Major Project Documentation

Joe Withers

Contents

1 Introduction

This document serves to detail the work I was responsible for during the Final Major Project assignment, in which I helped create an animated short film as part of a four person team. As an aspiring software developer, my responsibilities for this project were mainly the technical aspects within production, such as pipeline management and character rigging.

2 Pipeline Management

2.1 Requirements

- Reliably store all of the project data in such a way that is accessible to all team members.
- Provide artists with a tool that manages assets, allowing them to update, reference,
- Provide an automated method for caching the entire scene, with the aim of 'packaging' the project to make it suitable for rendering on different machines.

2.2 Limitations

Prior to developing the pipeline and associated tools, it was important to address the limitations imposed by the environment in which we would be working. The most important of these were:

- Lack of unified storage amongst users. Due to the way the university network is set up, it isn't possible to have a single network location for our shared data storage, without sacrificing one members allocated user storage.
- Lack of storage per user. The approximate storage limit per user is 50gb, which would quickly be hit in a complex production environment. It is therefore imperative that we are concious of the data that we keep hold of.
- Lack of storage space on the renderfarm. The approximate storage limit on the renderfarm is 30gb, meaning that all of the data required to render a scene or shot must fit well within this limit, as rendered frames are written to the same location.

2.3 Storage

For file storage we chose to use Resilio Sync, a peer-to-peer file synchronization service to store all of our working files. This ensured that each team member has their own local copy of the entire working directory, which is beneficial when creating backups. We chose Resilio Sync primarily because it is a free service that is compatible with the university computer network, however it does present us with some problems.

Due to being it peer-to-peer service, we often found that directories would fail to synchronize properly if not syncronized frequently with the other peers. This would not be a problem in a cloud hosted service as the directory state of the working directory would be reliably centralized, reducing the possibility of files becoming desynchronized, however these services are typically expensive and it was difficult to predict our exact storage requirements.

We also noticed a strange problem with Resilio Sync, in which the contents of files would be reduced to 0 bytes. Fortunately the data is usually not lost as it is sent to the 'Archive', which functions as a temporary recycle bin, though restoring these files manually each time it happened proved to become quite tedious. I decided to write a simple bash script to check through the working directory to identify any files with a size of 0 bytes, and to check if a matching file was present in the Archive. However, this wasn't particularly effective as often they would be missing from both the main working directory and the Archive, meaning I would have to search through backups to find the file to restore, which at times felt a bit like baby-sitting.

2.4 Asset Pipeline

When establishing the asset pipeline it was important to seperate pipeline steps not just by the type of work being done, but also by which of the team members is assigned to each discipline. Figure 1 describes the flow of the data between pipeline steps and the corresponding team member(s) that they are assigned to.

2.5 Asset Management

With the Asset Pipeline established, I decided to design and develop an asset management system as it was clear that we would be working with a large number of assets. The asset management system would need to provide the following:

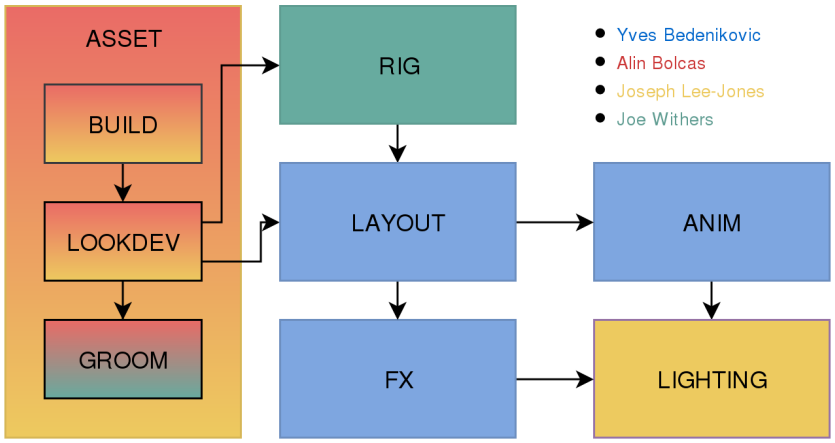

Figure 1: A Flowchart describing the asset pipeline, colours indicating the responsibilities of each group member.

- Provide artists with a simple method for releasing new versions of assets.
- Provide artists with a simple method for gathering assets, so that they can be referenced into a scene.
- Manage versioning of assets, accompanied by information regarding each versions release date, author, and description.
- Automatically update references to assets whenever an asset or one of it's depenencies has been updated.

With this in mind I developed a system that made extensive use of symbolic links (Symlinks). Symlinks create a file which acts like a reference to another file. By creating a master symlink for each asset, updating the version of an asset is as simple as changing the target file. Files such as Maya scene files which contain references to existing file paths can have these paths substituted for symlinks, allowing me to change the version of any file in the asset pipeline non-destructively, with changes being propagated automatically. An example of data flow for multiple assets consisting of symlinks can be seen in Figure 2.

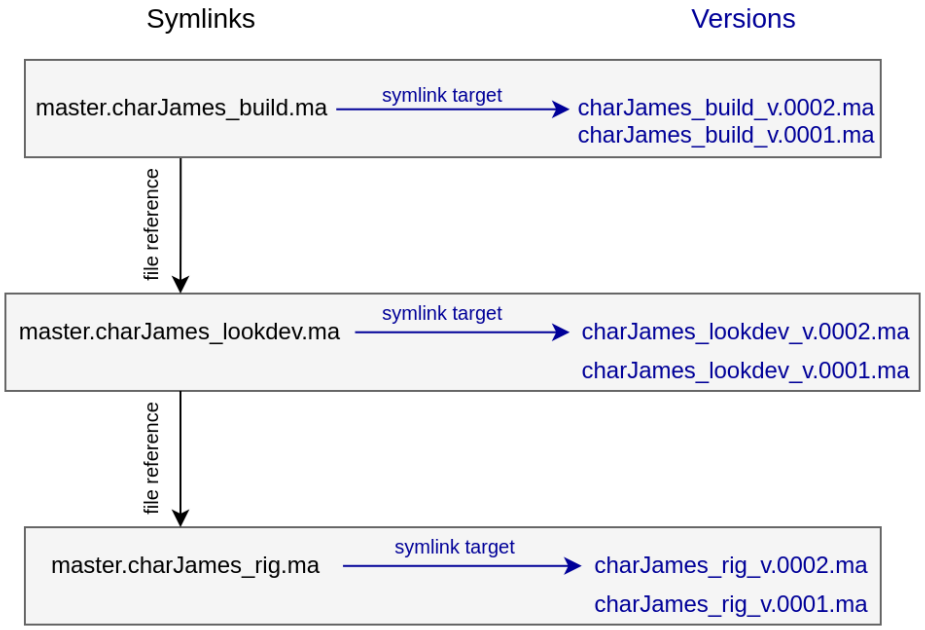

Figure 2: A Flowchart describing the data flow between pipeline steps for the 'charJames' asset. Note how the file versions for each pipeline step can be easily changed by retargeting the symlink.

With this in place the next step was to develop a way of storing metadata for each asset, so that a list of the released versions could be stored alongside important information about them, and that the current asset version could be easily identified. To implement this I created a simple Python dictionary that could be written to a file using the `cpickle` Python library [1]; An example of the data stored within the asset file can be seen in Figure 3.

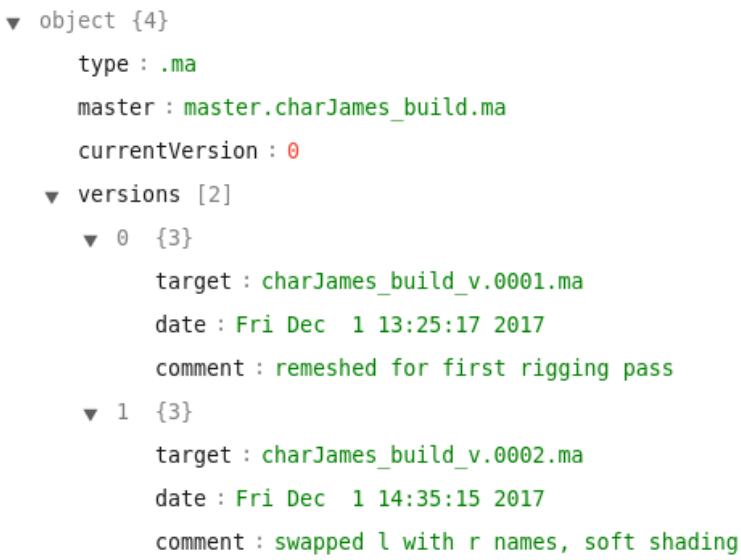

Figure 3: An excerpt from the file `charJames_build.asset`, showing how the data is structured.

The asset file contains the associated file type for the asset, the filename for the asset's symbolic link, and an index for where the current asset version is stored in the `versions` array. The `versions` array consists of Python dictionaries containing the target filename, the date of release, and a comment for each version in the array.

After testing this approach on a selection of non-production assets, I decided that this method of storing assets was fit for purpose, and decided to proceed by implementing a graphical asset manager that the other team members could use to author their own assets. For this I made use of Qt Designer to generate the user interface, primarily because I have experience using it to develop tools, but also because Maya 2017 supports Qt widgets with the inclusion of the PySide Python bindings. After multiple iterations per the requests of other team members, the GUI design shown in Figure 4 was used for both the standalone version of the asset manager, and the Maya

version which extends upon the functionality of the standalone version.

2.6 Production Management

For this project I was assigned the role of the production manager, a role I had previously fulfilled somewhat successfully during the second year group project. Early in the project I had decided to use Shotgun Studio to create an overall schedule for the project, allowing us to create tasks corresponding to each asset, and to assign them to one of the team members with a specified time frame. Figure 5 shows the schedule created during the first term of the project.

Unfortunately, after the first term I found it difficult to convince the other team members to continue using the Shotgun schedule, and we decided to abandon it as a result. Certain group members also found it difficult to adhere to the allocated deadlines for each task, be it a result of the scheduling of other univeristy assignments, or a reluctance to move onto other tasks. Regrettably, this concluded my attempts at production management during the project.

3 Character Rigging

3.1 Requirements

For this project I was responsible for the rigging of both characters, as well as the rigging of any props that the characters interact with. Prior to working on the character rigs, it was important to outline features that would be required to achieve an appealing animation. This ranged from features that would allow the animator (Yves Bedenikovic) to work with them more efficiently, to features that would improve the overall aesthetic of the animation such as cloth and hair simulation. The following features were found to be of highest importance:

- Controls should be familiar to the animator to allow them to work intuitively with the rig. This can be achieved by using previous rigs that the animator has used as reference when setting up controllers.
- Rigs should be capable of achieving the desired facial expressions and poses as dictated by the story.
- Rigs should include the necessary geometry and nodes to allow for cloth simulation to be applied to clothes.

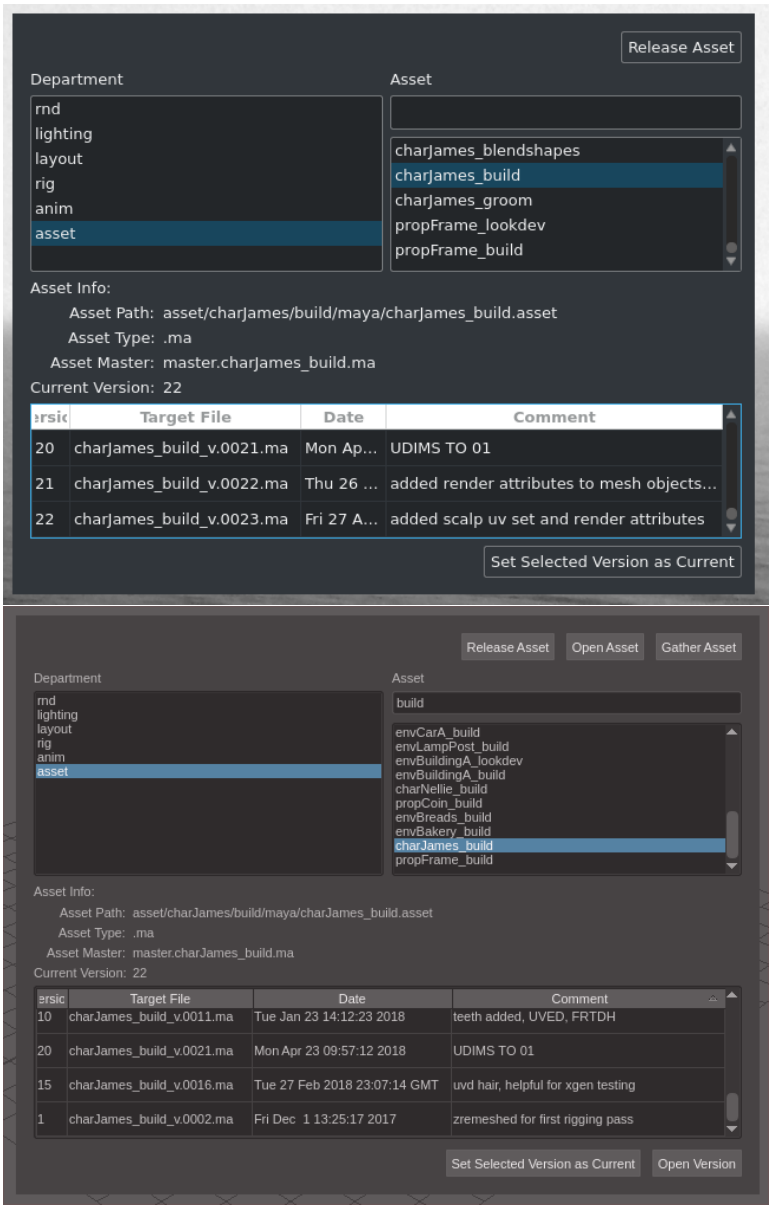

Figure 4: Comparison between the standalone version of the asset manager (Top), and the version of the asset manager designed for use within Autodesk Maya (Bottom). Note that the Maya version provides functionality for opening or 'gathering' assets directly into the current Maya session.

Figure 5: The production schedule created during the first term of the project, grouped by each team member.

3.2 Solution

With these features in mind, I decided to use an automated rigging system to speed up the creation of the character rigs. This allowed me to focus primarily on the features listed above, and let a tool automate the creation of the basic bipedal rig.

I first looked at Kraken [4], a rigging system included within Fabric Engine. This appealed to me as it was easily extensible through it's scripting language, which I thought I would find intuitive given that I had produced a basic automated rigging system for the specialism assignment in second year. Unfortunately, Fabric software went bankrupt at the beginning of the academic year so we were unable to get it working on the university computers.

I then found Advanced Skeleton 5 [5], an extensive rigging tool for Autodesk Maya, which I found to be extremely capable and was more than adequate for my needs.

4 Groom

Xgen stuff

4.1 Pipeline

As shown in Figure 1, Groom assets were not passed through the full asset pipeline, due to Maya not supporting referencing of XGen content. To solve this, the XGen collections and descriptions were exported, and later applied to Alembic caches of the scalp geometries in the cached scene for rendering.

4.2 Results

5 Rendering

5.1 Optimisation

Due to storage limits imposed by the renderfarm, it was necessary to cache the scene geometry to remove the need for large assets to be stored as part of the project. Whilst it would have been possible to use the Alembic file format for this, I opted to use RenderMan's RIB archive format, as it is capable of storing shading networks and render attributes in addition to scene geometry. One downside of this approach is that shaders cannot easily be changed after the cache has been made, however it eliminates the need for

reassignment of shaders, which would be necessary in a comparable Alembic caching workflow.

5.2 Distributed Rendering Tools

Due to inconsistent availability of the renderfarm, our only other immediate option for rendering was to use the computers in the teaching labs. To maximise our rendering output and to minimise disruption to other students, I decided to connect to other computers remotely over SSH. This allows me to connect to far more computers than I would be physically able log into, and at the same time allows other users to log into and use the same computer.

I was able to write a short Python script which used the `ping` command to check which of the computers on the university network were online and running linux, relying on the hostname naming convention using the name of the room in which the computer is located. Upon connecting to a computer, I could then check whether another user is logged into the computer by capturing the output of the `who` command, so that I could render without impacting another user's session. In situations where a computer was found to be logged in by another user, I could check the current CPU and memory usage by capturing the output of the commands in Listing 1.

As linux does not natively support issuing commands in parallel to multiple hosts over SSH, I made use of the `parallel-ssh`[6] Python library to provide this functionality. By following an article from `Linux.com`[2], I was able to use an array of hostnames combined with an array of arguments to render multiple frames using multiple hosts with ease.

Unfortunately I soon found the limit of the university network infrastructure, as having multiple computers reading from and writing to a single directory proved to extremely slow. It was therefore necessary to extend my original command to copy the project directory onto the transfer drive of that computer, execute the render, and finally copy the rendered frames back into the desired directory. Whilst this increased the time required to start a render, the speed of the render itself was much improved.

Whilst this method proved to be extremely powerful, it was certainly not reliable; If a computer is restarted into Windows it cannot be accessed over SSH, meaning that frames were often lost if someone needed to use the computer. As a result we primarily used this technique for test renders and shorter sequences, using the renderfarm to render shots which were known to render without error and for longer sequences.

[2]

6 Compositing

6.1 Workflow

In addition to being responsible for rendering, I took responsibility for compositing the shots, which was convinient as I was managing the storage of the raw renders and would be the first member of the team to get a chance to do any compositing. Being more technically inclined my compositing responsibilities mainly consisted of:

- Reconstructing the 'beauty' pass from the individual AOV passes. The AOVs we ended up using for reconstruction were direct diffuse, direct specular, indirect diffuse, indirect specular, subsurface, and transmissive, with the beauty pass being used for reference. The albedo pass was also used to generate a high pass filter for overall sharpening.

- Rotopainting out any visual errors if time did not permit re-rendering a shot. This primarily consisted of hiding any intersections between the shirt and the jacket, or intersections between the cornea and the eyelid when the eyes were closed.

- Removing Fireflies or any high variance visual noise from the individual passes, either through use of Nuke's denoiser or a third party 'Firefly Killer' gizmo found online [3].

- Setting the view transform as ACES Filmic, and setting up a node to bake the colour space prior to writing.

- Adding depth of field to shots where depth of field wasn't calculated in RenderMan.

As these tasks were common to all shots, I created a template Nuke script which configures everything for me. Unfortunately, when reading multichannel EXRs generated by Renderman, Nuke would swap the alpha channel with the red channel from the direct specular. However this was easily fixed using a pair of copy nodes.

As mentioned above, it was necessary to choose a view transform as RenderMan outputs images in linear colour space. The default option for this would be sRGB, however we opted to use ACES Filmic as it provides a very wide dynamic range, and avoids oversaturation of highlights on skin tones. Figure 7 illustrates the difference between sRGB and Filmic, using one of our renders.

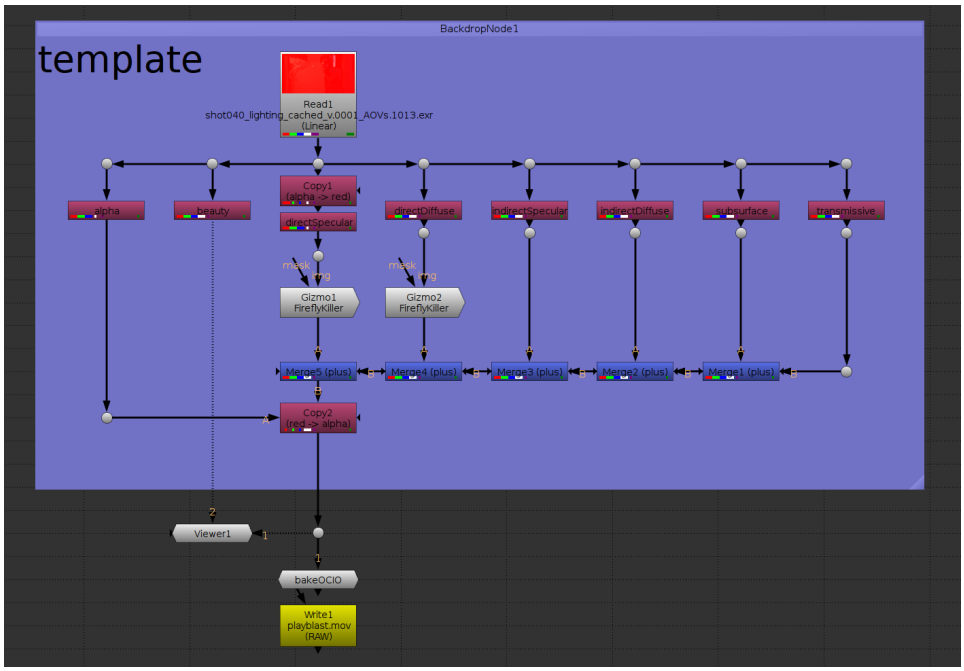

Figure 6: The template Nuke script which was used as the basis for all shots.

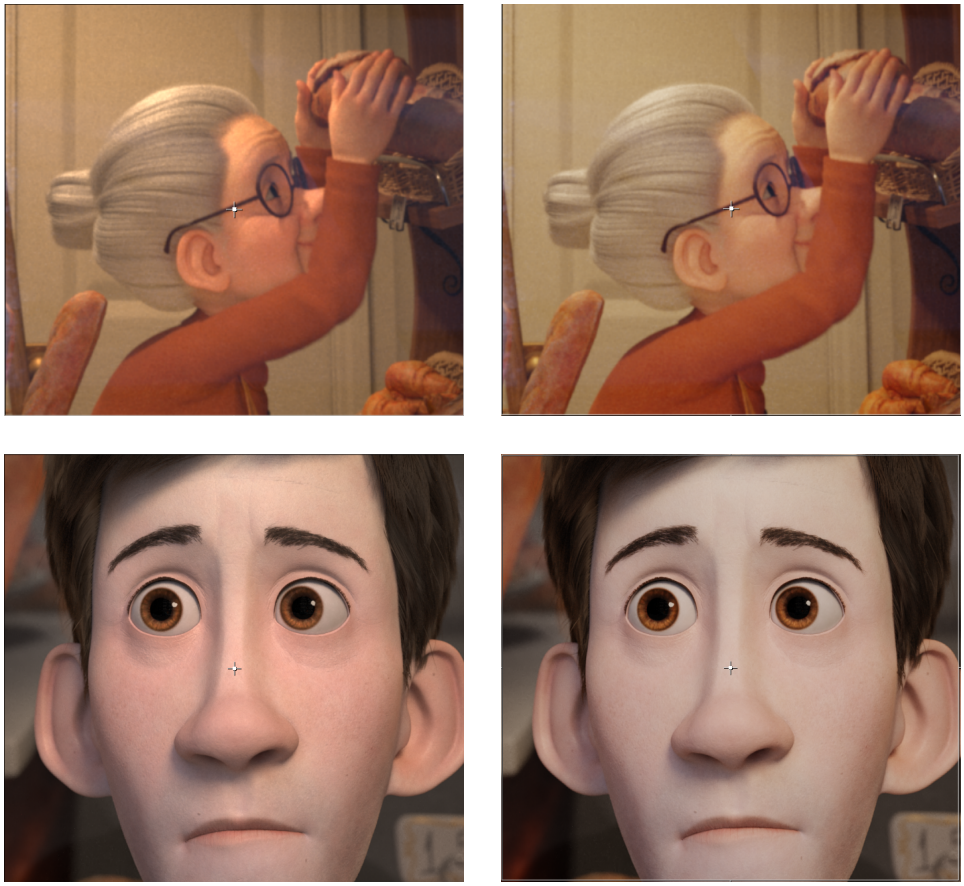

Figure 7: Comparison between sRGB viewing transform (Left) and Filmic 'Medium High Contrast' (Right). Note how skin highlights become oversaturated in the sRGB viewing transform, giving a 'sunburned' look.

For the opening shot, a more complex Nuke script was necessary as we needed to integrate FX elements, all of which were rendered seperately. We also chose to render the foreground and background seperately so that we could make adjustments to them independently.

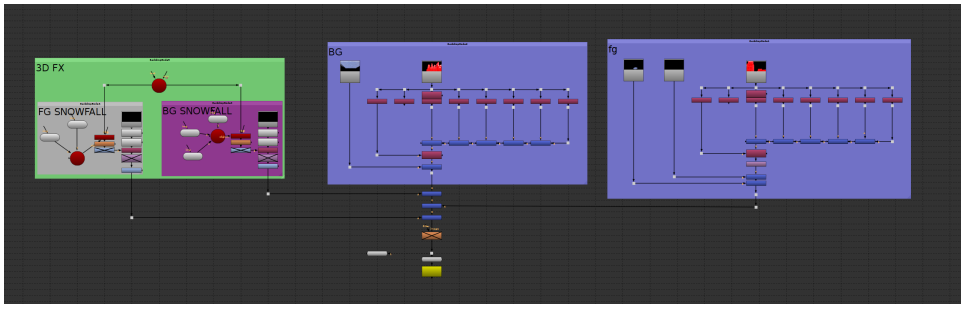

Figure 8: The Nuke script which was used for the opening shot.

6.2 Results

7 Reflection

References

[1] Python Software Foundation. pickle python object serialization, 2018. Accessed 09 June 2018.

[2] Ben Martin. Parallel ssh execution and a single shell to control them all, 2008.

[3] Stefan Muller. Firefly removal, Feb 2015. Accessed 16 May 2018.

[4] Fabric Software. Kraken - rigging framework, 2015. Accessed 23 December 2017.

[5] Animation Studios. Advanced skeleton, 2016. Accessed 23 December 2017.

[6] www.parallel ssh.org. Parallelssh - asynchronous parallel ssh client library., 2014. Accessed 8 October 2017.

Figure 9: Comparison between the 'beauty' pass generated by RenderMan (Left), and the compositing I was responsible for (Right).

Listing 1: Extract from the Python script used to determine which of the computers on the university network are suitable for rendering on. By capturing the output of the following commands I was able to get the current CPU and memory usage.

```
# return the current cpu usage
top -bn1 | grep 'Cpu(s)' | sed 's/.*, *\([0-9.]*\)
# return the current memory usage
free | grep Mem | awk '{print $3/$2 * 100.0}'
```