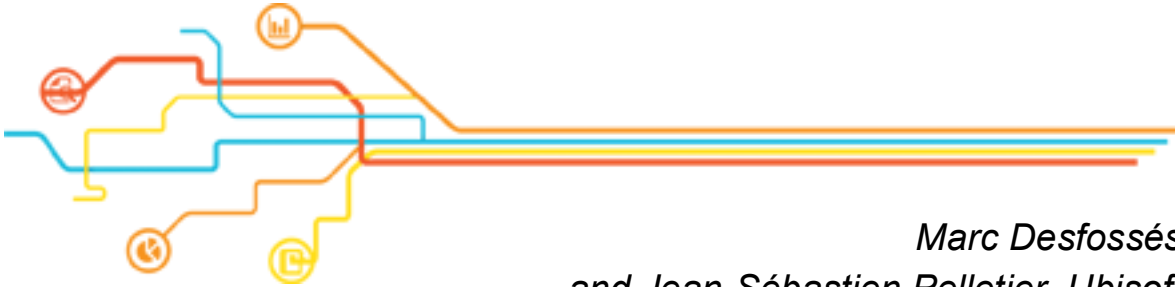


# Perforce Integration in an AAA Game Engine

## The Strength of the C++ API



*Marc Desfossés  
and Jean-Sébastien Pelletier, Ubisoft*

---

### Abstract

In 2004, Ubisoft started building the foundation of a new game engine for *Assassin's Creed*. It built a tight integration between the game engine and Perforce that could store massive amounts of game assets and be scalable and robust enough to support hundreds of people working on the same project in different locations. This white paper discusses the challenges faced during this development and the benefits of having a tight integration with Perforce.

---

### Introduction

Back in 2004, Ubisoft Montreal started building the foundations of Anvil<sup>1</sup>, the game engine behind the Assassin's Creed games. Since then, the Anvil game engine has been used for the production of more than a dozen games, and it is without a doubt one of the most important game engines developed by Ubisoft.

From our experience during the production of previous games, we knew we had to find a better way to store the massive amounts of game assets for the upcoming generation of consoles. The solution had to be scalable and robust enough to support the hundreds of people working on the project with teams spread all over the world. At that time, Ubisoft Montreal was using Perforce Software Version Management to store its source code, and it was a logical choice to also use it for the game assets. To achieve our goals, we came up with a very specific architecture, leveraging the C++ API to fully integrate Perforce into the engine.

In this paper, we will talk about the challenges faced during the development of our in-house Perforce integration library called Guildlib. We will also discuss the architecture, the technical choices that were made, and the benefits we get by having such a tight integration.

### History Must Not Repeat Itself

Part of the engine team behind *Assassin's Creed* previously worked on the game *Prince of Persia: The Sands of Time*. During the production of *Prince of Persia*, we had serious problems with our source control integration. We had data corruption on a weekly basis and people were literally waiting their turn to submit data assets. The system was unreliable and endangered the delivery of the game. Because committing local changes could take up to 30 minutes, our source control was clearly a bottleneck for the production. Precious hours were spent waiting and sometimes data was lost and had to be redone. It was a nightmare for build engineers, and artists were afraid they might lose their work every time they had to submit.

Obviously, we did not want history to repeat itself. One of the first things we decided when *Assassin's Creed* kicked off was that we would invest serious development time in improving the way our data assets were managed. Our first thought was to refactor the existing solution and fix the main corruption issues. It soon became obvious that no matter how much effort we would invest in a custom solution, we could not compete with the stability, reliability, and the performance Perforce provides. We decided to use Perforce, not only to store our game assets but also to fully integrate it into our engine. This approach allowed us to keep the engine's custom file system and have full control over the functionalities exposed to the user. It turned out to be a very good decision because the C++ API allowed us to achieve our vision and meet our performance expectations.

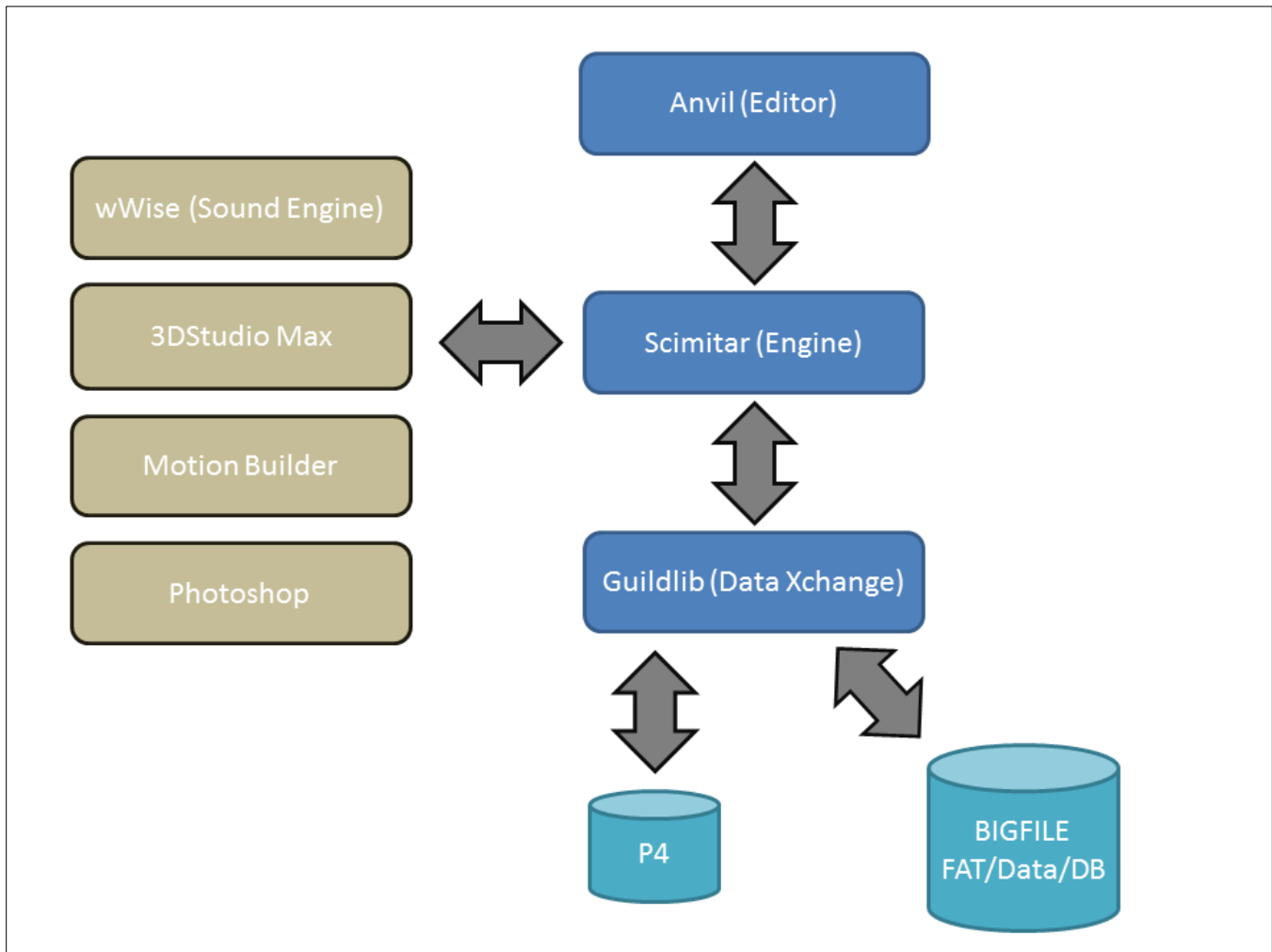
---

<sup>1</sup> [http://en.wikipedia.org/wiki/Anvil\\_\(game\\_engine\)](http://en.wikipedia.org/wiki/Anvil_(game_engine))

## Architecture Overview

The Anvil engine has a multi-tier architecture. Anvil (the engine's official public name) is the presentation layer. Scimitar is the engine itself and Guildlib is the data exchange layer. Guildlib's main purpose is to create the bridge among the core engine, the file system, and Perforce, and it is the subject of this white paper.

Figure 1 presents a simplified diagram of the engine's architecture.



**Figure 1: Overview of the Anvil engine's architecture**

## TheBigfile: Data Storage in the Engine

In Anvil, all files are stored locally in a virtual file system called a Bigfile. The Bigfile consists of a single file container for millions of data files. The Bigfile also embeds a database to store the information about game objects. There are several advantages in having a Bigfile on a game production:

1. Speed: Because all data is in a single file, we avoid opening/closing thousands of files when performing operations on the files (edit, load, sync).
2. We can copy a Bigfile from our network instead of syncing the files directly from Perforce. This can be an order of magnitude faster than syncing millions of files from Perforce. (Some sync comparisons appear later in this paper.)

## 4 | Perforce Integration in a AAA Game Engine

3. We have more control over the types of operations that can be performed on the files because all file modifications have to be made using our custom tools. This also means that operations on the Perforce client view must also go through our tools. This lets us implement features such as ghosted files that are synced only at their first read attempt. (We discuss custom file statuses later.)
4. Deleting a single Bigfile is much faster than deleting 2.5 million files from the HDD.
5. Users cannot clobber their environment, and this saves us trouble debugging crashes caused by bad user manipulations.

The Bigfile file system also contains a hierarchy of folders and files like the file systems we are accustomed to. Each of those files and folders is identified using a unique 64-bit object identifier that will NEVER change during the object's lifetime.

### The Perforce Depot Hierarchy

For performance reasons and to be able to simplify some Perforce operations such as rename and move (more on this later), we came up with a special mapping between the files in the Bigfile and the files in the Perforce depot.

Our files are stored in the selected depot branch using the following path:

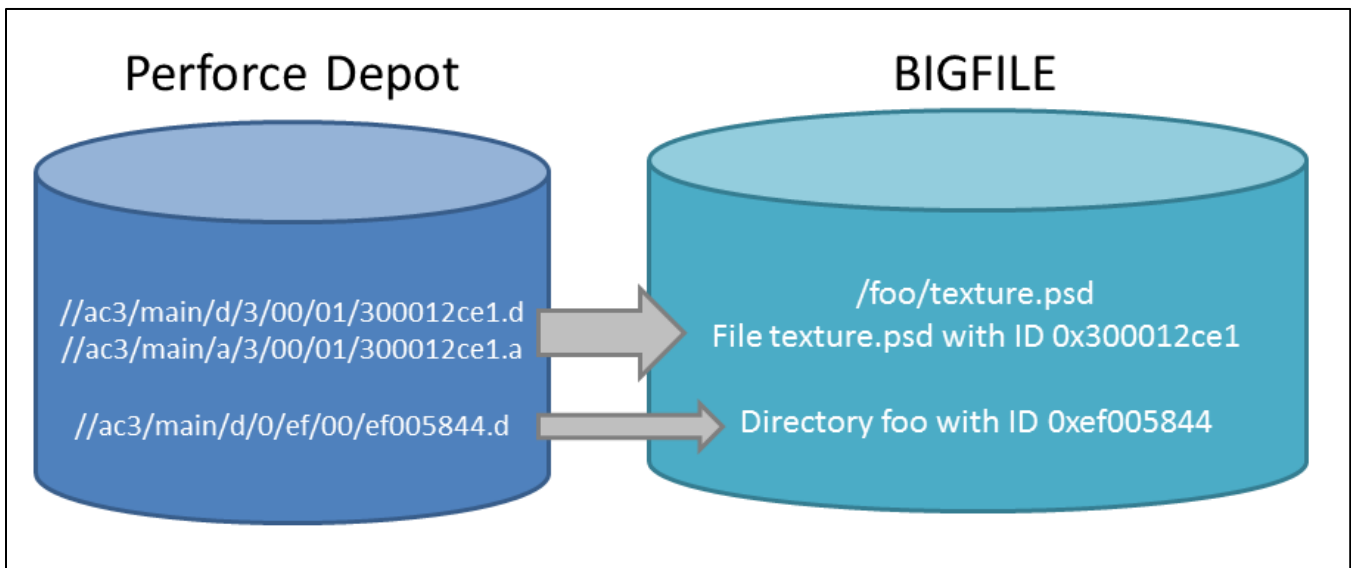
```
//<depot>/<branch>/<a|d|f>/<bytes 5-8>/<byte 4>/<byte 3>/<complete object id>.<a|d|f>
```

Where a = attribute file, d = data file, f = folder

Some examples:

```
//ac3/main/d/3/00/01/300012ce1.d - 33 chars - ID 0x300012ce1
//ac3/main/d/0/ef/00/ef005844.d - 32 chars - ID 0xef005844
//ac3/main/a/3/00/01/300012ce1.a - 33 chars - ID 300012ce1
//ac3/main/a/0/ef/00/ef005844.a - 32 chars - ID 0xef005844
//ac3/main/f/0/00/04/43d40.f - 29 chars - ID 0x43d40
//ac3/main/f/1/00/01/10001409d.f - 33 chars - ID 0x10001409d
```

Figure 2 shows how the mapping between the Bigfile and Perforce is done. The file texture.psd consists of a single file in the Bigfile. In the Perforce depot, it corresponds to a pair of an attribute and a data file. The data file contains the raw data and can be a very large file (hundreds of megabytes for Photoshop files). The attribute file is used to store some metadata information about the file such as its filename, the parent folder, and its thumbnail. It is meant to be a small file, no bigger than a few kilobytes.



**Figure 2: Mapping data from the Perforce shared repository to Bigfile**

In our Perforce integration, the folders are under revision control. For each folder, we store all its associated metadata in a file called the directory attribute file. This file contains pairs of key/values for storing directory information such as the owner directory and the directory name associated with the folder in our Bigfile.

Having some metadata associated with each file in a different file (attribute file) allows us to move files, rename files, update the thumbnails, or add new metadata without having to resubmit the data file. This is particularly useful for large files such as textures or 3D models. Moving a file is only a matter of editing its Perforce attribute file, modifying the owner directory, updating our Bigfile FAT content, and then submitting the attribute file.

This also means that moving a data file/folder around in the Bigfile doesn't move the file in Perforce and is instead a simple edit operation in our tool. This feature is a huge benefit for us because branching can be confusing for users. The workflow is totally different from what happens when you rename or move a source code file with P4Win or P4V.

In addition, we have the following file `//<depot name>/<branch name>/config/datacontrol.ini` used to store the data control configuration that can be dynamically modified at any time. This file is constantly watched and can be used to force users to upgrade their version or have someone as their changelist approver for a submit. This is useful when we are close to the end of certain milestones.

### Guildlib Attributes Versus Perforce Attributes

Someone might wonder why we haven't used the Perforce attributes feature to store our metadata instead of our own attributes files. First, when we started this project in 2004, this feature was just becoming available in a very limited form for thumbnails and was an undocumented feature. Second, some functionality was missing and is still missing in today's implementation of the p4 attribute that we would need in order to keep our existing logic.

In 2011, we investigated the possibility to migrate to the Perforce native attributes for two main reasons:

- To reduce the amount of files we store in the database

- To have a clean fix for the atomicity problem in our engine. Both the data and attribute file should be in sync all of the time, which complicates the code.

To be able to re-implement our metadata storage using the Perforce attributes, we would need the following additions:

- Be able to sync attributes along with files
- Be able to submit changes to attributes without having to submit the associated file
- Be able to diff attributes

### Data Synchronization

Because our Perforce depot tree does not match the Bigfile hierarchy as explained earlier, our synchronization process had to be adjusted to deal with the concept of attribute files and directory files. (Our directories are under revision control.) The general idea of the data synchronization process is to reconstruct the folder tree on disk and then write the pairs of data/attribute simultaneously in our Bigfile.

Data synchronization involves several steps that we'll summarize here:

1. We first execute a sync preview and keep what will be synched in memory data structures.
2. Synchronization of directory attributes: The attribute files are not “stored” in the Bigfile; instead we read and interpret all the directory attributes and write them on disk. The folder hierarchy is therefore reconstructed in the Bigfile.
3. Synchronization of file attributes (also buffered in memory): They are interpreted later, either when receiving the associated data file or at the end of the sync if only the attribute file was received (for instance, if the file was moved in a different Bigfile folder).
4. Perform file ghosting/unghosting (more on this later) using p4 flush commands on selected data files. These flushes affect the next step.
5. Synchronization of data files. When a file is completely received (we buffer it completely in memory for speed), we update the Bigfile. When updating a file in the Bigfile, we check if we've received an attribute file and we interpret it at that time and apply all the attributes when we update the data file (that update is ACID<sup>2</sup>).

### Client View Synchronization

In our Perforce integration, there is a central concept: the Bigfile is the master. Because we are able to copy a Bigfile from the network or to a colleague, we need to be able to synchronize the content of that Bigfile with its associated Perforce client on our server. This is what we call the client view synchronization. In order to do that, we store source control metadata in the Bigfile for each file contained in the Bigfile. This metadata includes information such as:

- Revision numbers for data, attribute, and directory files

---

<sup>2</sup> <http://en.wikipedia.org/wiki/ACID>

- Source control actions for the data and attribute files: Edit, Delete, Add, and Custom statuses (more on these later)

We also keep some global metadata information in the Bigfile metadata such as the last synchronized changelist, the server address, server depot branch associated with the Bigfile, and some guides to know if we need to synchronize the Bigfile with its associated Perforce client.

When we open a Bigfile, we first compute a unique client view name from several criteria such as the Bigfile path, depot branch, and machine's name. We then examine the client view and the Bigfile to know if we need to synchronize the Bigfile with the client view. One of the things we examine is a guid stored in the windows registry as well as in the Bigfile. This guid is used to indicate if a crash has occurred during a Perforce operation (e.g., a revert operation). Depending on the value of the guid, we will decide if the existing client view must be resynchronized or not with the Bigfile.

We then run several commands such as `fstat`, `revert -k`, `flush`, `delete`, `edit`, and `add` to replicate on the client view the exact state of each file as stored in the Bigfile. This synchronization process is incremental to minimize the execution time and will use the last synchronized change list number as a helper to accelerate the process if possible.

## Using the C++ API

Central to our Perforce integration, there is a very important C++ class that we've called `PerforceBigFileWrapper`. This class derives from the `FileSys`<sup>3</sup> class from the Perforce C++ API. We use this class in the Perforce API to implement file system access. Normally, users of the Perforce API don't have to overload this class to use the API, but in our case this class is a key to our integration with the game engine.

In this class we've intercepted all the virtual functions to redirect all I/O to our Bigfile. For instance, no directories or files are created in the file system when synching files because we've intercepted those calls.

We've found that the Perforce C++ API is quite easy to use and very powerful. Because it is callback based using virtual methods, this gives us a lot of flexibility about how to process a command's output via custom handling of error messages, text output, or the dictionaries of key-values.

The fact that the C++ API is callback-based is especially good for our solution because we have millions of files in the source control; if each command buffered its results in containers, it could amount to GBs of memory being consumed by dictionaries or text output. We have more control this way and can decide when and what we need to buffer in memory after a command runs.

We've seen several advantages with this type of file system integration:

- By being able to intercept the files access, we were able to implement our attributes files, which are not stored directly in the Bigfile but are interpreted at sync time and dynamically reconstructed when submitting or diffing files. This would have been impossible without full control on the file system operations.

---

<sup>3</sup> [http://www.perforce.com/perforce/doc.current/manuals/p4api/02\\_clientprog.html#1050939](http://www.perforce.com/perforce/doc.current/manuals/p4api/02_clientprog.html#1050939)



- When syncing, we dynamically serialize (using a generic serializer) the content of the data files we're receiving and then index the data into a database used to know the relations between game objects. This is all done asynchronously using the raw synchronization buffers from the data synchronization. We also update the content of the big file by using that buffer. Without that deep integration, this would have been less efficient to index our files because we would need to read back the synced files from the disk to be able to analyze them.

That deep integration has some disadvantages:

- Some of the behaviors of the virtual functions available in the FileSys class are not very well documented, and the interaction between the virtual functions and the Perforce commands are not always clear. For example, some Perforce commands will call the Stat() virtual method to verify if a file exists or not. But in our case because this is not the Windows file system, we must sometimes return a fixed return value depending on what kind of command is executed. For example, when reverting files, Stat() is called to verify if a file exists; we've found that in some cases it can be faster to always return 0 to avoid the creation of a temporary file. Therefore to help make those choices, we have access in our FileSys object to what kind of operation is being run (sync, revert, revert unchanged, etc.). This information is transmitted to the FileSys object at creation by the UI object invoking the current Perforce command.
- We need to keep up with new Perforce features. For example, we've done most of the work to integrate the shelving in our workflow, but some work is still needed before being able to push the feature in production. However, this is not critical because we have an import/export feature similar to p4tar.<sup>4</sup>
- When upgrading the server, we must ensure that everything is behaving as it should. We highly suggest that you lock your protocol version to a specific version.

Implementing a complete virtual file system similar to what we've implemented is several thousand lines of C++ code. It took a few months for two programmers to have the basics in place and then this has been an on-going work in progress for the past eight years. In the last eight years, we estimate that at least four to five years/person has been invested on this project. It is a lot of work, but we think it was worth it. It certainly is a major contribution to our engine's success.

### Challenges

When designing Guildlib, we wanted to make sure it could meet our high expectations in terms of scalability, user experience, and stability. We wanted a solution that could evolve over the years with the engine and support the constantly growing needs of the gaming industry.

### Scalability

In the early 2000s, the production teams at Ubisoft were relatively small. A group of 80 talented programmers and artists working together in the same studio for two to three years was a good recipe to create AAA games such as *Splinter Cell* or *Prince of Persia: The Sands*

---

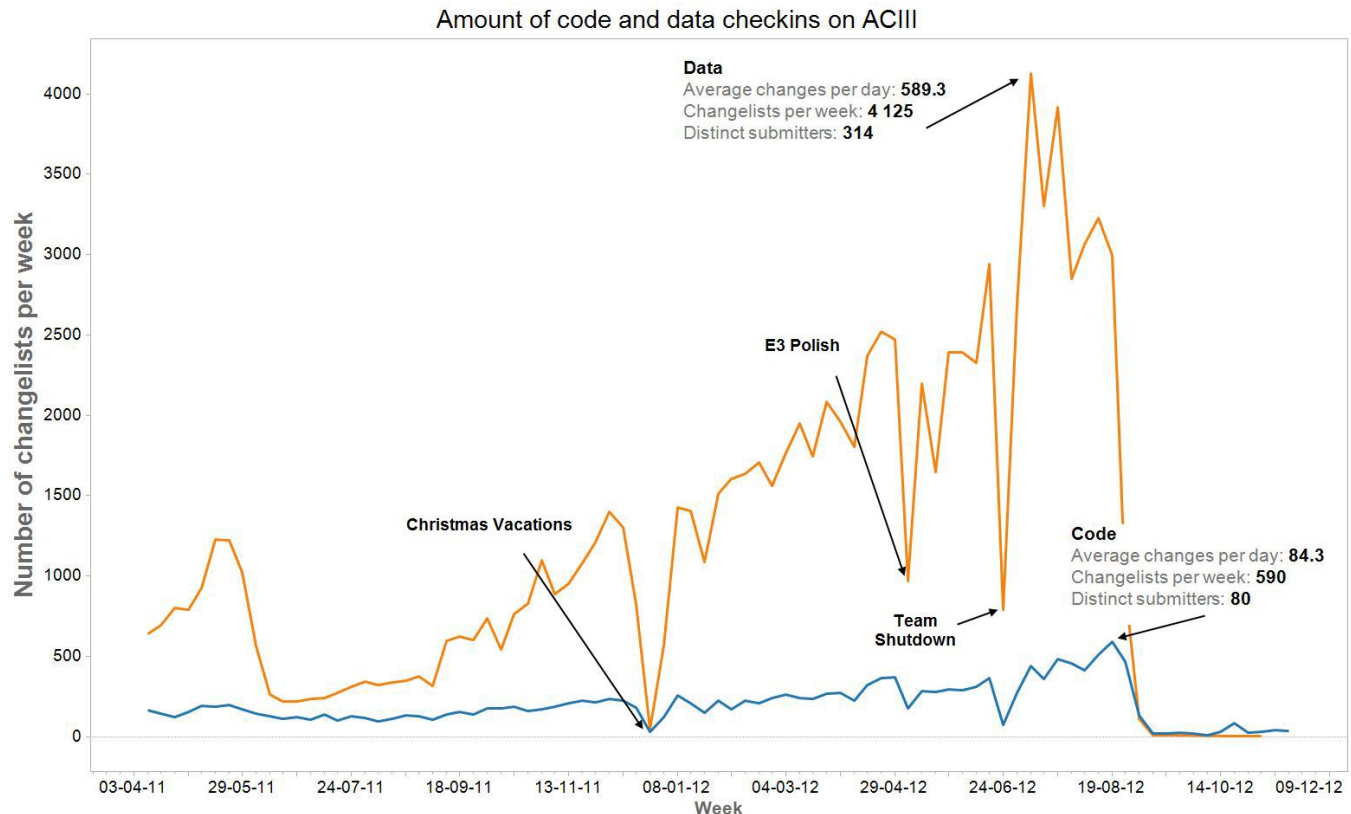
<sup>4</sup> <http://public.perforce.com/wiki/P4tar>



*of Time*. At the time, we thought that these were big teams and we were reaching the limits of our source control solutions. The arrival of the next console generation (Xbox 360 and Sony PS3) combined with the open world trend had a direct impact on the teams and data sizes. The choice to switch to Perforce was to address those growing needs. We were confident that Perforce could support:

- Hundreds of GB of data
- Millions of files
- Hundreds of users

Figure 3 shows the amount of code and data submits in the last year on ACIII.



**Figure 3: Code and data submits in the last year on ACIII**

## Stability

It was important for us to keep the stability Perforce already provides. The entire goal of this integration was to make sure we did not end up with corrupted data like our previous SCC integrations. This was a challenge, however, because our integration modifies the way we perform files operations on the disk. In short, re-implementing core functionalities with the Perforce API such as I/O operations (described in the “Using the C++ API” section earlier) meant that we were risking losing the existing stability of the Perforce clients.

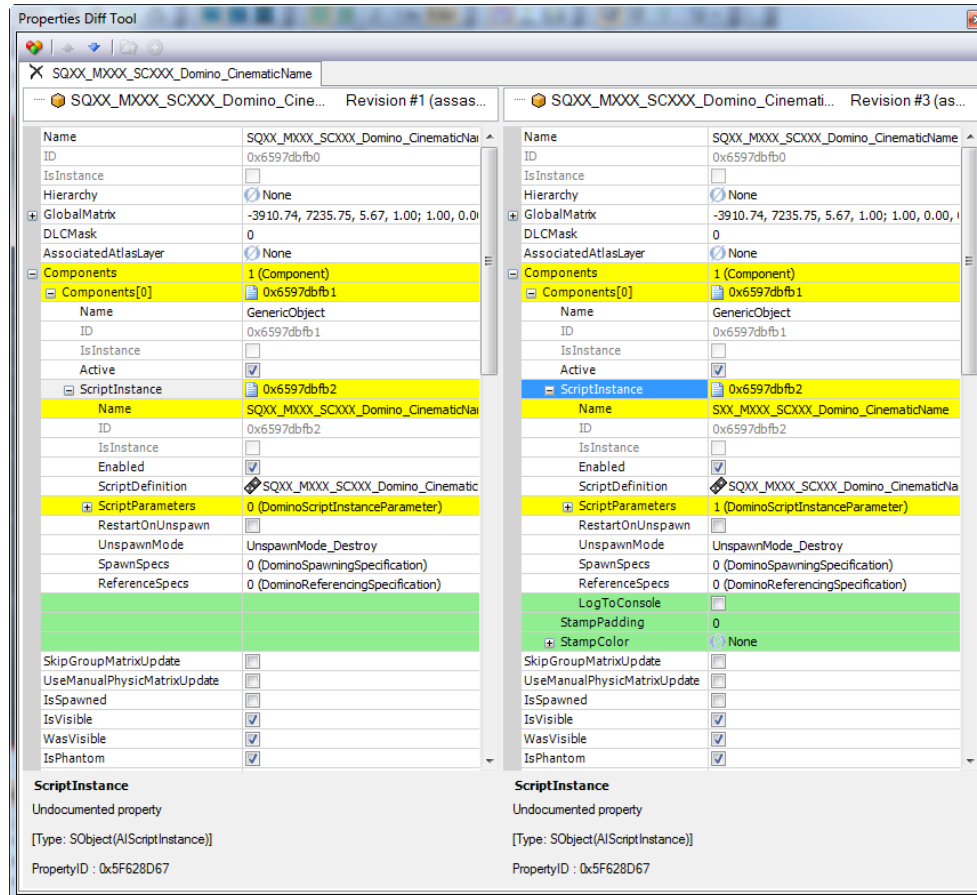
## User Experience

Our first experience with Perforce came with the P4Win client. Programmers really liked it at Ubisoft. P4Win was very basic and was not particularly user friendly for non-programmers; artists were reluctant to use it. We chose to rewrite the main parts of the user interface in order to expose only the Perforce commands and statuses that were relevant for our engine. The basic Perforce concepts such as changelists and opened files were kept, but some more

advanced features such as branching are hidden to the common users.

## Visual Diff of Binary Files

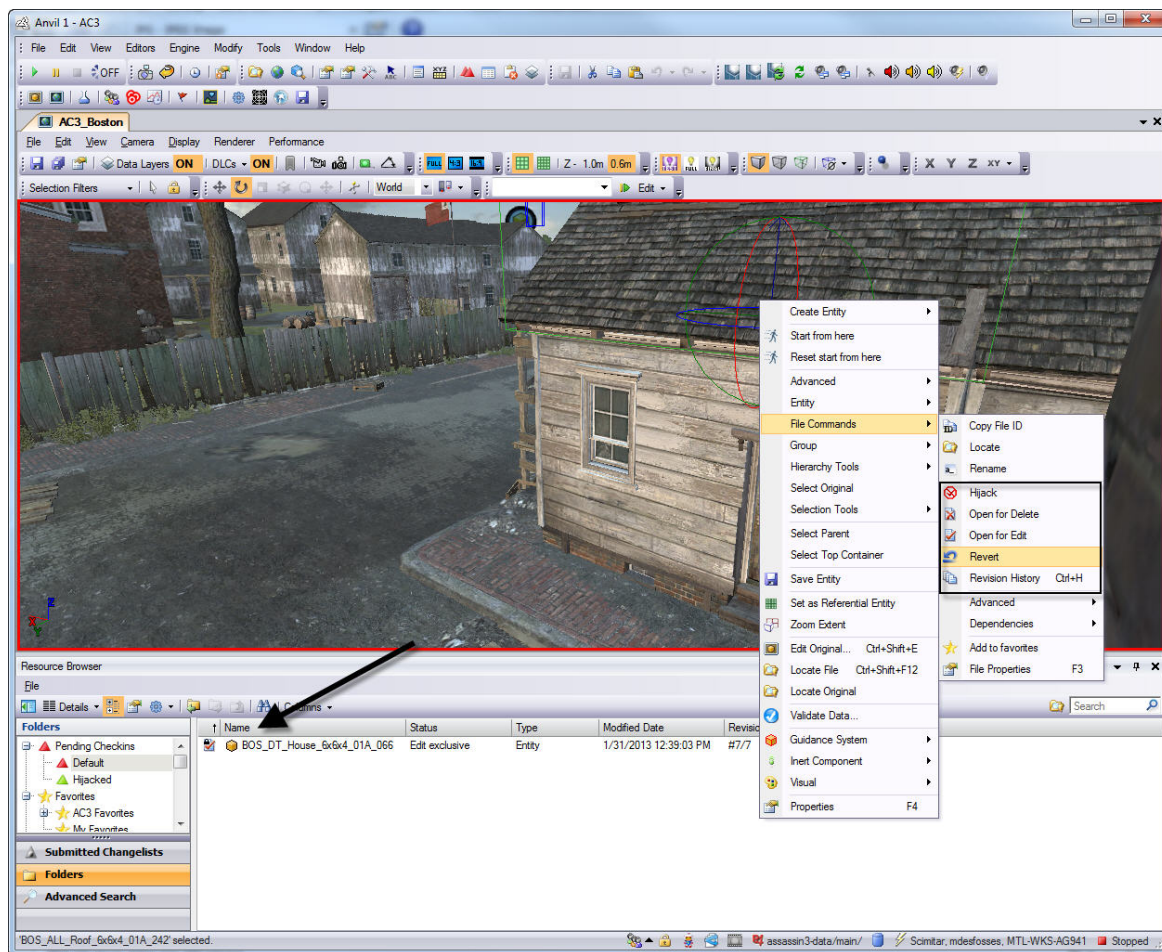
One of the nice features of our integration is the Visual Diff window (see Figure 4). It is basically a diff tool similar to P4Merge but with the ability to diff the game objects. This is especially useful for data validation prior to submission.



**Figure 4: Diffing game objects**

## Automatic Open for Edit

Having full control over the files allows us to automatically open for edit files and deal with clobbered files more easily. We can also see the file commands available for each file (see Figure 5).










**Figure 5: The integration allows for full control over file actions**

## Dependencies Viewer

The dependency viewer lets us see dependencies between all files (see Figure 6). We can also see our source control file status displayed in the window.

CHR\_CIN\_Feathers

Auto-Refresh | History | Columns

| Name  | Status       | Type        | Modified Date       | ID          |   | Via Properties                |
|---|--------------|-------------|---------------------|-------------|---|-------------------------------|
| Original  |              |             |                     |             |   |                               |
|  CHR_CIN_Feathers      | Edit excl... | Material    | 6/5/2012 11:37:...  | 0x233466fe9 | Original  |                               |
| Reference   |              |             |                     |             |   |                               |
|  CHR_CIN_Feathers_Set  | Hijacked     | Texture Set | 6/5/2012 11:37:...  | 0x233466fca |  Reference | Material.TextureSet (1)       |
|  AC3_Tex1_Unit         | Opened f...  | Shader      | 6/18/2012 10:30:... | 0x30132eee  |  Reference | Material.MaterialTemplate (1) |
| Referer   |              |             |                     |             |   |                               |
|  CHR_CIN_Feathers_LOD0 |              | Mesh        | 6/5/2012 11:37:...  | 0x233466feb |  Referer   | SubMesh.Material (2)          |

**Figure 6: The dependency viewer shows file dependencies and file status**

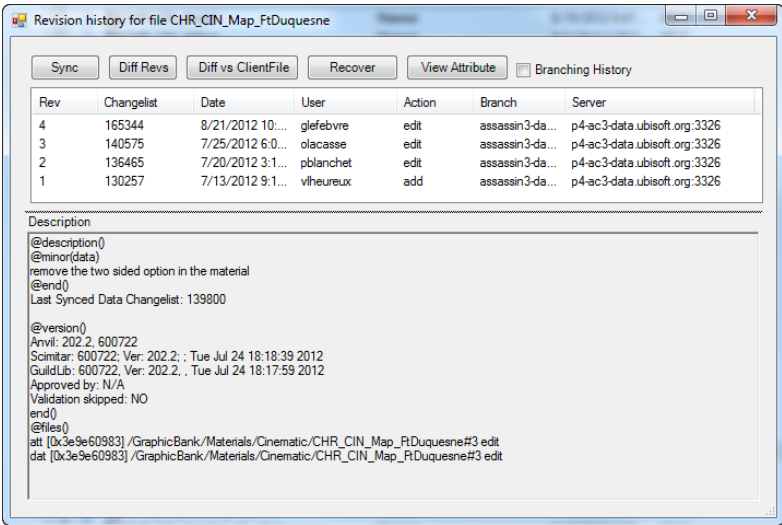


Figure 7: Custom revision history control

Submit Assistant

The submit assistant is our implementation of a submit dialog (see Figure 8). The assistant also integrates with JIRA and does many custom client-side data validation to ensure data consistency prior to submission.

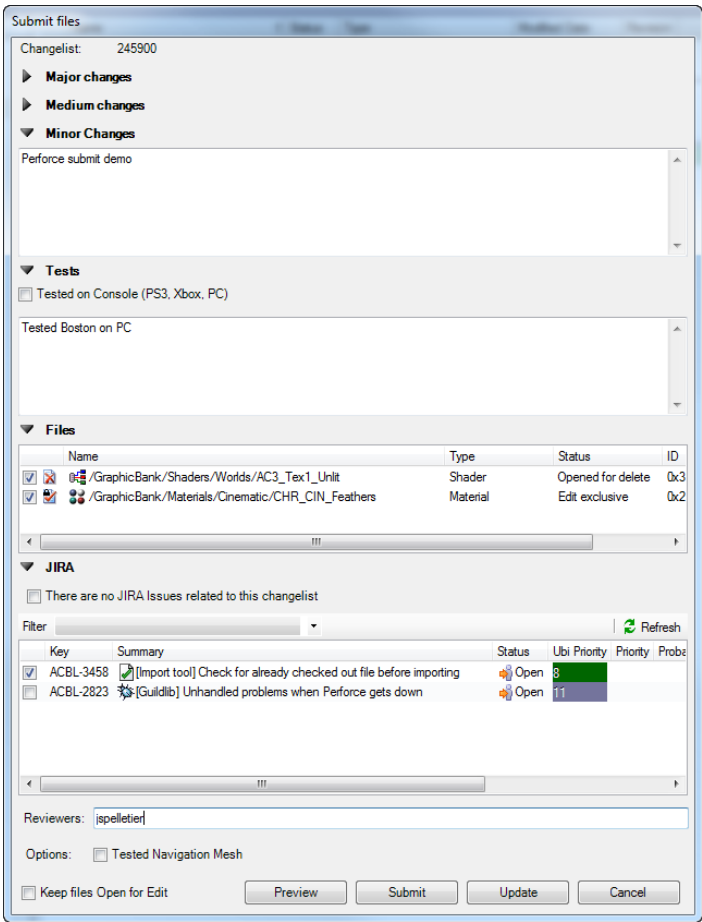


Figure 8: JIRA integration

Performance

During the development of Guildlib, we always kept performance in mind. It was important for us that the end user did not suffer from the millions of files and Gigabytes of data required for building the game.

Some operations are costly, especially when working from a client in a remote location. For instance, the client synchronization required when a user starts from a fresh Bigfile or after a crash requires running a p4 flush command. This command is very inefficient because it calls the API after each file. To work around this problem, we came up with our own command proxy that executes the command on the site where the server sits and sends output to the remote client. This is much more efficient because the output is sent only once in a compressed format without any latency occurring for each of the files to process. However, recent testing showed that on a very recent Perforce server, we could run a p4 flush -q and it won't call the callback if the client is not running with tagged output.

Benefits

Faster Sync Operations

It was not planned or expected but synchronizing millions of attribute files in memory and saving them in batch turned out to be very efficient performance-wise. Our sync operations are faster in Guildlib than they are when we use P4V or any other Perforce client that directly writes the files on the disk (see Figure 9).

|                          |  |
|--------------------------|--|
| Client                   | HP Z420 with 12 threads (6 cores), 32 GB RAM, 2 TB HD and Intel 520 (480 GB) |
| Total Data Size (MB)     | 15 147   |
| Number of Files (Sample) | 328 093  |

| HDD | Sync Type | Time(Best) | Time(secs) | Speed(MB/sec) |
|-----|-----------|------------|------------|---------------|
|     | FakeP4    | 00:03:04   | 184        | 82.32         |
|     | BigFile   | 00:03:28   | 208        | 72.82         |
|     | P4        | 00:13:38   | 818        | 18.52         |
|     | P4V       | 00:08:12   | 492        | 30.79         |
|     | P4Win     | 00:18:27   | 1107       | 13.68         |

| SSD | Sync Type | Time(Best) | Time(secs) | Speed(MB/sec) |
|-----|-----------|------------|------------|---------------|
|     | FakeP4    | 00:03:04   | 184        | 82.32         |
|     | BigFile   | 00:03:24   | 204        | 74.25         |
|     | P4        | 00:09:59   | 599        | 25.29         |
|     | P4V       | 00:06:07   | 367        | 41.27         |
|     | P4Win     | 00:13:42   | 822        | 18.43         |

Figure 9: Sync operations are faster in Guildlib than P4, P4V, or P4Win

The FakeP4 line is for a custom replacement of p4.exe that we've done to replace all file operations by no-ops. This means that the times taken by this executable are the best



theoretical ones and are only limited by server and network speed.

As you can see, the sync times with our Bigfile implementation are excellent and we are very close to the theoretical speed. Our implementation is also not really affected by the type of drive because we've got almost the same time with an SSD and HDD. Also, you can see that all official clients are taking much more time than our client implementation with both disk types but with SSD having much better sync times.

### Custom File Statuses

#### Ghosted Files: Sync-on-Demand of Large Assets

The number of files we have in each depot branch is enormous, with millions of files amounting to more than 100 GB. Syncing all of those files takes time and takes a lot of disk space. Also, some types of files are useless for 99 percent of our users.

For example, a texture is updated using Photoshop by opening a .psd file. Those files are wrapped in our Bigfile; when a user wants to edit a texture, the texture is extracted from the Bigfile, stored on the disk, and edited in Photoshop. When the user is finished editing the texture, it is reimported in the Bigfile and converted to formats usable by the engine. The engine never directly loads .psd files.

Those files are really big; some of the .psd are over 500 MB with many of them exceeding 100 MB. To avoid syncing those files, we had to come up with a creative solution to save on disk space and avoid syncing them.

Those files are stored in our Bigfile but are only useful for artists editing those textures. So we sync those files on demand from Perforce when the user tries to access the file. This also means that when syncing the associated data file for those files, we need a way to skip them.

Because we synchronize our file attributes first, we are able to determine which files correspond to the types of files we would like to ghost. Once we've identified those files, we perform a p4 flush //filepath/...@changelist on all the data files we want to exclude from the sync operation.

Then when we sync the data files, ghost files are skipped since we ran a p4 flush on them with the revision found in the sync preview. Also in the Bigfile we store that revision number in the associated FAT file entry for the ghost file.

### Results

On AC3, if someone would try to sync all the data including ghost files, then he or she would have to sync 1,018,231 data files for a total size of 147.15 GB. But with our implementation of ghost files, a sync from scratch results in a Bigfile of 19.4 GB, which is a huge saving in size and sync time.

#### Hijacked Files: Our Own Implementation of Clobbered Files

When we started the deployment of our Perforce integration, we were supporting multiple checkouts on attributes files (along with auto-merge of attributes) as well as exclusive locking for data files (that exclusive locking is enforced in our tool set). It didn't take long before we got some reports from users who needed to be able to locally modify files to make tests even if someone else locked the files.

What we've come up with is a mode that we've called hijack. A hijacked file acts as a kind of source control operation but it is local to the Bigfile and completely independent of Perforce. This lets users modify/delete files locally and mark them as such in their Bigfile. Then they are able at any moment to revert their local change or put the change in a change list (provided nobody else has the file checked out or has submitted a new version of the file).

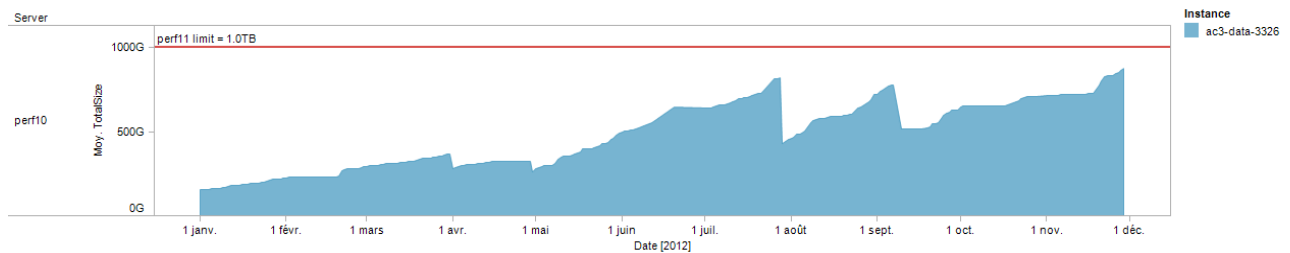
When users sync from Perforce, they generally revert all their hijacked files before proceeding with the sync.

## Perforce Metadata and Large Client Specs

### Database Optimisations

A few months ago the instance holding the data for the *Assassin's Creed* brand had a total .db size exceeding 700 GB (with about 92 percent in db.have). This prompted us to think about how we could reduce the total size of our database files.

Figure 10 shows the Perforce databases size in 2012.



**Figure 10: Perforce databases in 2012 exceeded 700 GB**

After some thinking and simulations, we ended up changing a lot of things to reduce the length of the file paths stored in the databases.

Until a few months ago, our files were stored as the following form:

```
//assassin3-data/main/data/hi/3/00/01/2c/0000000300012ce1.scc_dat - 66 chars
//assassin3-data/main/data/ef/00/58/ef005844.scc_dat - 53
//assassin3-data/main/attr/hi/3/00/01/2c/0000000300012ce1.scc_att - 66
//assassin3-data/main/attr/ef/00/58/ef005847.scc_att - 53
//assassin3-data/main/dir/00/04/3d/00043d40.scc_dir - 52
//assassin3-data/main/dir/hi/1/00/01/40/000000010001409d.scc_dir - 65
//assassin3-data/main/data/hi/ffff/aa/01/2c/0000ffffaa012ce1.scc_dat - 70 - WORST CASE
```

We've changed that to this shorter form:

```
//ac3/main/d/3/00/01/300012ce1.d - 33 chars
//ac3/main/d/0/ef/00/ef005844.d - 32 chars
//ac3/main/a/3/00/01/300012ce1.a - 33 chars
//ac3/main/a/0/ef/00/ef005847.a - 32 chars
//ac3/main/f/0/00/04/43d40.f - 29 chars
//ac3/main/f/1/00/01/10001409d.f - 33 chars
//ac3/main/d/ffff/aa/01/ffffaa012ce1.d - 36 chars - WORST CASE
```



**Optimization 1: Shorter Depot Names**

We've renamed the depot `//assassin3-data` to a shorter `//ac3`. This saves 12 characters per file.

**Optimization 2: Renamed Root Directories and Extensions**

We've renamed the root directories from `data`, `attr`, and `dir` to `d`, `a`, `f`. We've also renamed file extensions from `.scc_dat`, `.scc_att`, and `.scc_dir` to `.d`, `.a`, and `.f`.

Savings are nine characters for data and attribute files and eight for folder files.

**Optimization 3: Client Name Optimization**

Also, we've changed the name of our client views from something like `scimitar-dataserver-ac3-mtl-wks-ag650-1` (40 chars) to a `crc32` of several criteria merged together. To save even more characters, the name is then converted using a scheme similar to `base64`. Our client names now range from four to eight characters with an average of six characters.

**Optimization 4: Client View Mapping Now Omitting the Depot Path from Mapping**

We now omit the depot path from the right part of the client view mapping because we only map one branch at a time with those clients and client views are created by our tool.

Before:

No optim (53 characters per file):

```
//ac3/main/... //scimitar-dataserver-ac3-mtl-wks-ag650-1/ac3/main/...
```

Optim 2 (21 characters per file):

```
//ac3/main/... //7cc82a5f/ac3/main/...
```

After optim 2 and 3:

All optims (13 characters per file):

```
//ac3/main/... //c7cc82a5f/...
```

In this specific case, this reduces from 53 to 13 characters the client path for each file referenced by `db.have`. Given that we have more than 2 million files mapped in each client view, this quickly amounts to enormous savings.

**Effect on Database Size**

With all these changes, we've reduced the total size of our `.db.have` by 55 percent (size after optims: 550 GB) with freshly recovered checkpoint.

**Effect on Performances**

We've measured the time to run a flush command over instances without and with our changes. Flush `//...@now` is now 19 percent faster. And flush `//...#0` is now 23 percent faster. These results are similar to those in a paper from Michael Shields.<sup>5</sup>

---

<sup>5</sup> <http://www.perforce.com/user-conferences/2009/repository-structure-considerations-performance>

## Conclusion

The development of Guildlib using Perforce and its C++ API has been an ongoing task for the past eight years. The initial problems we encountered with our previous integration such as data corruption are long gone.

Even though it required a lot of work, our architecture and design choices allowed the game productions to scale to a level we did not even anticipate. The server performance did not suffer much and we never lost precious data.

The decision to store our game assets in Perforce was a good one. Having Perforce embedded in the engine allowed us to meet our scalability, stability, and performance requirements. The Perforce C++ API is well designed and allowed us to realize our vision.