# Using Perforce Attributes for Managing Game Asset Metadata

Mark Allender
Volition/THQ
mark.allender@volition-inc.com

## Introduction

Many modern video games are built with hundreds of thousands of different kinds of assets. Volition has written a system that uses Perforce attributes to organize and manipulate these assets. Attributes are used for many different functions, including categorization, dependency graphs, and source-to-target asset conversion information. This talk will focus on how we settled on using Perforce attributes and how we dealt with many of the hidden complexities such as branching, resolving, and performance.

## History

In early 2006, an internal group at Volition, called the Core Technology Group (CTG), was formed to create tools, pipelines, and technology to meet the following goals:

- Provide as much cross project similarity between the tools and pipelines as possible
- Enable more efficient code reuse
- Provide more efficient tools to artists and designers (faster content iteration, tool ease-of-use, etc)

One of the fundamental foundations of these tools was to provide ways in which we could easily manage, categorize, and manipulate the myriad of assets that are part of video game development. Before this project, assets were managed by on-disk structure, various off the shelf packages (such as ACDSee), and several internally build tools that hooked directly into Photoshop or 3DS Max.

Extra asset information (metadata) was either stored in the filename itself, or in text files that sat alongside the assets. There were various drawbacks to these solutions: encoding information in the filename itself was prone to error, and having configuration files sit alongside asset files merely increased the number of files visible in the developer's workspace. Additionally, an assets configuration file could be out of date (i.e. not synced to head) and thereby providing incorrect metadata. The idea of storing metadata in a global database was appealing not only because it removed many of the annoyances in the current pipelines, but it also centralized the storage in a more natural format for this data.

## Hardware

During the initial phases of tool development, Volition was using a Windows based Perforce server:



| | |
|---|---|
| **Server** | Dell PowerEdge 2950 |
| **Processor** | 2x Intel Xeon X5140 @ 2.33GHz |
| **RAM** | 32 GB |
| **Operating System** | Windows Server 2003 |
| **Metadata** | Local storage (RAID-5) (NTFS) |
| **Repository** | Redundant NetApp iSCSI switches (failover only) (NTFS) |

In early 2010, Volition moved to a Unix server with the following specs:

| | |
|---|---|
| **Server** | Dell PowerEdge R710 |
| **Processor** | 2x Intel Xeon X5570 @ 2.93GHz |
| **RAM** | 74GB DDR2-1066 |
| **Operating System** | SuSE Linux Enterprise Server 10SP2 (x86_64) |
| **Metadata** | 6x143GB 15k SAS (RAID-10) (XFS) |
| **Repository** | Redundant NetApp FC switches in multipath configuration (XFS) |

Current database size is 126G. Almost all testing and implementation was done on the Windows server.

## Early Investigation

An obvious method for storing metadata was a database. A database provides a central location for storing large sets of data, is scalable, and has fast response for both storage and retrieval. We investigated two options for storing metadata:

- Use of a database (MySQL, SQLite, etc) alongside Perforce.
- Use of Perforce attributes

Perforce attributes were a relatively new and undocumented feature. Note that attribute related commands, specifically 'p4 attribute' and 'p4 fstat –Oa' are still undocumented, although as of release 2010.1 help for 'p4 attribute' has made its way out of undoc.

For a variety of reasons, an initial implementation was chosen that stored all asset metadata in a database while storing asset content in Perforce. Several drawbacks were eventually discovered:

- The database schema grew complicated, especially when support for branching was added
- Support for outsourcers was worrisome, specifically how asset metadata would be merged between two existing and active databases
- Edge case coordination between Perforce and a database proved difficult, especially with respect to the idea of



atomic commits

After pursuing this path for over a year, the decision to revisit the fundamental architecture was made. Perforce attributes were examined in more detail. After numerous emails to Perforce Support, the decision was made to move to Perforce attributes exclusively for metadata storage. This decision occurred in early 2009.

Specific performance information on attributes will be covered later in this paper.

# Initial Findings

## General Use of Attributes

The use of attributes themselves is straightforward. The two key commands are:

- p4 attribute – Sets/clears attributes on a file (or files). Attributes can be set as propagating or not. Propagating attributes will be set for current and all future revisions of the file, unless explicitly removed.
- p4 fstat -Oa – Gets attributes that are set on a file (or files).

Perforce Support warned on several occasions that the use of fstat was relatively slow, and that the use of attributes was even slower (again, relatively). It is important to put those relative terms into perspective. Performance issues regarding attribute usage is expanded in detail later in this paper.

The initial plan was to store all metadata as attributes in the Perforce database, and then use 'p4 fstat' to retrieve information from the Perforce server. At the time these investigations started, we were running version 8.2 of the Perforce server.

## Basic Searching

One of the main features of our toolset required searching for files that had a specific attribute (or group of attributes) set. More specifically, plans also called for users being able to specify fairly complex search queries in order to limit the type and kinds of assets visible to them in the tools. We initially looked at using 'p4 fstat -A <pattern>' which allows for restricting attributes returned from the fstat command to those that matched the given pattern. In practice however, there is no way to specify search patterns that were complex enough for our needs. Therefore, use of the –A flag for searching attributes was not a viable option.

We also looked at use of the fstat –T flag. This flag limits field output to those fields specified with the flag. Running fstat as follows: 'fstat –T "depotFIle" <filename>' will produce only a single line of output – the depot filename. (p4 help fstat outlines possible values that can be used with –T). Attributes are prefaced with 'attr-'

Below are some examples of fstat output using the –T flag:



```
>p4 fstat -Oa brute_miniD1.tga
... depotFile //sr3/main/sr3/data/maps/weapons/brute_miniD1.tga
... clientFile d:\projects\sr3\main\sr3\data\maps\weapons\brute_miniD1.tga
... isMapped
... headAction edit
... headType binary+l
... headTime 1303849043
... headRev 7
... headChange 868571
... headModTime 1291320872
... haveRev 7
... attr-resourcelib-builtin-categories dir__data__maps__weapons
... attr-resourcelib-builtin-display_name brute_minid1.tga
... attr-resourcelib-builtin-resource_type texture
... attr-resourcelib-builtin-timestamp 5599980395809788819
... attr-resourcelib-int-alpha_channel 0
... attr-resourcelib-int-height 128
... attr-resourcelib-int-texture_separate_high_mip 1
... attr-resourcelib-int-width 128
```

To use the –T flag with attributes, the entire attribute name must be known, meaning that –T cannot be used to return all attributes. Below are examples:

```
>p4 fstat -Oa -T "depotFile" brute_miniD1.tga
... depotFile //sr3/main/sr3/data/maps/weapons/brute_miniD1.tga

>p4 fstat -Oa -T "attr" brute_miniD1.tga
Field attr doesn't exist.

>p4 fstat -Oa -T "attr*" brute_miniD1.tga
Field attr* doesn't exist.

>p4 fstat -Oa -T "attr-resourcelib-builtin-resource_type" brute_miniD1.tga
... attr-resourcelib-builtin-resource_type texture
```

Use of the –T flag is useful in limiting the amount of data that needs to be parsed, but it can really only be used when specific attributes are needed.

Additionally, consideration was given to using the –F flag with the fstat command to help filter results. Like the –T flag, the attribute name has be known in advance, and like the –A flag, the filters could not be made complex enough for our needs.

Ultimately, the decision was made to copy attribute information in a database local on developer machines that would essentially serve as a searchable cache for metadata information. These changes were made to our internal tools approximately two months before p42db was released. Due to internal time constraints and availability, this option from Perforce was not, and has still not been fully evaluated.

## Use of Local Database

As previously mentioned, attribute information is stored in a local SQLite database. This method provides several benefits:

- Reduced load on Perforce server
- Ability to form more complex queries to find specific attribute information



The updating of the local database works as follows:

1. 'p4 have' is run to determine which revision is synced to the client
2. Versions returned from the 'have' command are compared against the version of the file's attributes that are stored in the local database
    a. A list of new and updated files is generated. 'p4 fstat -Oa' is run on this list of files and the local database is updated with the attribute information returned from this command
    b. A list of deleted files is generated and all attribute information for these files is deleted from the local database
3. 'p4 opened' is run to determine which files are open for add. Attributes from files opened by the user are inserted into the local database
4. The filesystem is then monitored for filesystem changes
    a. 'p4 fstat –Oa' is called for all new files that are written to the local filesystem (i.e. through a sync). The results are entered into the local SQLite database
    b. Files that are deleted locally (i.e. though a sync) have their attribute information removed from the local database.
    c. Files that move from writable to read-only (i.e. through submit) have their attribute information stored in the local SQLite database.

When the tools need attribute information for a file or set of files, this information is always retrieved from Perforce via the 'p4 fstat' command. When the tools need a list of files that match a set of attribute criteria, then the local database is searched. The local database is not a trusted source of information, so when specific information about files is needed, Perforce is the authority.

The SQLite database updates are performed in a background thread only when the toolset is running.

Our experience is that this combination of Perforce and the local database provides excellent performance, as the majority of operations for any given user is on files that are not open for edit, keeping the load off of the Perforce server. The fstat commands that do hit the Perforce server do not currently have a negative impact on server response.

## Performance

Perforce Support cautioned us that attributes were relatively slow. Relative speed can be put into context by understanding a bit about how attributes are stored. Attributes are stored in the db.traits table. According to Perforce, they are stored randomly on disk, meaning that even if two files sit side-by-side in the depot, their associated traits could be far apart in the traits table. Therefore, relative performance is really a measure of disk seeks for attributes vs. scanning (potentially) contiguous blocks in other database tables keyed.

In practice, performance of attributes has not been an issue.

# Integration of Attributes

## General Information

One of the biggest problems that had to be addressed was integrating assets with metadata between branches. Unlike



text files, there is no “merge” capability for attributes. For any give source/target file in an integration, the following might apply:

- Attribute/value pairs are the same on the source and target file
- The set of attributes is the same between the source and target file, but the values of those attributes are different (either changed on the source side, the target side, or both)
- New attributes have been added to either the source or target side
- Attributes have been deleted from either the source or target side

The toolset had to work without user intervention if possible. The tools were designed so that people without any technical skill in using Perforce would still be able to work efficiently in the tools. Writing a UI for merging attributes was ruled out quickly and the decision was made to use a trigger to fix up attributes on files that were resolved.

## Attribute Trigger

The goal of the attribute trigger was to “do the right thing” with respect to attributes and integration targets. A set of rules was defined by the team which specified under what conditions attributes would be copied from the source to the target file during an integrate.

The trigger behaves by the following rules:

- Attributes are preserved on target files whose resolve action is ‘branch’. (Perforce will preserve attributes on a target file whose resolve action is ‘branch’ by default – no action was needed in this case.)
- If there are no attributes on the source file, attributes on the target file (if they exist) will not be changed, even if resolve action is copy_from
- If there are no attributes on the target file, the attributes from the source file will be copied to the target file even if the resolve action is ignore
- For all other types of resolves:
    - All other attributes (new and existing) values are copied to the target file (again using the ‘p4 attribute’ command)
    - Attributes are never removed from the target file
- If attributes on the target file have been changed:
    - The list of attributes that have been changed are stored in another attribute so that they are not touched in future resolves. This attribute is stored as non-propagating
    - The changelist is rejected. This step is required so that attributes will properly be attached to the target file. (This step was only added later. The reasoning behind this step is explained below.)

This trigger was originally a change-commit trigger, but it was determined that attribute changes made in the trigger were lost on the target file. The trigger was then applied during the change-submit phase. This method seemed to work well for quite a while. The benefits to putting this trigger in place during change-submit is that attribute changes could be applied to a file during the trigger and those changes would stick with the file after the submission was complete. When the trigger was run as a change-commit or a change-submit trigger, attribute changes were “lost” after the



submission was complete.

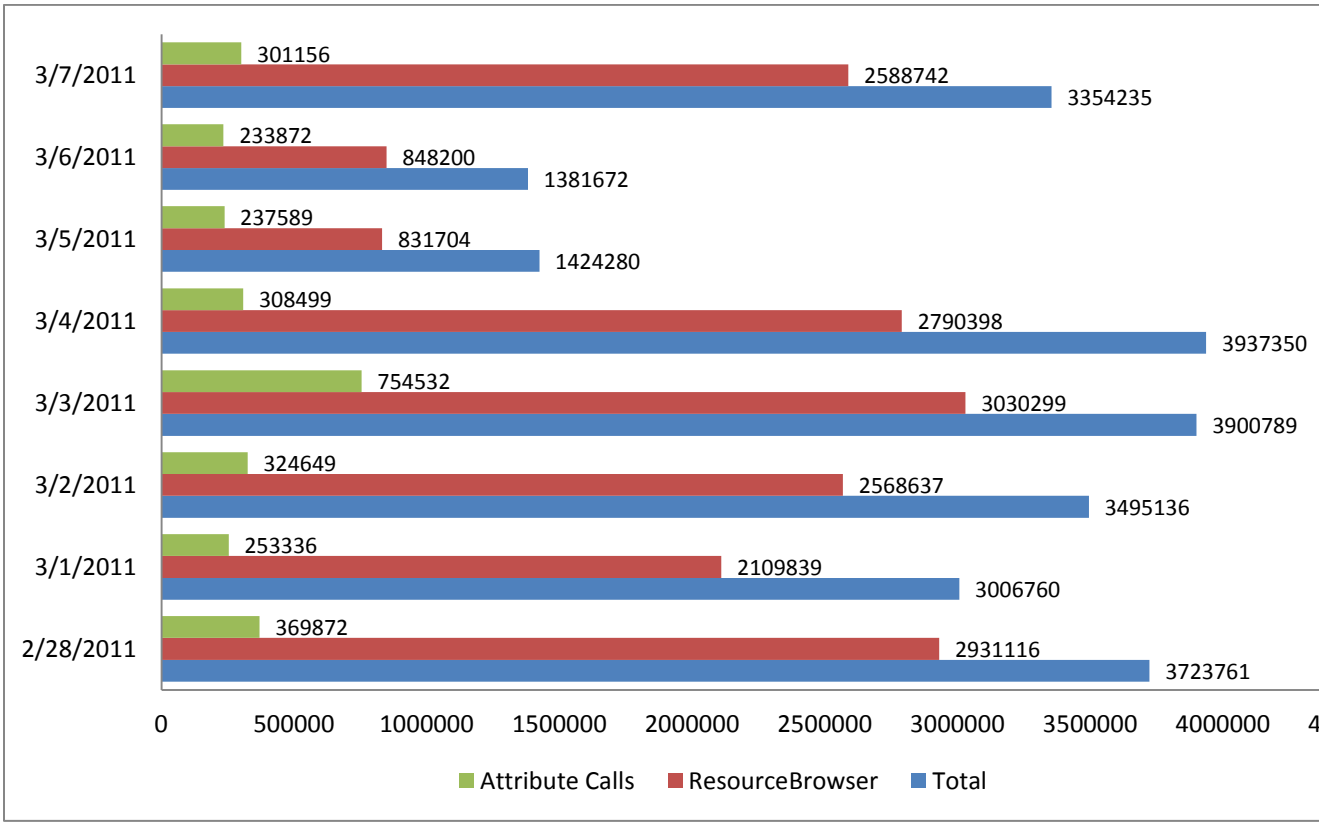
Unfortunately after several months of having this trigger in place, we started seeing librarian errors on the server where the actual repository files were missing on the server, even after a successful submit. It turns out that some file content was being submitted that was unchanged from the version in the depot, but these files had attribute changes. There appeared to be some "disagreement" on the server as to whether the file should be reverted (due to 'revert-unchanged' being specified in these cases), or submitted, due to the presence of changed attributes on the file. This problem manifested itself in a new revision in the history for the file, but no corresponding file in the repository.

The two solutions that presented themselves were to use 'submitunchanged' on all submissions, or to move the trigger to a change-submit trigger (and then reject the submission due to changed attributes) forcing some users to submit a changelist twice. The second method was chosen as the use of 'submitunchanged' had the potential to drastically increase the size of the repository due to the number of assets in the depot.

The above librarian error has not been fixed as of version 2010.1 so the attribute trigger still remains of type change-submit.

## Supporting Data

The chart below shows an overview of one week of activity on Volition's Perforce server. Along with the total number of commands executed on the server (per day), the number of Perforce commands that can be directly tied to the tool set (called the 'ResourceBrowser') and the number of specific 'attribute' commands are also shown. As seen, the server is quite active and supports a large number of calls that are in direct support of the tools.



Currently, there are 1.6 million attributes across 320,000 files for our current project.

## Conclusion

The decision to use Perforce attributes to store metadata for Volition's game assets was the right decision. Doing so has greatly simplified many aspects of managing and supporting asset attributes. Despite the undocumented nature of attributes, new enhancements are seen from Perforce on a regular basis, helping provide performance boosts as well as functionality changes. Speed of both the 'attribute' and 'fstat –Oa' command have not been an issue.