there needs to be a way to backup a running PIPELINE database without stopping the database server.

Similiarly, there is currently no support for using the database to determine which files can safely be offlined or how to locate and restore offlined files associated with nodes when needed. The database representation of nodes need to be extended to contain information which will identify the tape or other permanent storage media which contains an offlined set of repository files. Some additional PIPELINE commands are needed to identify rarely used nodes and package up the needed data to archive these nodes. This functionality should be supported by pipeline(1) so that it can be used in automated setting such as cron(1).

## 7 Documentation

### 7.1 User Manual

There already exists man-page style documentation for all command line based PIPELINE programs, but this is inadequate to explain the bigger picture of how PIPELINE is actually used in a production environment. A user manual is needed which will explain the PIPELINE model of production and the philosophies behind the design decisions. The user manual should also include explanations of the graphical

components of *plui(1)* as well as a set of tutorials demonstrating the PIPELINE approach to some common CG tasks.

## 7.2 System Adminstration Manual

System administrators need a document which includes information about how to install, configure and support PIPELINE for their site. This manual should also include a section on the backup and restore issues mentioned above.

## 7.3 Developers Manual

Writers of Java based Editor, Action and Tool plugins need a manual which includes code examples and tutorials. Information about compiling and installing this plugins is also needed.

## 7.4 Javadoc for APIs

A standard set of Javadoc based web pages needs to be generated for the Java classes which make up the PIPELINE APIs for Editor, Action and Tool plugins.