Vladislav Kazakov

The Role of Python in Visual Effects Pipeline

Case: Talvi Tools

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Media Engineering

Thesis

21 September 2016

Helsinki
Metropolia
University of Applied Sciences

Author(s) Title Number of Pages Date	Vladislav Kazakov The Role of Python in Visual Effects Pipeline Case: Talvi Tools 41 pages + 1 appendix 21 September 2016
Degree	Bachelor of Engineering
Degree Programme	Media Engineering
Specialisation option	Audiovisual Technology and Production Systems
Instructor(s)	Antti Laiho, Senior Lecturer Kaj Lydecken, Animation Supervisor

The purpose of this thesis was to study the concept of visual effects pipelines and analyze how programming language Python can fit into the post-production phase of filmmaking. In the extremely competitive environment of visual effects industry, companies are forced to constantly look out for newer technologies and research more optimal production methodologies. In search of feasible solutions studios often come across Python.

The theoretical part of the study outlines a brief history of Python and illustrates the power of this programming language with two exemplified use case applications. In addition to that, various possible Python implementations in the production of computer generated imagery are extensively reviewed. To give a more complete picture of how this programming language aids post-production, the adoption of Python by Industrial Light & Magic is examined. As it became clear that Python would prevail in the production of computer graphics, software vendors started embedding Python support in their products. This claim is further supported by the analysis of Python integration within major contect-creation applications.

The outcome of this study is the add-on for Blender developed in Python. The purpose of the add-on is to facilitate and accelerate the export of character and camera animation data, which is specifically useful for projects that require a substantial amount of computer animation to be moved from Blender to another software for later use.

Keywords	Blender, VFX, pipeline, Python, post-production, filmmaking, effects, animation

Contents

1 Introduction 2

2 Python in Visual Effects Pipeline 3

2.1 The concept of a Pipeline 3
2.2 Overview of Python 9
2.3 Python in post-production workflows 12
2.4 ILM case study 15

3 Overview of Python integration in Visual Effects Software 16

3.1 Autodesk Maya 16
3.2 SideFX Houdini 20
3.3 The Foundry Nuke 23
3.4 Blender 25

4 Development of Blender add-on Talvi Tools 27

4.1 Case Company Introduction 27
4.2 Talvi Digital's Pipeline 27
4.3 Initiation 28
4.4 Planning 29
4.5 User Interface 30
4.6 Development 33

5 Practical implementation and Analysis 36

6 Conclusion 38

References 39

Appendixes
Appendix 1. Talvi Tools v 1.0 source code (Written in Python programming language).

Helsinki Metropolia University of Applied Sciences

Terms and Abbreviations

3D	Three Dimensional
MEL	Maya Embedded Language
API	Application Programming Interface
CG	Computer Graphics
OOP	Object Oriented Programming
VFX	Visual Effects
OBJ	The Wavefront 3D object file format
MDD	Motion Designer Document file format
FX	Effects
GUI	Graphical User Interface
DAM	Digital Asset Management
FPS	Frames Per Second
DCC	Digital Content Creation
TV	Television
IDE	Integrated Development Environment
R&D	Research and Development
VEX	Vector Expression Language
FBX	Autodesk Filmbox file format
ILM	Industrial Light & Magic

1 Introduction

It is hard to deny that nowadays computer generated imagery and visual effects have become an indispensable part of TV, film, animation and media industry as a whole. Ever since the beginning of film making visual effects have provided directors a freedom to express the vision of a story that a narrator aims to convey. The early years of filmmaking relied on practical effects, such as stop-motion animation, miniatures and traditional matte paintings, among others. Of course, they were truly stunning and caused “wow effect” from the audience. Nevertheless, as time progressed, technological advancements caused a massive leap forward in visual effects and pushed visual effects in movies to an entirely new level. Notably, one of the greatest advancement in terms of computer-generated imagery was the first cartoon feature Toy Story created entirely through the use of a computer by Pixar in 1995. This, as a consequence, contributed to the success of Pixar and boosted the popularity of full-CG animated feature film productions. What is more, the evolving technologies used in these productions also promoted the quality of computer-generated elements composed into feature films. [1] The rapidly developing tools have enabled visual effects and digital artists to render remarkably realistic life-like results. These days, when the digital content creation software is becoming more and more accessible to a wide audience, computer graphics has spawn the interest among many enthusiasts. Hence, there is actually no surprise that the amount of VFX used in films is continually growing.

As in many other industries, VFX companies are eager to streamline production process that would lead to financial cost and time savings while increasing overall productivity and profitability. While most visual effects work is completed during post-production, it should be carefully planned and prepared beforehand - during the pre-production and production stages [2]. However, for the purpose of this thesis the latter post-production part, likewise the production of CG animation will be given in-depth attention. A production process or production workflow in CG industry is often called with a term “CG pipeline” or “VFX pipeline”. In brief, it is extremely crucial to design a strong pipeline that would fit the needs and capacity of a studio and provide the ability to empower the production process. The following chapters will shed some light on the benefits of a solid pipeline and will provide a better understanding of the role of pipeline management plays in production.

This study aims to demonstrate how programming language Python fits into visual effects workflows and serves as the backbone of any successful project. The thesis is divided into two parts: theoretical and empirical. The theoretical part will explore the concept of a pipeline

and its architecture with an illustration of an exemplary case from a well-known visual effects company. Apart from that, the theoretical overview and a brief development background of programming language Python will be covered. Next, Python employment within major digital content creation packages will be examined and extensively reviewed. The empirical part of the study will expose a use case scenario of how Python programming contributed to the post-production of Finnish feature film project and will go over the steps involved in the development of Blender add-on with the use of Python.

2 Python in Visual Effects Pipeline

2.1 The concept of a Pipeline

In regards to visual effects, the term pipeline refers to a flow of steps, or production stages needed to generate a VFX shot. A parallel can be drawn with a manufacturing production line, where raw materials, comparable with the live-action plates, come in and move through a series of repeatable processes that transform them into finished products. The finished product in visual effects, also called a “final deliverable” is a collection of image sequences that are later on edited and stitched together to form a movie. Even though making of effects and animation resembles a manufacturing production model, there are a few noteworthy factors that make a visual effects production distinguish from a manufacturing line. Essentially, a production or assembly line implies considerably linear process whereas VFX production is far more complex and ideally adequately flexible to support data flowing back and forth through the ”pipe”. [3, 9.]

Evidently, visual effects pipelines differ among the companies depending on the structure of a company, size of the crew, computational capacity and production needs. The chart below represents an exemplary schematic illustration of a simplified generic pipeline.

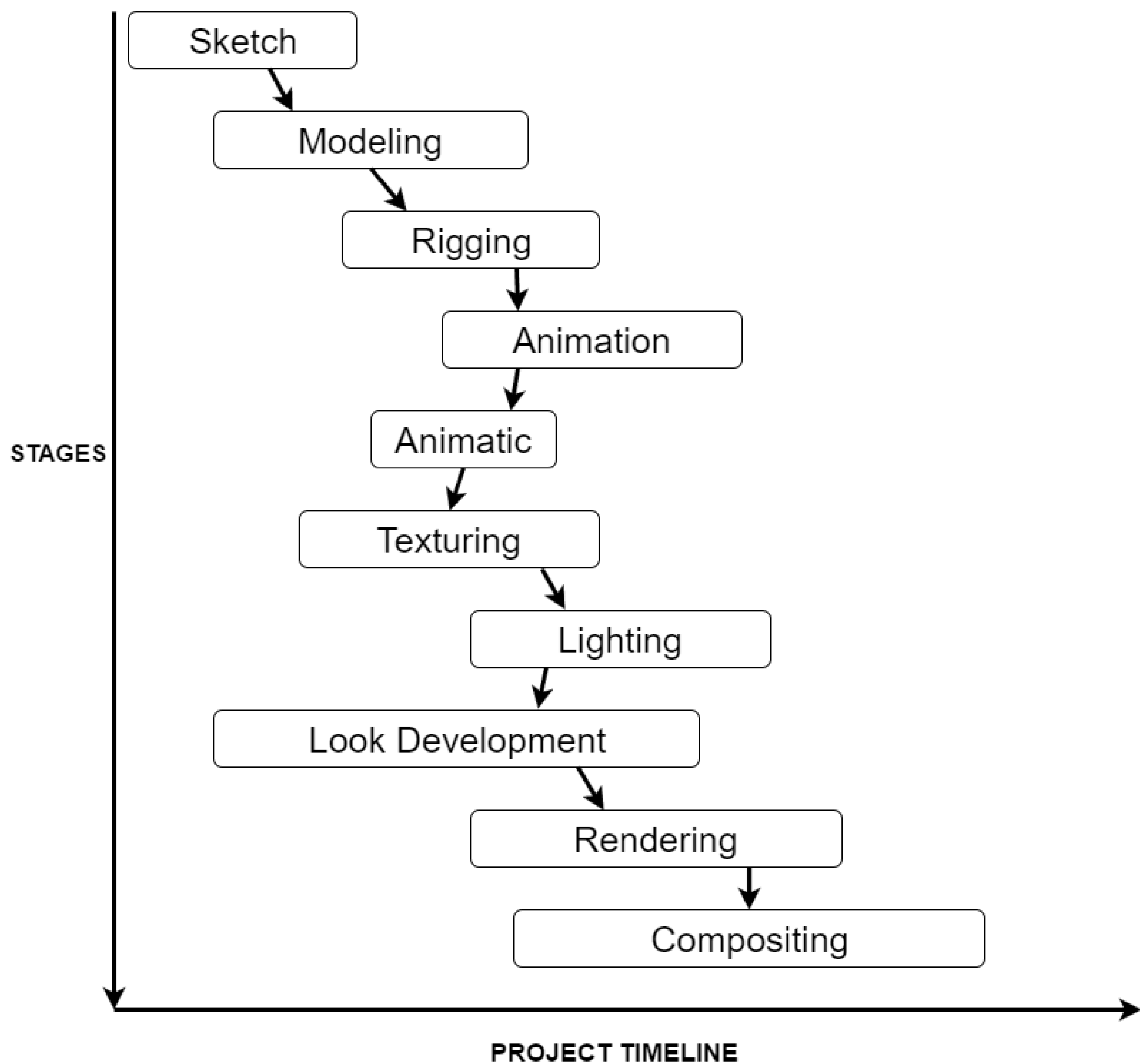

Figure 1. Example of a visual effects pipeline. [4]

With the reference to flow chart presented in figure 1, the pipeline is distributed into production stages. As a matter of fact, every production stage has its own specific time period. What is more significant, there are intersecting time periods where production stages overlap. To clarify, a rigging stage usually takes off when a model is ready to rig and an animation can start as soon as a rig is prepared. In other words, a rigging stage does not necessarily have to wait until all the modelling is finished and so forth. Each stage gets allocated to a respective department and depending on the size of a facility, it can be a single person or a group of people. [4]

Important to realize, that there are a number of measures a VFX company must take in order to stay afloat, complete work on schedule, while keeping rates as competitive as possible. One of them is to develop a robust pipeline, which would help to avoid bottlenecks, support

iteration and maximize efficiency. One example of a such "VFX powerhouse" can be considered the pipeline designed and implemented by Rhythm & Hues in 2006.

Rhythm & Hues (R&H later in text) was one of the top global post-production and animation company founded in 1987 and headquartered in Los Angeles, California. Among the most notable projects produced by R&H are award-winning "Babe", "The Golden Compass" and "Life of Pi". Besides headquarters in Los Angeles, R&H also had facilities in other parts of the world: Vancouver, Mumbai, Hyderabad, Kaohsiung and Kuala Lumpur. [5]

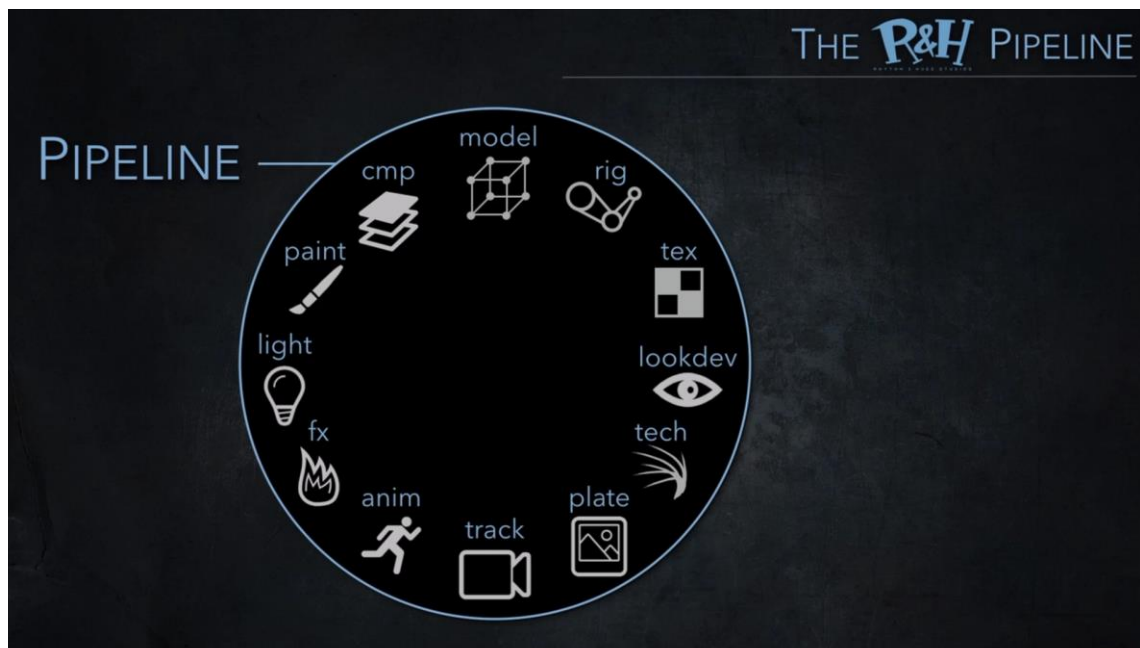

Figure 2. The R&H pipeline compromises of 12 creative disciplines. [6]

One of the key points of Rhythm & Hues pipeline is that every production discipline with regards to both 2D and 3D disciplines treats and communicates with the data flow in the same manner. For this particular reason, R&H pipeline is considered to be unified.

Other notable features include:

- Context – everybody works within the same context
- Interaction – everybody interacts with the file system in the same way
- Sharing – the way everybody shares the work remains consistent across all stages
- Unified toolset – everybody uses the unified toolset
- Language – everybody speaks a common language

All Rhythm & Hues facilities are operated through a single pipeline. Nowadays, in the age of digitalization in the movie industry, the innovative technologies of high-speed file transfers have led to a reduction in barriers to entry. With this in mind, from the workflow point of view there is no big difference between a colleague located nearby or in the R&H location on the other side of the world, e.g. in Kuala Lumpur. As a result, work is divided among all facilities based on employee availability at a given location and therefore production does not get isolated to one single facility. Moreover, work within a single pipeline stage can be accomplished by different locations at the same time. It can start in one facility and then head over to another one if necessary.

Figure 3. Division of work of one shot from “Life of Pi”. [6]

1. Background plate preparation in Kuala Lumpur
2. Matchmoving and tracking in Hyderabad
3. Animation in Mumbai
4. Lighting and FX in Los Angeles
5. Rendering in Kaohsiung
6. Compositing in Mumbai

The technology that allowed Rhythm & Hues to spin a shot around geographically diverse facilities as one R&H representative stated justly that “the sun never sets on Rhythm and Hues”, also allows other companies to outsource their projects to 3rd party vendors or distribute work based on the human resources available and possible financial incentives.

Furthermore, the R&H pipeline supports comprehensive sharing of assets and other production data across locations by thoroughly tracking production dependencies inside out. It relies on a subscription model, which links everything together and make it possible to track what versions of which assets are in use in which locations. That is to say that production assets can be synchronized to other R&H locations without any additional stress on an artist.

What makes R&H pipeline especially outstanding is the fact that it was developed to support any type of production that could be ordered to the company. For this exact reason, the pipeline framework was originally planned without any expectations about the type of projects being done, software being used or the types of data being passed around the facilities. As a result, this has allowed to build the highly adaptable unified pipeline that embraces all steps of post-production. As previously mentioned, there is a subscription model, which not only allows smart synchronization and seamless multisided asset management from any location, but also stores the entire revision history of production, including both the work area and asset versioning. The pipeline was shown clearly to be highly scalable and has been in use on feature film and commercial projects of different scale whether it is a 10-shot commercial project or a 1000-shot feature-length production. Due to the fact that the pipeline was designed with no assumptions of what type of projects might be awarded to the company, it was developed flexible enough to handle any types of shows: effects-heavy, character-heavy or/and Full-CG. [6]

A significant change is going on in the movie industry as the use of VFX dramatically increases. As we all know, it is not just the number of shots, but also is the complexity of visual effects elements in filmmaking that is constantly growing. Apparently, and so does the challenge of managing and tracking of data throughout the pipeline. These days when the project deadlines are getting shorter and shorter and the pressure to complete work on time is rapidly increasing, a pipeline organization becomes absolutely indispensable if a company aims to deliver the ever-increasing shot complexity and volume in an effective way without obstructing the creative side of work. Compounded by shrinking budgets and schedules, post-production studios need ever more complex and comprehensive digital production management and shot tracking solutions. These changes, in turn, outline what factors are inherent to a CG pipeline of efficient post-production:

1. *Communication.* Communication should always be open and maintained on every possible level: within a department, in-between departments, between artists and other studio personnel, e.g. Human Resources, IT departments and such. Everybody should know what to do and be informed of the changes if there are any. This concerns also

communication between the studio and its customers. Supervisors should thoroughly understand customer needs and communicate them to the artists. [3, 23.]

2. *Benchmarking.* Tracking the time and resources to complete a certain types of shots can help to estimate the cost of production for future projects [3, 24].
3. *Flexibility.* Since a customer or a director can often change his mind, a pipeline needs to be highly adaptable to such changes. These changes can often impact different stages of production. In the best possible scenario, a pipeline should be adjustable enough to support iterations and asset tweaking at any stage of production. It is important to understand, that while artists are often happy to make revisions, they can have a significant impact downstream. Therefore, a CG pipeline should be bidirectional in case data needs to pass upstream or downstream. [3, 13.]
4. *Scalability.* A pipeline should impeccably be capable take on a production of varied scale. This means not only the overall scale of project, but also the functioning capacity of different departments involved in the production can be diverse.
5. *DAM.* Both asset and project supervision are the integral aspects of any CG pipeline, which allow managing supervisors to review asset preparation, versioning and observe the status shots produced [3, 23].
6. *Automation.* Nearly every phase of CG production can be standardized and automated. The main purpose is to avoid repeatedly manual labor as much as possible. By automating some of the tasks, e.g. inputting file paths, adjusting settings and presets, setting up plugins and add-ons, one can make production smoother, provide artists with non-distracting environment and give more freedom to perform their job. [4]

As it has been previously highlighted, one of the major challenges remained in post-production pipeline is handling the staggering amount of data that is required to output a high-grade imagery. Typically, a standard character in an effects-heavy movie can encompass several hundreds of sub-assets that must be compiled together to generate a working result. Some of them are high-resolution geometry caches and texture maps, rigs, skin, muscle, cloth and hair simulations, along with others. [7]

Referring back to figure 1, as has been pointed out, a pipeline is divided into stages, which get allocated to a corresponding department. Keeping this in mind, there is one pivotal nuance that each department operates with different software or sometimes with a mixed set of software tools. While they are great on their own, these tools were not intended to interact nicely with each other in the first place. Still, often times there is a need to synchronize sub-assets

and assemble all the pieces into a masterpiece. Therefore, the issue arises of finding an optimal solution as how to get this diverse set of tools to collaborate with one another. This is where programming language Python comes in handy.

2.2 Overview of Python

Python is a high-level multi-purpose programming language that supports a variety of programming paradigms. It was created by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands [8]. The first version Python 1.0 was released in February 1994, followed by the next major update to version 2.0 in October 2000 [9]. Python 2.x development cycle was a huge step forward in terms of development – by adding the unification of Python built-in types and classes created by the user into same hierarchy, which made Python totally object-oriented language. In addition to that, the support of a garbage collection system capable of collecting reference cycles was also added. This update to the class system of Python introduced a number of new features, which overall improved the programming experience. [10;11] The next serious milestone was the release of a backwards-incompatible Python 3.0 in 2008. Although this version was designed with the same philosophy as the previous ones, the 3.x development cycle put bigger emphasis on getting rid of duplicate designs and modules, which have become redundant and obsolete by that time. [12] Python 2.x and Python 3.x series coexisted in parallel for a few releases, with Python 2.x series released mainly for the sake of compatibility and included several features being back-ported from the 3.x series. For example, Python 2.6 was released to accompany Python 3.0, and had some features included in that version and so forth. In the similar way, the next release of Python 2.7 included functionality from Python 3.1. Most of its main features were also back-ported to the backwards-compatible Python 2.6 and 2.7. The support of multiple programming paradigms such as object-oriented, imperative, procedural and functional makes Python very powerful and thus applicable in a wide spectrum of the following programming areas:

- Web development
- Database programming
- Server and network programming
- Rapid Prototyping
- Scientific and numeric calculations
- Graphical user interfaces
- Game development
- System Administration
- Education

In order to give a more complete picture, some real life applications will be illustrated as an example. In particular, Dropbox, one of the most popular global cloud-based file hosting service that offers personal and corporate cloud storage.

Dropbox is a service that allows users to send and store files on a remote server using a client software or a browser using web interface. Once a file or a folder is sent, Dropbox then automatically synchronizes it so that it will be available as the same directory with all the files, no matter which device is used to browse it, whether it is a desktop computer, laptop or a smartphone. Files contained in this folder will also be accessible via the Dropbox website or Dropbox mobile applications. [13]

What is more significant is the fact that Dropbox development team used Python for nearly everything. It is utilized in the desktop client software, back-end infrastructure, website logic and usage analytics. In addition, it runs on a single code base and relies on the following Python libraries: Twisted, PyObjs, WxPython, py2exe, py2app and PyWin32. [14]

Another striking example to illustrate the flexibility of programming language Python is one of the first major multiplayer online games EVE Online, which was released by CCP Games in 2003. EVE online is a science fiction adventure game that features a massive game set that takes place in the distant future where players can explore and battle with other players across the universe comprised of more than 7000 star systems and planets. Stackless Python, a lightweight micro threaded version of Python was tightly integrated since the early stage of development of the game. Python significantly assisted in the development of the client side of EVE Online as well as the back-end architecture. One major advantage of Stackless Python over the default Python is that it has allowed EVE Online to effectively scale up to a comparatively large number of users performing tasks with no overhead of using the call stack that comes with standard Python distribution. EVE Online has around 40 thousand simultaneous active players on average. [15]

In fact, Python ranks in the top five most popular programming languages, along with Java, C, C++ and C#. It has become a higher priority programming language for many big companies, such as Google, Netflix, NASA, Reddit, IBM and many other organizations start to follow this tendency and employ Python for their solutions. Here are the most compelling reasons, why it is embraced by many users who want to take full advantage of all the benefits it has to offer:

- **Accessible**. Being an open-source language, Python is completely free to download and use. This can already contribute to project cost reductions. Besides that, it requires very little setup to run.
- **Cross platform**. Python is supported by all major operating systems: Windows, Linux distributions and Mac OS.
- **Easy to grasp**. The reason behind that is a very simple syntax, which in turn reduces syntactical overhead and steepens the learning curve. It is largely recommended as a good choice for a first programming language to learn.
- **Readability**. Indeed, Python has a very clear syntax enhanced by a set of punctuation rules and is easy to read and understand because it bears close resembles to English language.
- **OOP support**. By creating, using and re-using data structures, one can minimize the amount of coding required to accomplish a certain task.
- **Extensive libraries**. Python comes with a massive standard library and extensions for any type of programming tasks.
- **Interpreted**. Python is processed at runtime and there is no need to wait for the compilation phase to complete thus reducing development time and stimulating the learning process.
- **Large community.** Not only Python user community is very large, but also it offers a lot of support, shares resources and in general stimulates the learning process.
- **Transferable.** Python is very portable and can work on a wide gamut of various hardware devices.
- **Great documentation.** Python is equipped with comprehensive documentation making problem solving much easier.

2.3 Python in post-production workflows

This chapter puts the emphasis on how Python fits into a post-production pipeline. There are several ways how Python can aid the making of visual effects. First and foremost, one of the most crucial application of Python in the field of computer graphics is the development of pipeline on any scale of production facility whether it is a large or small studio, or enhancing productivity output within a personal work place. The main question often arise during production is how to optimize the project production workflow in as many ways as possible.

On a global scope, there are several primary concerns to be addressed that can be classified according to the following categories:

Production global variables:

a. *Settings.* This refers to digital content creation software settings and preferences, which can be general or project wise. These global variables should be pre-defined and synchronized for all work stations across the facility. For example, when an artist launches a program, it automatically loads the correct settings for the project he/she is working on. Such settings include but are not limited to render resolution, FPS, deliverable destination, and etcetera. Ideally, this procedure should be highly customizable and automated through the use of global variables.

b. *Modules.* This concerns various scripts, plugins, add-ons and other software extensions, including both 3rd party and in-house solutions. For instance, if a company has a dedicated R&D department whose job is to develop software scripts and extensions, these extensions should be automatically updated and put in sync for all computers.

Project working environment:

a. *Project root.* Assignment of a software for a certain environment, which tells the program what project it works with. For example, a $JOB variable in Houdini. By doing that, a software should intelligently detect the hierarchical project folder structure and fetch necessary files (models, bitmaps, caches and etc.) automatically.

b. *External data.* In case the assets are stored externally or remotely, the software should automatically know where to reference this data from.
[4]

Having to deal with the aforementioned concerns, pipeline engineers and technical directors face some of the most challenging problems. As a rule, they use Python programming to tackle these problems and help facilitate the production workflow.

Here is a more detailed analysis of pipeline related issues that can be solved with Python:

1. Automation. It is often alleged that automation of repetitive and routine tasks is one of the main reasons why Python found its way into VFX industry. Such tasks might involve any number of monotonous animations, e.g. copy-pasting animation keyframes, setting up similar character rigs, copying light setup from one scene to another, setting up and adjusting several scene parameters, defining output destination and many different others. As long as the task can be described as an algorithm, it can be scripted in Python. The ultimate goal is to automate the workflow as much as possible in order to remove the burden from the artists and let them focus on art creation.
   Another often overlooked aspect has to do with the human error component, as people do occasionally make mistakes, typos, spelling errors and such. By automating some of the tasks, one can reduce the human error component that in turn would result in much more predictable outcome. [4]
2. Software extension. Some post-production companies have a dedicated R&D department that works on proprietary in-house tools and also extends the commercial software by addressing the needs of artists who depending on the complexity of a task assigned might require an off-the-shelf solution. Granted that the majority of 3D programs have a Python API, which allows to communicate with the core of software, in conjunction with rich Python library modules presents the environment where there are countless possibilities to build new tools, scripts and plugins to go far beyond the default feature-set. Specifically, it is commonly adopted in the character rigging process, when technical directors need to build complex dependencies between parts of a model and control or constraint systems and Python is used to link them together. [16]
3. Pipeline/workflow enhancers. There is a vast array of utilities that can be made with Python and be helpful in visual effects productions, that are not directly related to DCC software. For example, various scripts that can execute shell commands for various purposes: scan directories and delete bitmap files with specific metadata, batch convert files, resize images that match a certain criterion. There are also multiple modules to work with image processing in Python. Such include OpenImageIO, OpenEXR and PythonMagick. They allow for basic analysis and manipulation of both image data and metadata. Apart from that, there also modules for color manipulation that enables to convert image from one color space to another or linearize the image – OpenColorIO

and ColorPy. [17] Although most established VFX studios rely on comprehensive DAM systems, individual artists and start-ups can make use of Python programming and develop their own solutions in the matter of file transferring, management and synchronization. The same applies for project management. It is not uncommon for a facility to have some small desktop apps that artists would use to track jobs, deadlines and deliver project statistics for benchmarking. By and large, Python can be integrated into any aspect of the post-production – it all comes down to the specific problem at hand.

4. Smooth stage transition**.** Bringing up the fact, that in certain cases every stage of pipeline is done in different software and sometimes a single stage can be completed with a diverse assortment of tools, the issue of management and configuration of digital content creation software occurs. Most of the tools used in VFX production were not developed to easily interact with each other, besides maybe supporting a few common file formats. In this scenario Python serves the underlying role of the glue that ties pieces of assets together. There are different Python modules for content and data transfer across distinctive software tools. Alembicgl is a module that allows to work with alembic, a file format often used for geometry caching. Similarly, pyopenvdb is a module that provides with an access to VDB volumes data management with Python. [17]

Taking into account all the advantages of Python listed in the previous chapter, there is clearly no doubt it has become the de facto industry standard programming language in visual effects productions and software vendors started implementing Python support into their products. It is widely supported by an extensive list of digital content creation applications such as:

1. Autodesk:
   - Maya
   - 3ds Max
   - MotionBuilder
   - XSI
2. The Foundry:
   - Nuke
   - Katana
   - Hiero
   - Mari
   - Modo
3. Blender Foundation: Blender

4. SideFX: Houdini
5. MAXON: Cinema 4D
6. Blackmagic Design: Fusion
7. NewTek: Lightwave 3D
8. Next Limit Technologies: RealFlow
9. Shotgun Software: Shotgun

2.4 ILM case study

Founded by George Lucas in 1975, ILM is one of the leading global visual effects company with facilities in San Francisco, London, Vancouver and Singapore. It is famous for the projects like Jurassic Park, Star Wars, Forrest Gump and Terminator 2. In past, ILM's post-production pipeline relied mostly on Unix shell scripting. As the amount of visual effects used in movies started to grow, ILM started to research for more optimal ways of overseeing progressively complex production processes. By that time, Python 1.4 was released as simple yet effective programming language that could be utilized in place of Unix shell scripting. It caught the attention of ILM, who was looking for more superlative solutions to replace their older scripts. Due to the simplicity and smooth learning curve it was chosen by ILM, which appreciates the speed of development. Besides that, Python could be integrated with more sophisticated software systems and interact with other programming languages. Taking advantage of that, ILM implemented Python into their proprietary tools. [18]

> Python is everywhere at ILM. It's used to extend the capabilities of our applications, as well as providing the glue between them. Every CG image we create has involved Python somewhere in the process.

Philip Peterson, Principal Engineer, Research & Development, Industrial Light & Magic. [18]

Indeed, not only Python was used to extend the content-creation tools, but it was also implemented to enhance the user interface of ILM's DAM system. After the embracement of Python in 1996, it has been applied in many different ways in ILM – asset management and batch control, software extensions, database development and many others. [18]

> Python plays a key role in our production pipeline. Without it a project the size of Star Wars: Episode II would have been very difficult to pull off. From crowd rendering to batch processing to compositing, Python binds all things together.

Tommy Burnette, Senior Technical Director, Industrial Light & Magic. [18]

3 Overview of Python integration in Visual Effects Software

The following part of thesis will more closely examine Python integration within some of the popular visual effects applications. Despite the fact that most of these applications are shipped with their own scripting language, the support of Python has definitely helped to extend the standard functionality and get them to communicate better with one another.

3.1 Autodesk Maya

Initially developed and released by Alias|Wavefront in 1998 Maya has become undoubtedly one of the most popular 3D software packages used in the production of animation, games, film and TV. Acquired by Autodesk in 2006 Maya has been packed with many tools for a wide gamut of tasks, ranging from polygonal modeling and sculpting to complex FX dynamics and rendering with native and 3rd party engines. Over the years, Maya has grown into feature-rich program that is favoured by a majority of visual effects studios, regardless of their scale, along with individual artists. Some of the latest features include BiFrost, a realistic fluid-implicit particle simulator, xGen, procedural geometry instancing and scattering framework and the new sculpting toolset migrated from Autodesk Mudbox, just to name a few [19]. For this reason, many companies position Maya in a crucial place of their production pipeline.

Maya is equipped with MEL, its native scripting language that has a similar syntax with its predecessor Tcl, a scripting language used in the very early days of Maya. MEL is not just a powerful scripting language – it is a tool and a method to configure and modify the basic functionality of Maya, as most of Maya's environment and tools are written in MEL. As an example, similar to creating actions in Photoshop, a Maya user can record his commands as a script in MEL that would build a user-friendly macro. MEL can help speed up repetitive tasks, as well as give an access to features that are not directly available through Maya interface. MEL is also platform independent language meaning that it is not tied to the operating system. As a result, the code written in it will run on any platform, which Maya runs on. [20]

In addition to MEL scripting, a support of Python was added since Maya 8.5 was released in 2007 [21,15]. Python scripting can be used for many tasks in Maya, from running simple commands to developing plug-ins, and several different Maya related libraries are available to target different tasks. Maya has a bilateral Python and MEL compatibility. In other words, Python scripts can be executed with MEL and the vice versa. The following list is a brief overview of Python libraries shipped with Maya:

- **Maya.cmds** – also known as Maya commands. This module has the same capabilities as MEL. Everything that can be done with MEL can be done using this module.

- **Maya.OpenMaya** - grants an access to Maya API with Python. Different types of extensions or plugins can be made using this module.

- **PyMEL.core** - Pymel is an open-source Python library and is another wrapper for MEL. Although it is not officially supported by Autodesk, it still is shipped with Maya for user convenience. [22]

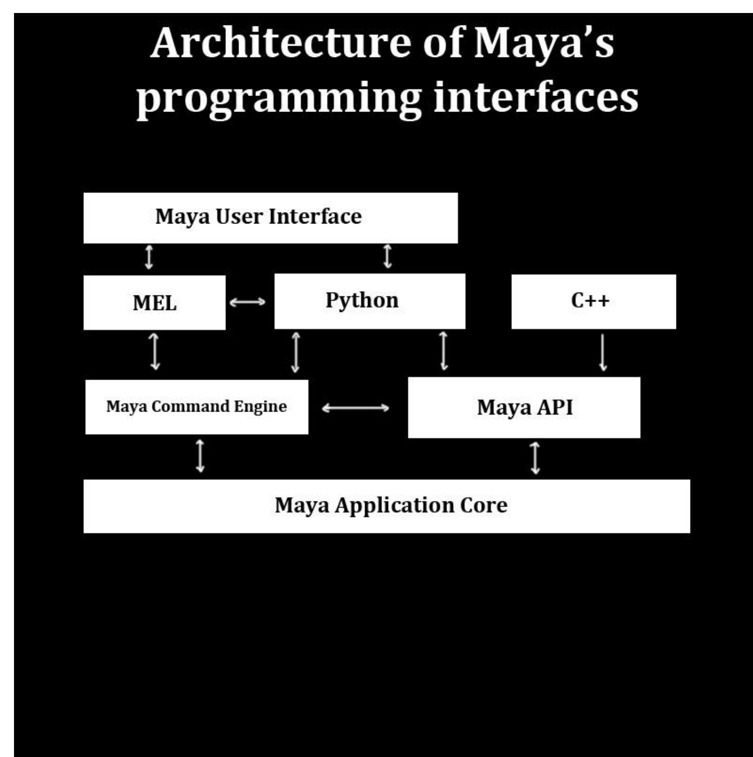

Figure 4. The architecture of interaction between Maya's Core and a user. [23]

As illustrated in figure 4, both Python and MEL enable a user to communicate with Maya Application Core. However, unlike MEL, Python allows this communicate through Maya API in addition to Maya Command Engine.

The table below will take a closer view at some of the key differences between MEL and Python and should help a user to choose between the two.

Table 1. Differences between MEL and Python in Maya. [21,15-16; 24]

Difference	MEL	Python
Concept	Native Maya language, which is used to communicate with the core of Maya and only works inside of Maya	VFX industry standard scripting language that can be used with other pipeline elements
Development speed	MEL has not changed much it since the time it was introduced with the first versions of Maya. It mainly grows by commands that are added to Maya itself	Python is an advanced scripting language, that continues to rapidly grow
Community size	Small community of Maya users	Huge community and user base that continually increases
Library size	Large library of MEL scripts	Extensive libraries for a variety of purposes
Dynamic typing	No dynamic typing	Supports dynamic typing
Object-orientation	No support for object-oriented programming	Supports object-oriented programming
API access	No API access, i.e. Maya functionality cannot be extended with MEL	API access allows the use of classes and methods specified in Maya API and enables the development of new nodes and plugins
Variety of IDEs	Small variety of IDEs, mainly Maya script editor	Wide variety of IDEs

To execute Python code in Maya, it can be entered in multiple ways:

1. Script Editor. The most common way to input Python is via Maya's Script Editor, which can be accessed through: Window → General Editors → Script Editor. The Script Editor supports both MEL and Python scripting and has dedicated tabs for each language. The Script Editor's user interface consists of two main panels – History Panel on the top and Input Panel in the bottom. Maya's Script Editor also has an auto-completion feature that developers can benefit from. [25]

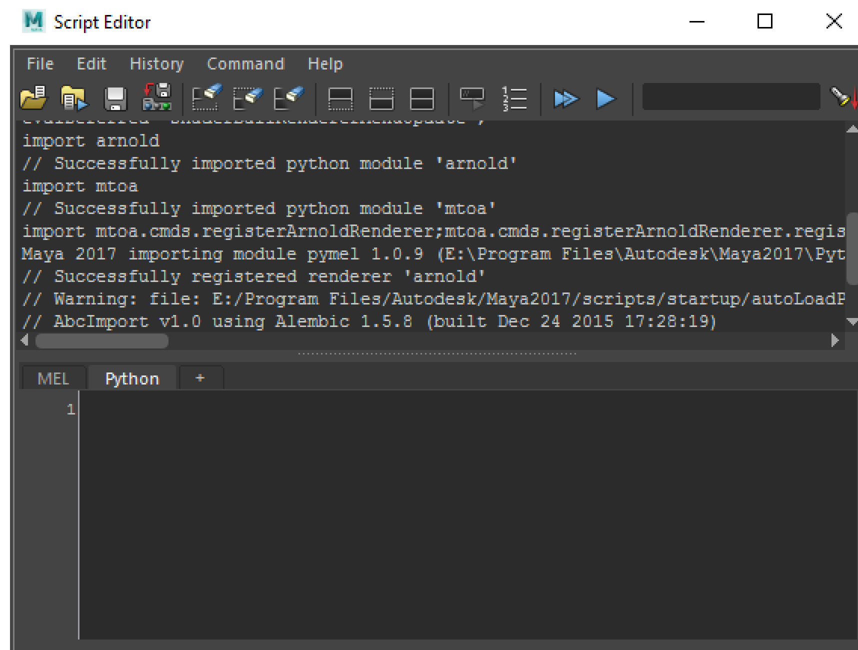

Figure 5. Script Editor in Maya 2017.

2. Shelf Editor. Additionally, in case a user wants to maintain the script and call it several times during the session, or just to have the script always in hand, there is a convenient way to store it as a shelf tool, that would execute it once clicked.

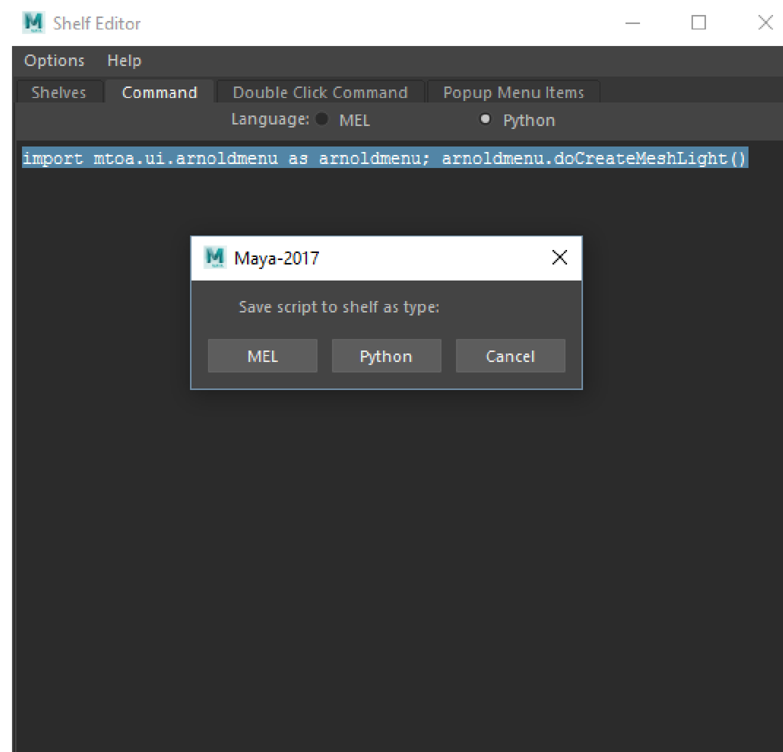

Figure 6. Shelf Editor in Maya 2017.

3. Command line. Another possible way to input Python is through Maya's command line. A user can toggle between Python and MEL to choose, which language to execute. A command line has a limitation to input only one line of code.

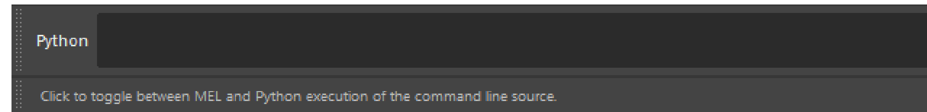

Figure 7. Command line in Maya 2017.

3.2 SideFX Houdini

SideFX Houdini is a node-based 3D animation software package that is famous for its broad capabilities and powerful procedural tools. What makes Houdini so unique and different from other content-creation applications is the fact that if offers a visual programming environment where artists have a freedom to create their own tools and expedite their workflow in nearly limitless possible ways. That is why many visual effects studios opt for this software and implement an end-to-end Houdini pipeline.

Houdini comes with HScript, the old Houdini scripting language that nowadays is primarily used for compatibility with older Houdini project files and scripts. HScript has some parallels with MEL in Maya. While HScript is the early scripting language used in Houdini, it still remains handy in some parts of the software. Given that, HScript can be applied to script some object transformation expressions, edit properties of lights, cameras, render outputs and other types of objects. Nevertheless, it is not so efficient when it comes to modifying geometry and doing geometry or point related operations. In this scenario, the more practical solution would be to go with VEX. [26]

VEX is the core Houdini language, and is extremely efficient and bears similarities with C and C++ languages. Unlike Python, VEX is very well multi-threaded and for that reason is suitable in the following parts of Houdini: [27]

- Shading computation
- Particle simulation
- Geometry modification
- Fur
- Channel operators

Python scripting came with the release of Houdini 9. Possibly, Houdini has one of the finest Python integrations among CG software, which consequently has fostered the development progress and enabled artists to extend the toolset even more. As figure 8 shows, Houdini is bundled with an embedded Python interpreter that lets technical artists to create their own modules and import other Python modules too.

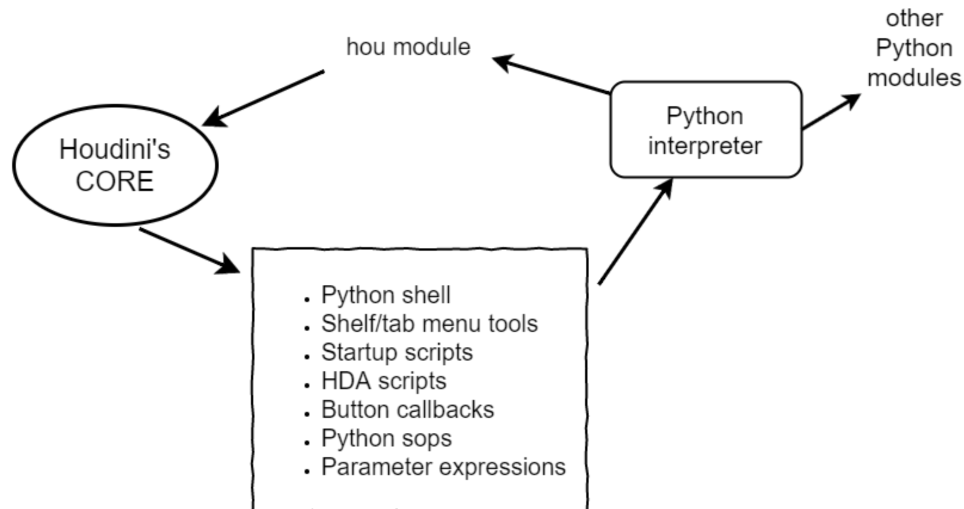

Figure 8. Houdini’s Operating System Process [28]

There are a number of ways to use Python within Houdini:

1. Python Shell. Python Shell can be accessed via: Windows → Python Shell, or alternatively via a shortcut Alt+Shift+P. Houdini’s Python Shell is very well suited for prototyping scripts. [29] The top line in Python Shell indicates the version of Python installed in Houdini.

```
Python Shell
Python 2.7.5 (default, Oct 24 2013, 17:49:49) [MSC v.1700 64 bit (AMD64)] on win32
Houdini 15.0.244.16 hou module imported.
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Figure 9. Houdini’s Python Shell.

2. Python source editor. This editor can be accessed via Windows → Python Source Editor. It is worth mentioning, that with this editor the text code can be saved in the project file and can be run on project startup.

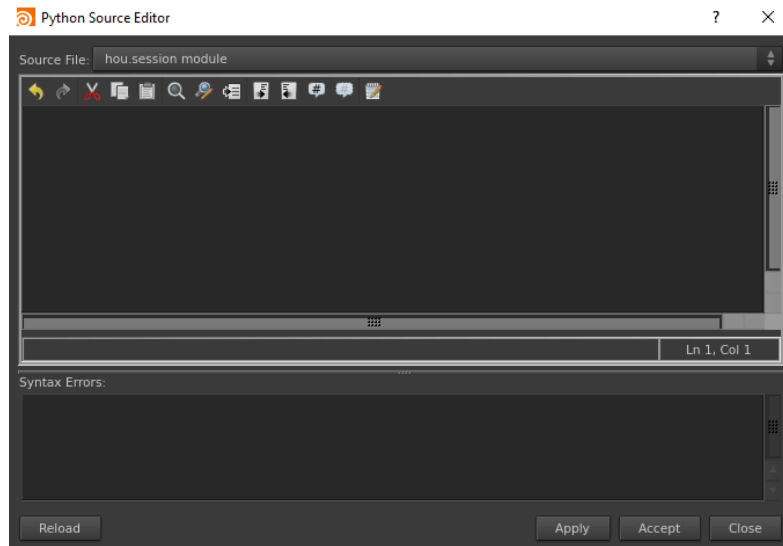

Figure 10. Python Source Editor in Houdini.

3. Shelf button. Similar to Maya, Python code can also be hooked to a shelf button. The majority of native Houdini shelf tools refer to Python script and can be used an example for custom tools. Optionally, Shelf button editor allows to choose between Python and HScript. [29]

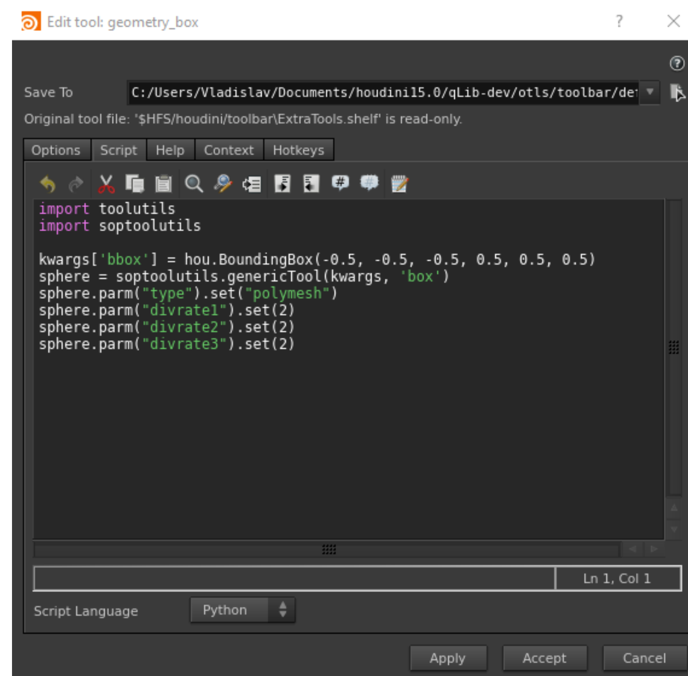

Figure 11. Shelf button editor.

4. Expressions. Most Houdini node's parameters can be interpreted with expressions. By default, HScript language is used. However, a user can switch it to Python and use it to script a parameter.

Figure 12. Choice of expression language in Houdini.

5. Python SOP. Like VEX wrangles, Python SOP is a node that allows to enter Python code and modify the geometry or points with it. It can be useful in certain cases, when it is more convenient to write a few lines of code, rather than dealing a mess of node trees. [29]

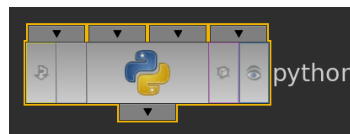

Figure 13. Python SOP node in Houdini network view.

In addition to that, there are also numerous other ways to use Python in Houdini – create new UI elements with PySide, edit Digital Assets, write SOHO scripts and many others. For the most part, Python interpretation in Houdini is mainly targeting user interface adjustments and pipeline-related optimizations. Unlike VEX, Python in Houdini does not handle easily performance demanding operations. Overall, due to procedural nature of Houdini that makes the workflow non-destructive to a certain degree, Python is more sporadically used to automate a repetitive task or reuse a part of a process, in comparison with other CG software supporting Python.

3.3 The Foundry Nuke

Nuke is a powerful node-based visual effects compositing software made by The Foundry. It is widely used in film and television post-production, mainly for editing of video clips or image sequences. Among many other features, Nuke offers very robust node-based workflow, multichannel editing, stereoscopic workflow, particle system, chroma keying and the ability to extend the software via plug-ins. Even a brief list already reveals how much the program is capable of.

Nuke's functionality can be extended via external plug-ins written in Tcl or C++. As of version 5, the support of Python language was introduced. Taking into account the node-based approach, Python gives the powerful ability to create and manipulate nodes in Nuke. [30] The following list is an overview of other Python capabilities within Nuke:

- Animation, creation of animation curves
- Geometry manipulation with geometry related operations
- Expressions
- GUI customization and creation new interfaces using PySide libraries, icon menus and modification of existing ones
- Pipeline optimizations
- Editing default node parameters
- Node graph manipulation

One way to enter and edit Python code in Nuke is through built-in Script Editor. It can be accessed by: Viewer → Windows → Script Editor. Script Editor allows to save and load ".py" files for later execution and editing.

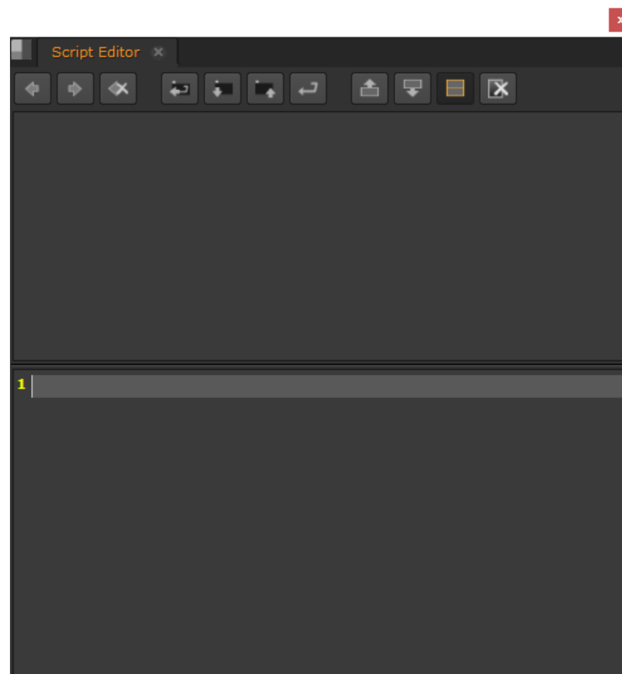

Figure 14. Script Editor window in Nuke 10.

Mainly, Python implementation in Nuke is not quite appropriate for handling heavy computational tasks, such as complex image or 3D manipulations, but is perfectly suitable for bridging the pipeline gap with other software.

3.4 Blender

Unlike many commercial software for visual effects and CG production, Blender is free and open-source, which apparently makes it explicitly attractive for smaller companies, start-ups and individual artists. Blender carries the entire production toolset meaning that a wide range of disciplines are supported within just a single application: polygonal modelling and sculpting, rigging, animation, fluid and particle simulation, rendering, camera and object tracking, compositing and video clip editing. Importantly, Blender has a large community of professional 3D artists and enthusiasts that are keen on further developing the software and are eager to contribute in a varied capacity. For the most part Blender is being developed by a small team of developers employed by Blender Foundation. Yet, a significant portion of advancements comes from the community of Blender users who take advantage of Blender's API, write new scripts and tools or extend the existing ones, generally seeking to improve their workflow. This became possible as a result of exceptionally tight Python language integration in Blender. Python scripting is an effective way to extend Blender's standard functionality. The majority of tasks done in Blender can benefit from Python scripting - rendering, export and import, animation, rigging and so on.

Other Python capabilities in Blender include:

1. Do manipulations that are related to scene, geometry, particles and other types of in-blender objects
2. Modify preferences, blender themes and appearance
3. Call different tools with modified settings
4. Manipulate GUI elements (tool menus and etc.)

The primary way to extend Blender's functionality is by writing a script. However, scripts are usually made for some simple tasks and intended for one-time execution. Whether a more sophisticated functionality that might require graphical interface interaction or multiple executions within a Blender session needed, it is generally recommended to wrap a script into add-on. [31] Python scripts can be executed in Blender by entering commands in the python console or via the built-in text editor.

1. Text Editor. Blender is supplied with a built-in text editor. It is an averagely basic editor that allows writing, saving, loading, editing and executing Python scripts. Besides this, there is a repository of templates available for those who want to take a closer view at some of the examples of how a certain Blender element is made. [32]

2. Python Console. Blender's Python console gives an overview of the entire Blender's Python API and provides with an auto-complete feature to check API for certain commands. That is the reason why Python Console is often used to explore new commands that later on can be copied to the whole script. [33]

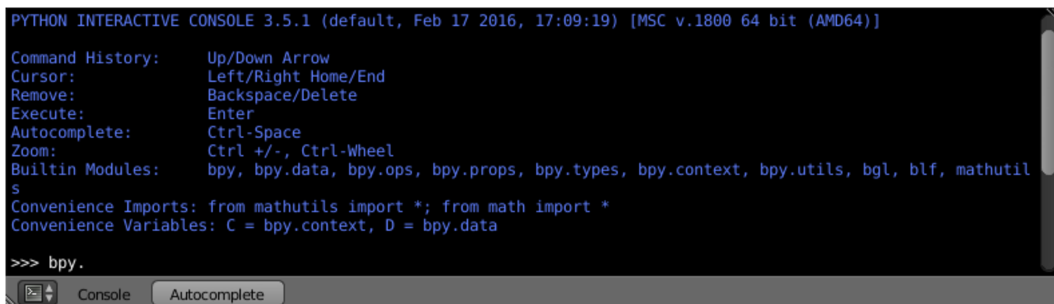

Figure 15. Blender Python Console.

3. Animation Drivers. Analogous to expressions in Maya or Houdini, Blender animation drivers can also be expressed in Python. By simply typing the expression in the parameter, Blender will automatically assign a driver to it.

The majority of digital content creation applications besides Blender have support for Python 2.x series. This is largely due to the fact that Python integration started when Python 3 was not yet released. If one software package would upgrade to Python 3, this can cause compatibility issues with other pipeline elements and other CG applications. After all, most of the challenges in VFX production can be handled with Python 2.x.

4 Development of Blender add-on Talvi Tools

4.1 Case Company Introduction

The empirical part of this study is covered in this chapter. The study was conducted within a Helsinki-based company, Talvi Digital. It is a post-production company that specializes in creating, editing and publishing video content for film, commercials, games and entertainment. It was founded in 2002 and now has more than 10 years of experience in media industry. Main clients include advertising agencies, production companies and game studios from Finland, Russia, Germany and Sweden.

4.2 Talvi Digital's Pipeline

For the feature film project awarded to the company, a team of artists and creative professionals was assembled from multiple disciplines. Taking into a consideration the relatively small size of the post-production team, a few people served as generalists. While a single-discipline specialist artist has a rather defined scope of work, a generalist artist serves as the multitalented professional who needs to possess an expertise of multiple visual effects specialties. Overall, an approximate graphic representation of Talvi Digital's post-production pipeline was carried as follows:

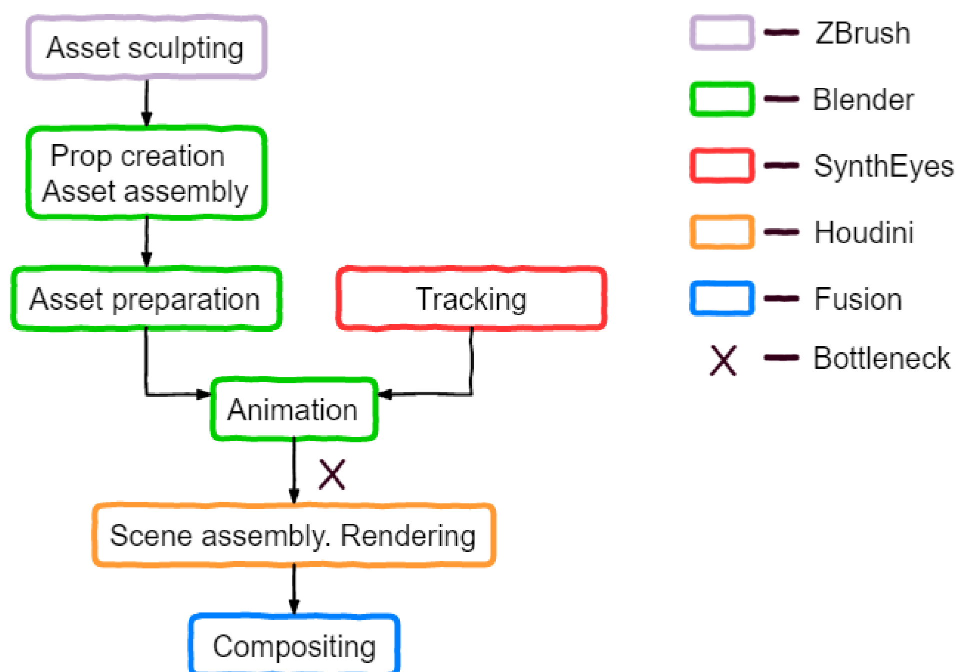

Figure 15. Talvi Digital's pipeline.

1. Asset sculpting – Character modelling and sculpting.
2. Prop creation. Asset assembly - Modelling and texturing of props and assets, character texturing.
3. Asset preparation – Character rigging for later use by animators.
4. Tracking – Camera and object tracking.
5. Animation – Character and camera animation.
6. Scene Assembly & Rendering – Assembling of scene layout, setting up materials and lights, look development, rendering.
7. Compositing – compositing of rendered elements, chroma keying.

According to figure 15, an immense part of work, especially during the first stages is completed with the use of Blender, whereas the latter, yet equally crucial part is done in Houdini. The severe incompatibility between Blender and Houdini project types would have resulted in the production bottleneck, which in turn would lengthen the turnaround time and weaken communication between stage 5 and stage 6. The "X" sign represents the location of bottleneck and suggests where the pipeline should be accelerated.

4.3 Initiation

As a rule of thumb, Alembic framework is commonly used to handle geometry and animation data send-offs across distinct software packages. SideFX Houdini has Alembic format support as of version 11. On the contrary, Blender did not have Alembic implementation at the time of this study. Therefore, other options of animation interchange between Blender and Houdini had to be explored. For static meshes the solution was found to pass the models via OBJ format. For animated meshes the OBJ models had to be accompanied with geometry point caches in MDD format. This method of animation data transfer was stress tested between animation department working in Blender and scene assembly department working in Houdini several times and then confirmed to be production approved.

As previously mentioned, one of the most challenging aspects of visual effects production is managing the sheer volume of data that has to pass across different departments. In case of Talvi's pipeline that was character and camera animation data compromising of several digital characters and cameras in the volume of dozen movie scenes with multiple shots in each scene.

Essentially, to manually export a character animation from Blender to Houdini with the chosen method, an animator would have to first export the geometry itself. This can be accomplished by selecting the desired object → File → Export → Wavefront (.obj) → Export OBJ. During these steps, a user would also have to specify the destination folder and select export parameters.

It is worthwhile to mention that OBJ export could be partly taken out of the equation, since animators work with instanced objects that are identical across all scenes. To put it differently, there is no need to export OBJ file every single time, but for testing purposes this feature was still implemented in the add-on. Similar to OBJ, MDD exporting procedure is done by selecting the object → File → Export → Lightwave Point Cache (.mdd) → Export MDD. In the course of export operation, animators would also have to define the output destination, filename and extra parameters, if applicable.

Having animators to go through each of these steps to just export a character would be rather tedious and unbelievably time-consuming. Bearing in mind that the movie features multiple CG characters, a few dozens of scenes and several shots in each scene. On top of that, from time to time animators would have to make revisions of animation, in case any adjustments are required. Each revision would provoke another export operation. Another important point concerns the human error component, which cannot be neglected for the reason that at the time of exporting a file from Blender a user is expected to manually enter the output destination. This can cause export files to proliferate extremely quickly as the time goes by. Under these circumstances, it became indubitable that export operations should be performed in automated fashion to remove the unnecessary burden from animation artists and allow them to spend more time on animation. Otherwise, the production schedule could simply spiral out of control.

4.4 Planning

In essence, it was proposed to develop a script in Python that would automate monotonous repetitive export operations. The principle behind the script is quite straightforward. As soon as the animation is ready for export, a user would execute the script, which in turn will complete export operations in the background.

Consequently, various possible ways to script that operation were investigated and researched. For a someone, with basic knowledge of Python language and very little experience

of scripting in Blender it was comparatively smooth to get the hang of Python scripting within Blender. Online tutorials and other educational sources provided with a good starting point and softened the learning curve. Along with that, Blender's documentation supplied with an extensive reference of commands and arguments.

4.5 User Interface

The following part of thesis will take a look at the user interface of the extension and break down how certain UI related decisions were made and what factors affected those decisions. Initially, the script was expected to run from the Text Editor and there was no user interface intended. Later, as the set of features became more extensive, it was clear that the script should be wrapped up in GUI interface.

As of version 2.70 Blender's standard user interface has undergone some changes and was slightly renovated. Prior to that the toolbar used to have one continuous row, which displayed all the tools grouped into panels. Although these panels were collapsible, a user still often had to scroll through the whole toolbar to find a desired tool. With the release of 2.70 vertical toolbar tabs were introduced. This novelty has improved overall user interface design and bettered the usability of the toolbar.

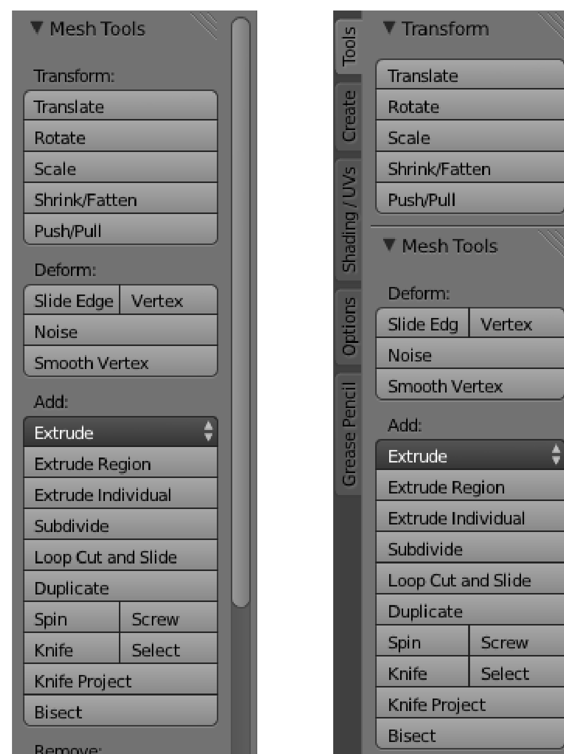

Figure 16. Comparison of Blender 2.69 toolbar (on the left) and Blender 2.70 toolbar (on the right).

More significantly, this has allowed to place add-ons in the dedicated tabs of the toolbar and create new ones as well. In order to avoid confusion with native Blender operators, it was decided to assign a new tab for the add-on. After some discussion, the very first UI mockup was designed as shown on figure 17 below:

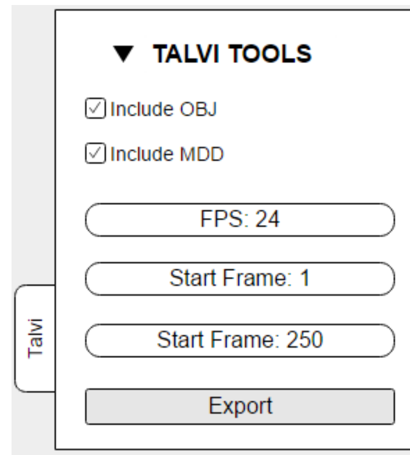

Figure 17. Prototype of the first Talvi Tools GUI.

By the time the basic functionality was implemented, the script was upgraded with a user interface to make it user-friendlier and provide animators with a slightly greater control over the export operations.

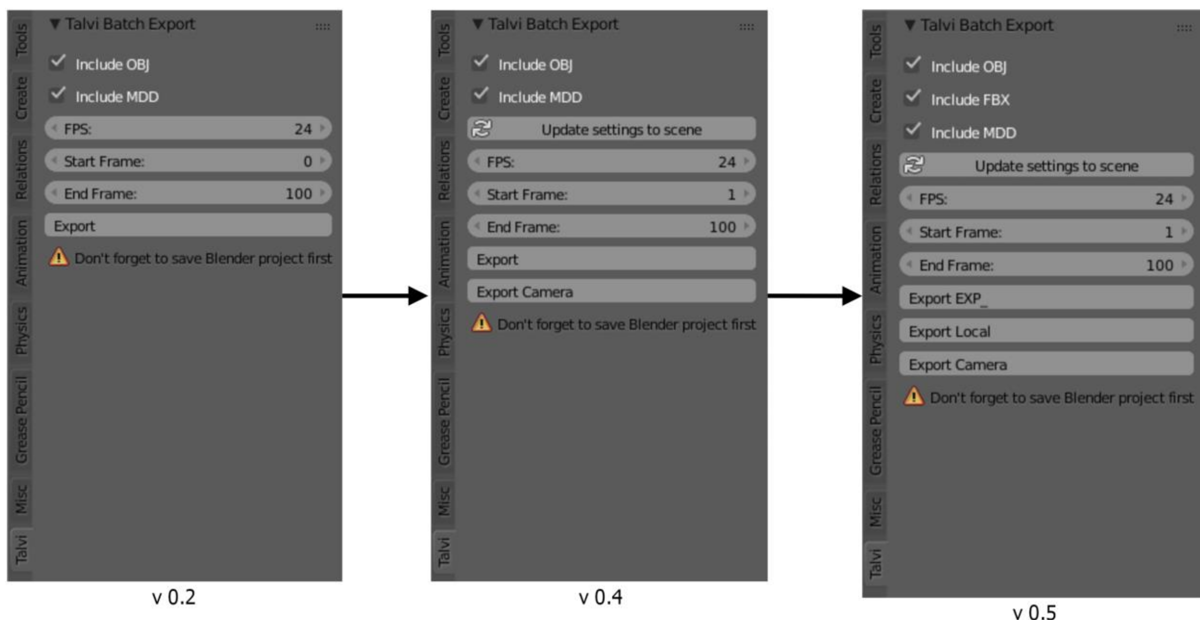

Figure 18. First phase of development of Talvi Tools GUI.

The introductory graphical release v 0.2 was developed with close reference to the prototype. Afterwards, over the course of development, the user interface had to accommodate the corresponding changes in the add-on itself. During the first phase of development several buttons were added. Some interface elements were complemented with Blender's icons. As an illustration, the text box beneath "Export Camera" button is marked with "!" icon to warn a user about the importance of saving a project file first.

Figure 19. Second phase of development of Talvi Tools GUI.

During the second phase of development, the extension was populated with a few more tools. In addition to export features, batch project creation tools were introduced. To adapt for these changes, each set of tools was grouped to an individual panel. With the approved release of Talvi Tools v 1.0, the user interface consisted of two collapsible panels, three checkboxes, seven sliders, eights buttons and one string input field. Certain buttons offer extra help in form of tooltips, which are shown when the mouse cursor is hovered over.

As can be seen the add-on's user interface is unambiguous and very intuitive. It contains well-known standard Blender's UI elements.

4.6 Development

This chapter will bring into focus the Talvi Tools extension from a development perspective and explore the major components on the back-end side of the plug-in. From the very beginning the development was heavily supervised by the lead animator who addressed the needs of animation department and translated them into feature requests. First and foremost, it must be noted that throughout the whole development cycle Blender's version 2.76 was used and thus the corresponding version of API was referenced.

In order to access the Blender's Python API, the bpy module had to be imported. Apart from bpy, the script also relies on the following modules: os, sys, subprocess, shutil and time. Moreover, for the purpose of building a user interface, the appropriate application modules Types and Property Definitions were supplied as listing 1 illustrates:

```
from bpy.types import Menu, Panel, UIList
from bpy.props import BoolProperty, FloatProperty,
IntProperty
```

Listing 1. Application modules import

In a nutshell, to build UI elements with a certain appearance, they had to be defined with a correct property type, as it is demonstrated in figure 20.

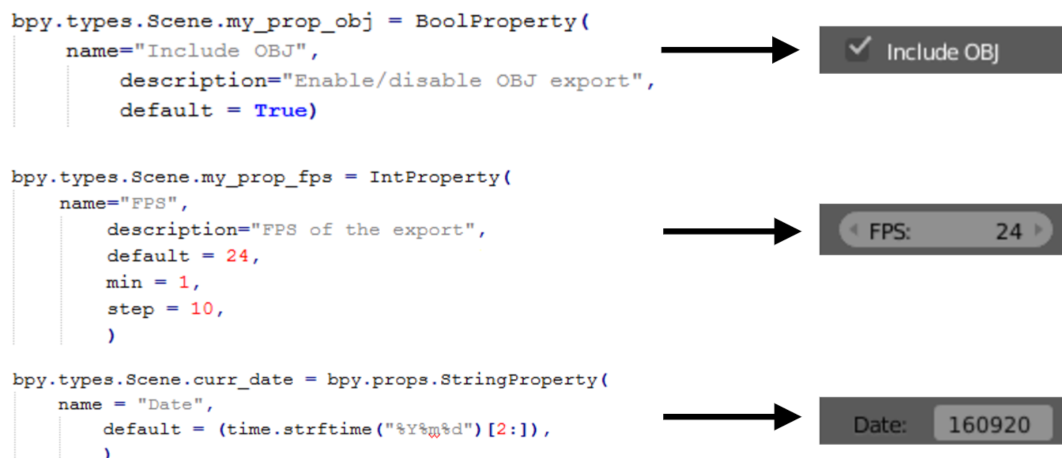

Figure 20. Property Appearance

These UI elements, such as checkboxes, sliders and string input fields allow a user to communicate with the script. Thereupon, the script gets the necessary parameters from those inputs as a result of the "communication".

Buttons, on the other hand, are the real executive UI elements. Once clicked, they will call operators, which in turn will run through the code and perform the operation they are assigned to. As stated earlier, the primary scope of the add-on is to accelerate the export of character and camera animation data. Sometimes animators work with several characters and cameras in one scene but do not necessarily wish to export them all. In that case, it was agreed to establish a special naming convention that would make distinctions between objects valid for export and the ones that are not. Objects approved for export are identified by the prefix "EXP_". As listing 2 demonstrates, the script is responsible for the selection of geometry objects that fall into that category.

```
for ob in scene.objects:
    if ob.type == 'MESH' and ob.name.startswith("EXP_"):
        ob.select = True
    else:
        ob.select = False
```

Listing 2. Object selection based on prefix

In the event that a user wishes to export objects with "Export EXP" button, the script looks for valid objects in the scene, makes a selection of them and loops through the selection with the command - *bpy.ops.export_shape.mdd.*

```
if bpy.context.scene.my_prop_mdd == True:
    path = bpy.path.abspath('//export_from_blender//MDD\\')
    if not os.path.exists(path):
        os.makedirs(path)
    bpy.ops.export_shape.mdd(filepath=str((path + ob.name + '.mdd')),
        check_existing=True,
        fps=bpy.context.scene.my_prop_fps,
        frame_start=bpy.context.scene.my_prop_start,
        frame_end=bpy.context.scene.my_prop_end)
```

Figure 21. MDD export inside of the "For" loop.

As figure 21 illustrates, the MDD export will proceed, if the corresponding checkbox is enabled. During the execution, the script will create a new folder in the project file location. With this intention, it is particularly important to save a project file beforehand. The operator *bpy.ops.export_shape.mdd* will be invoked with the parameters entered by a user in the corresponding UI inputs – FPS, Start Frame and End Frame. Another key point, for the operator

to work properly the default MDD export feature must be enabled in Blender's user preferences. In like manner, OBJ and FBX exports are carried out using export scene operators: *export_scene.obj* and *export_scene.fbx* respectively.

Likewise, the camera export is based on the same naming convention. However, the export procedure is realized with a different approach. Once the camera animation is approved for export, the script will "bake" the animation for every frame specified. This step is required to clear all the possible dependencies (e.g. rigs, parent objects, constraints) and get absolute camera object positions in 3D space.

```
def get_anim_info(obj):
    return "{}  {}  {}  {}  {}  {}  {}  {}  {}".format(
        scene.frame_current,
        round(obj.location.x, 4),
        round(obj.location.y, 4),
        round(obj.location.z, 4),
        round(obj.rotation_euler.x*57.2957795, 4),
        round(obj.rotation_euler.y*57.2957795, 4),
        round(obj.rotation_euler.z*57.2957795, 4),
        round(camera.lens, 4),
        round(camera.sensor_width, 4)
    )

for frame in range(scene.frame_start, scene.frame_end + 1):
    scene.frame_set(frame)
    anim_info.append(get_anim_info(ob))
    with open(path + ob.name + '.chan', "w") as f:
        f.write("\n".join(anim_info))
```

Figure 22. Camera export part.

According to figure 22, during the next step the script fetches camera translation, rotation, focal length and sensor size values for each frame and saves this data to a text file in ".chan" format. This file format is recognized by SideFX Houdini, which makes it possible to import camera animation on the next pipeline stage.

```
bl_info = {
    "name": "Talvi Tools",
    "description": "Accelerated export of objects in OBJ,
FBX and MDD formats",
    "author": "Vladislav Kazakov",
    "version": (1, 0),
    "blender": (2, 76, 0),
    "location": "View3D > Tools > Talvi",
    "category": "Import-Export"}
```

Listing 3. Add-on metadata

Last but not least, as shown in listing 3 the add-on is supplied with metadata, which is visible in Blender User Preferences panel. The metadata makes it easier for new users to locate the toolset among Blender's dense interface.

5 Practical implementation and Analysis

During the beta stages of development, the add-on was handed to the animation department and had immediately found its use in the daily workflow of animators. The add-on was complemented with a concise guide briefly describing the functionality of each button. Coupled with that, the lead animator gave more precise instructions to the rest of animation team.

To get the extension up and running it first has to be installed and enabled via Blender user preferences. Once enabled, it should appear in the separate tab of the toolbar as described in figure 23.

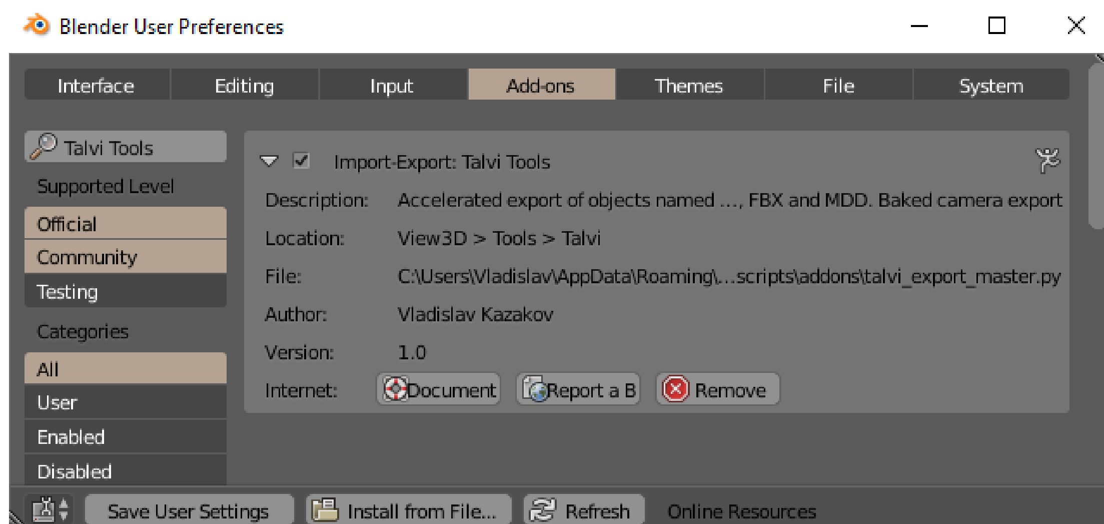

Figure 23. Enabling add-on in Blender User Preferences.

In the first place, Talvi Tools were oriented to speed up the boilerplate export processes. Later on, when batch project creation tool was added, it is believed to has helped foster the post-production even more.

Because the majority of Talvi Tools features are built upon native Blender operations, the development boiled down to finding a specific operator in Blender API library. From time to time Blender's Python Console was exploited to identify a specific command and paste it into script. The most daunting part was to implement camera export. The reason that this was so

comes down to the difference between camera object types in Blender architecture. The challenge was to find the common properties and link them together. Whenever any flaws occurred, animators submitted bug reports, which had to be resolved. Once in a while, blender stackexchange and stackoverflow platforms were asked for a helping hand and were looked up for other people solutions. Moreover, a great amount of support was provided by other post-production team members.

When considering the future development of the add-on, it should be noted that with the upcoming release 2.78 Blender is expected to have Alembic export and import support [34]. This positive change should definitely help Blender communicate better with other components of a pipeline. In this respect, Talvi Tools can be advanced with automated Alembic export feature. In effect, it should make the add-on more useful for studios that do animation in Blender and want to transfer it to another software.

6 Conclusion

Even today, many people in the screen industry might not fully understand how advantageous it can be to have an organized pipeline regardless of the scale of a workplace or its organizational structure. The purpose of this study was to raise awareness about visual effects methodologies and illustrate the beneficial impacts of a solid workflow management. All things considered, a pipeline is a very broad subject that touches many facets of a post-production company, not just the movie-making workforce, but also human resources, IT and finance departments are affected as well.

Furthermore, this research examines the main obstacles faced by visual effects studios during the post-production phase. Specifically, the extreme incompatibility between different content-creation suites used in various creative departments was analyzed. In order to cope with this problem, pipeline engineers often rely on Python, which in addition helps to deal with a slew of other technical challenges that need solving. While software vendors address these problems, a significant portion of them is handled by users themselves.

As a vivid example of that, the research is backed by the empirical part, which showcases how a subtle hack developed with a beginner knowledge of Python can evolve into a software extension with a comprehensive feature set. The final outcome of the practical part is the add-on for Blender, which helps facilitate the export of character and camera animations and can yield to considerable pipeline optimizations.

Upon completion of this study, a better insight of pipeline management was acquired with the development of theoretical framework needed to support this research. The practical part has contributed to the improvement of Python knowledge and scripting skills within a digital content creation software.

To conclude, Python integration can be found in almost every visual effects software and nearly every creative discipline can benefit from Python scripting to a certain degree. More and more individuals who are discontented with the limitations of commercial software embrace this programming language and develop own Python solutions that assist in their visual effects productions. Not only this, but a constant demand in specialist expertise indicates the undeniably important role of Python in visual effects industry.

References

1. Masters M. The Evolution of VFX [online]. Pluralsight.
URL: http://blog.digitaltutors.com/evolution-vfx-movies-60s-till-now/. Accessed 21 September 2016.

2. Bergeron H. The Power of Visual Effects [online]. Relghim; 18 August 2016.
URL: http://relghimthemovie.com/2016/08/18/the-power-of-visual-effects/. Accessed 21 September 2016.

3. Dunlop R. Production Pipeline Fundamentals for Film and Games. Burlington, MA: Focal Press; 2014.

4. Python in CG Pipeline. Free masterclass [online]. Vimeo.com; 3 March 2014.
URL: https://vimeo.com/88080700. Accessed 21 September 2016.

5. Stech K. Rhythm & Hues Looks to Finish ‘Seven Son’ [online]. The Wall Street Journal; 21 February 2013.
URL: http://blogs.wsj.com/bankruptcy/2013/02/21/rhythm-hues-looks-to-finish-seventh-son/. Accessed 21 September 2016.

6. Rhythm & Hues - A Framework for Global Visual Effects Production Pipelines - SIGGRAPH 2014 [online]. Vimeo.com; 9 January 2015.
URL: https://vimeo.com/116364653. Accessed 21 September 2016.

7. Hye Jean Chung. Global Visual Effects Pipelines: An Interview with Hannes Ricklefs [online]. Media Fields Journal; 2011.
URL: http://www.mediafieldsjournal.org/global-visual-effects/. Accessed 21 September 2016.

8. Venners B. The Making of Python A Conversation with Guido van Rossum, Part I [online]. Artima; 13 January 2003.
URL: http://www.artima.com/intv/pythonP.html. Accessed 21 September 2016.

9. Van Rossum G. A Brief Timeline of Python [online]. 20 January 2009.
URL: http://python-history.blogspot.fi/2009/01/brief-timeline-of-python.html. Accessed 21 September 2016.

10. Kuchling A, Zadka M. What's New in Python 2.0 [online]. Python Software Foundation.
URL: https://docs.python.org/3/whatsnew/2.0.html. Accessed 21 September 2016.

11. Kuchling A. What’s New in Python 2.2 [online]. Python Software Foundation.
URL: https://docs.python.org/3/whatsnew/2.2.html. Accessed 21 September 2016.

12. Wood S. A Brief History of Python [online]. Packt; 14 October 2015.
URL: https://www.packtpub.com/books/content/brief-history-python. Accessed 21 September 2016.

13. Dunn S. Dropbox File Sync Service [online]. PCWorld; 31 July 2008.
URL: http://www.pcworld.com/article/149058/dropbox.html. Accessed 21 September 2016.

14. Hoff T. 6 Lessons From Dropbox - One Million Files Saved Every 15 Minutes [online]. High Scalability; 14 March 2011.
URL: http://highscalability.com/blog/2011/3/14/6-lessons-from-dropbox-one-million-files-saved-every-15-minu.html. Accessed 21 September 2016.

15. Keen Gamer. Eve Online [online]. keengamer.com.
URL: http://www.keengamer.com/Game/eve-online/detail. Accessed 21 September 2016.

16. Thivierge E. Python: Bridging Technologies #VFX #Animation #Coding [online]. Youtube.com; 7 January 2015.
URL: https://www.youtube.com/watch?v=GerXSeJoJiU. Accessed 21 September 2016.

17. Lajoie D. Python in a VFX/Animation pipeline [online]. Youtube.com; 7 January 2015.
URL: https://www.youtube.com/watch?v=wXZ041KzcKg. Accessed 21 September 2016.

18. Fortenberry T. Industrial Light & Magic Runs on Python [online]. Python Software Foundation.
URL: https://www.python.org/about/success/ilm/. Accessed 21 September 2016.

19. Autodesk, Inc. Maya User Guide: What's New in Autodesk Maya 2016 Extension 2 [online]. Autodesk; 11 May 2016.
URL: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-9EB2C812-0755-422A-B472-FB4BD6ACC4E2-htm.html. Accessed 21 September 2016.

20. Autodesk, Inc. Maya User Guide: MEL Overview [online]. Autodesk; 11 May 2016.
URL: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-60178D44-9990-45B4-8B43-9429D54DF70E-htm.html. Accessed 21 September 2016.

21. Mechtley A, Trowbridge R. Maya Python for Games and Film: A Complete Reference for Maya Python and the Maya Python API. Boca Raton, USA: CRC Press; 2011.

22. Autodesk, Inc. Maya User Guide: Python in Maya [online]. Autodesk; 11 May 2016.
URL: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-C0F27A50-3DD6-454C-A4D1-9E3C44B3C990-htm.html. Accessed 21 September 2016.

23. Leong T. Interacting with Maya: A Flow Chart [online]. 16 December 2011.
URL: http://teknicalanimation.blogspot.fi/2011/12/interacting-with-maya-flow-chart.html. Accessed 21 September 2016.

24. Python vs MEL [online]. Vimeo.com; 2 July 2013.
URL: https://vimeo.com/69563564. Accessed 21 September 2016.

25. Autodesk, Inc. Maya User Guide: Using Python [online]. Autodesk; 11 May 2016.
URL: https://knowledge.autodesk.com/support/maya/learn-explore/caas/CloudHelp/cloudhelp/2016/ENU/Maya/files/GUID-55B63946-CDC9-42E5-9B6E-45EE45CFC7FC-htm.html. Accessed 21 September 2016.

26. Estela M. Houdini: Hscript, Vops, Vex (and python), which to use? [online]. Cgwiki; 2 July 2016.
URL: http://www.tokeru.com/cgwiki/index.php?title=Houdini#Hscript.2C_Vops.2C_Vex_.28and_python.29.2C_which_to_use.3F. Accessed 21 September 2016.

27. Side Effects Software Inc. Houdini 15.0 documentation: VEX [online]. sidefx.com
URL: https://www.sidefx.com/docs/houdini15.0/vex/. Accessed 21 September 2016.

28. Moore L. Python in Houdini for Technical Directors [masterclass]. Side Effects Software; 2007.

29. Fowler D. Python in Houdini [online]. Deborah R. Fowler website/Python Resources; 16 April 2015.
URL: http://www.deborahrfowler.com/PythonResources/PythonInHoudini.html. Accessed 21 September 2016. Accessed 21 September 2016.

30. The Foundry. Release Notes for Nuke 5.0v1 [release notes]. The Foundry; 2005.

31. Williamson J. Writing Your First Blender Script [online]. CG Cookie, Inc; 11 December 2014.
URL: https://cgcookiemarkets.com/2014/12/11/writing-first-blender-script/. Accessed 21 September 2016.

32. Blender Documentation Team. Blender 2.77 Manual: Text Editor [online].
URL: https://www.blender.org/manual/editors/text_editor.html. Accessed 21 September 2016.

33. Blender Documentation Team. Blender 2.77 Manual: Python Console [online].
URL: https://www.blender.org/manual/editors/python_console.html. Accessed 21 September 2016.

34. Thacker J. Check out the new features in Blender 2.78 [online]. CG Channel Inc; 1 September 2016.
URL: http://www.cgchannel.com/2016/09/check-out-the-new-features-in-blender-2-78/. Accessed 21 September 2016.

Appendixes

Appendix 1. Talvi Tools v 1.0 source code (Written in Python programming language).

```
bl_info = {
    "name": "Talvi Tools",
    "description": "Accelerated export of objects named with 'EXP_' in OBJ,
FBX and MDD. Baked camera export",
    "author": "Vladislav Kazakov",
    "version": (1, 0),
    "blender": (2, 76, 0),
    "location": "View3D > Tools > Talvi",
    "warning": "",
    "wiki_url": "http://wiki.blender.org/index.php/Extensions:2.5/Py/",
    "category": "Import-Export"}

import bpy, os, sys, subprocess, shutil, time
from bpy.types import Menu, Panel, UIList
from bpy.props import BoolProperty, FloatProperty, IntProperty

class View3dPanel():
    bl_space_type="VIEW_3D"
    bl_region_type="TOOLS"

class PanelTalvi_Export(View3dPanel, Panel):
    bl_category="Talvi"
    bl_label="Talvi Batch Export"
    def draw(self,context):
        layout=self.layout
        scene = context.scene
        layout.prop(scene, "my_prop_obj")
        layout.prop(scene, "my_prop_fbx")
        layout.prop(scene, "my_prop_mdd")
        row4 = layout.row()
        row4.operator(operator = "dh.upd_opt",text = "Update settings to
scene", icon = "FILE_REFRESH")
        layout.operator(operator = "dh.oft_opt",text = "Offset 10 frames",
icon = "FORWARD")
        row1 = layout.row()
        row1.prop(scene, "my_prop_fps")
        row2 = layout.row()
        row2.prop(scene, "my_prop_start")
        row3 = layout.row()
        row3.prop(scene, "my_prop_end")
        if bpy.context.scene.my_prop_mdd == False:
            row1.enabled = False
            row2.enabled = False
            row3.enabled = False
            row4.enabled = False
        layout.operator(operator = "dh.exp_opt",text = "Export EXP_")
        layout.operator(operator = "dh.exp_local",text = "Export Local Se-
lected")
        layout.operator(operator = "dh.exp_linked",text = "Export Linked")
        layout.operator(operator = "dh.cam_opt",text = "Export Camera")
        layout.label (text = "Don't forget to save Blender project first",
icon = "ERROR")

class PanelTalvi_Create(View3dPanel, Panel):
    bl_category="Talvi"
```

```
    bl_label="Talvi Project Create"
    def draw(self,context):
        layout=self.layout
        scene = context.scene
        layout.operator(operator = "dh.main_file",text = "Open Base File")
        layout.prop(scene, "scene_start")
        layout.prop(scene, "shot_start")
        layout.prop(scene, "scene_end")
        layout.prop(scene, "shot_end")
        layout.prop(scene, "curr_date")
        layout.operator(operator = "dh.prj_create",text = "Create Projects")

class UpdateOperator(bpy.types.Operator):
    bl_idname="dh.upd_opt"
    bl_label="Simple Operator"
    def execute(self,context):
        scene = bpy.context.scene
        if bpy.data.scenes["Scene"].my_prop_fps !=
bpy.data.scenes["Scene"].render.fps:
            bpy.data.scenes["Scene"].my_prop_fps =
bpy.data.scenes["Scene"].render.fps
        if bpy.data.scenes["Scene"].my_prop_start !=
bpy.data.scenes["Scene"].frame_start:
            bpy.data.scenes["Scene"].my_prop_start =
bpy.data.scenes["Scene"].frame_start
        if bpy.data.scenes["Scene"].my_prop_end !=
bpy.data.scenes["Scene"].frame_end:
            bpy.data.scenes["Scene"].my_prop_end =
bpy.data.scenes["Scene"].frame_end
        return{'FINISHED'}

class OffsetOperator(bpy.types.Operator):
    bl_idname="dh.oft_opt"
    bl_label="Simple Operator"
    def execute(self,context):
        scene = bpy.context.scene
        for ob in bpy.context.scene.objects:
            if ob.animation_data is not None:
                action = ob.animation_data.action
                if action is not None:
                    ob.select = True
                    bpy.context.scene.objects.active = ob
                    action.name = action.name + '.export'
                    track = ob.animation_data.nla_tracks.new()
                    track.strips.new(action.name, action.frame_range[0], ac-
tion)
                    bpy.context.object.anima-
tion_data.nla_tracks["NlaTrack"].strips[action.name].action_frame_start = -100
                    bpy.context.object.anima-
tion_data.nla_tracks["NlaTrack"].strips[action.name].action_frame_end = 800
                    bpy.context.object.anima-
tion_data.nla_tracks["NlaTrack"].strips[action.name].frame_start = -100
                    bpy.context.object.anima-
tion_data.nla_tracks["NlaTrack"].strips[action.name].frame_end = 800
                    bpy.context.object.anima-
tion_data.nla_tracks["NlaTrack"].strips[action.name].extrapolation = "NOTHING"
                    bpy.context.object.anima-
tion_data.nla_tracks["NlaTrack"].strips[action.name].frame_start = -90
                    ob.animation_data.action = None
        scene.frame_end = scene.frame_end + 20
        scene.my_prop_end = scene.my_prop_end + 20
        return{'FINISHED'}


class SimpleOperator(bpy.types.Operator):
```

```
    bl_idname="dh.exp_opt"
    bl_label="Simple Operator"
    bpy.types.Scene.my_prop_obj = BoolProperty(
        name="Include OBJ",
            description="Enable/disable OBJ export",
            default = True)
    bpy.types.Scene.my_prop_mdd = BoolProperty(
        name="Include MDD",
            description="Enable/disable MDD export",
            default = True)
    bpy.types.Scene.my_prop_fbx = BoolProperty(
        name="Include FBX",
            description="Enable/disable FBX export",
            default = True)
    bpy.types.Scene.my_prop_fps = IntProperty(
        name="FPS",
            description="FPS of the export",
            default = 24,
            min = 1,
            step = 10,
            )
    bpy.types.Scene.my_prop_start = IntProperty(
        name="Start Frame",
            description="Start frame of the export",
            default = 1,
            min = 0,
            step = 10,
            )
    bpy.types.Scene.my_prop_end = IntProperty(
        name="End Frame",
            description="Final frame of the export",
            default = 250,
            min = 1,
            step = 10,
            )
    def execute(self,context):
        scene = bpy.context.scene
        b = scene.objects
        a = bpy.data.objects
        c = list(set(a) - set(b))
        gen = (object for object in c if object.type == 'MESH' and ob-
ject.name.startswith("EXP_"))
        group_name = "8l60yDmQrcY6"
        if group_name in bpy.data.groups:
            group = bpy.data.groups[group_name]
        else:
            group = bpy.data.groups.new(group_name)
        for object in gen:
            scene.objects.link(object)
            if not object.name in group.objects:
                group.objects.link(object)
            print(object.name)
        for ob in scene.objects:
            if ob.type == 'MESH' and ob.name.startswith("EXP_"):
                ob.select = True
            else:
                ob.select = False
        for ob in bpy.context.selected_objects:
            bpy.ops.object.select_all(action='DESELECT')
            ob.select = True
            bpy.context.scene.objects.active = ob
            if bpy.context.scene.my_prop_obj == True:
                path = bpy.path.abspath('//export_from_blender//OBJ\\')
                if not os.path.exists(path):
                    os.makedirs(path)
```

```
                bpy.ops.export_scene.obj(filepath=str((path + ob.name +
'.obj')),
                                                use_selection=True)
            if bpy.context.scene.my_prop_mdd == True:
                path = bpy.path.abspath('//export_from_blender//MDD\\')
                if not os.path.exists(path):
                    os.makedirs(path)
                bpy.ops.export_shape.mdd(filepath=str((path + ob.name +
'.mdd')),
                        check_existing=True,
                        fps=bpy.context.scene.my_prop_fps,
                        frame_start=bpy.context.scene.my_prop_start,
                        frame_end=bpy.context.scene.my_prop_end)
            if bpy.context.scene.my_prop_fbx == True:
                path = bpy.path.abspath('//export_from_blender//FBX\\')
                if not os.path.exists(path):
                    os.makedirs(path)
                bpy.ops.export_scene.fbx(filepath=str((path + ob.name +
'.fbx')),
use_selection=True)
        ob.select = False
        for object in group.objects:
            object.select=True
            bpy.ops.object.delete(use_global=True)
        bpy.data.groups.remove(group)
        return{'FINISHED'}

class SimpleOperatorLocal(bpy.types.Operator):
    bl_idname="dh.exp_local"
    bl_label="Simple Operator"
    def execute(self,context):
        for object in bpy.context.selected_objects:
            if object.type == 'MESH':
                bpy.ops.object.select_all(action='DESELECT')
                object.select = True
                bpy.context.scene.objects.active = object
                if bpy.context.scene.my_prop_obj == True:
                    path = bpy.path.abspath('//export_from_blender//OBJ\\')
                    if not os.path.exists(path):
                        os.makedirs(path)
                    bpy.ops.export_scene.obj(filepath=str((path + object.name
+ '.obj')), use_selection=True)
                if bpy.context.scene.my_prop_mdd == True:
                    path = bpy.path.abspath('//export_from_blender//MDD\\')
                    if not os.path.exists(path):
                        os.makedirs(path)
                    bpy.ops.export_shape.mdd(filepath=str((path + object.name
+ '.mdd')),

            check_existing=True,

            fps=bpy.context.scene.my_prop_fps,

            frame_start=bpy.context.scene.my_prop_start,

            frame_end=bpy.context.scene.my_prop_end)
                if bpy.context.scene.my_prop_fbx == True:
                    path = bpy.path.abspath('//export_from_blender//FBX\\')
                    if not os.path.exists(path):
                        os.makedirs(path)
                    bpy.ops.export_scene.fbx(filepath=str((path + object.name
+ '.fbx')), use_selection=True)
                object.select = False
        return{'FINISHED'}
```

```
class SimpleOperatorLinked(bpy.types.Operator):
    bl_idname="dh.exp_linked"
    bl_label="Simple Operator"
    def execute(self,context):
        scene = bpy.context.scene
        b = scene.objects
        a = bpy.data.objects
        c = list(set(a) - set(b))
        for object in c:
            if object.type == 'MESH':
                scene.objects.link(object)
                bpy.ops.object.select_all(action='DESELECT')
                object.select = True
                bpy.context.scene.objects.active = object
                print(object.name)
                if bpy.context.scene.my_prop_obj == True:
                    path = bpy.path.abspath('//export_from_blender//OBJ\\')
                    if not os.path.exists(path):
                        os.makedirs(path)
                    bpy.ops.export_scene.obj(filepath=str((path + object.name
+ '.obj')), use_selection=True)
                if bpy.context.scene.my_prop_mdd == True:
                    path = bpy.path.abspath('//export_from_blender//MDD\\')
                    if not os.path.exists(path):
                        os.makedirs(path)
                    bpy.ops.export_shape.mdd(filepath=str((path + object.name
+ '.mdd')), check_existing=True,

            fps=bpy.context.scene.my_prop_fps,

            frame_start=bpy.context.scene.my_prop_start,

            frame_end=bpy.context.scene.my_prop_end)
                if bpy.context.scene.my_prop_fbx == True:
                    path = bpy.path.abspath('//export_from_blender//FBX\\')
                    if not os.path.exists(path):
                        os.makedirs(path)
                    bpy.ops.export_scene.fbx(filepath=str((path + object.name
+ '.fbx')), use_selection=True)
                scene.objects.unlink(object)
                object.select = False
            else:
                object.select = False
        return{'FINISHED'}

class CameraOperator(bpy.types.Operator):
    bl_idname="dh.cam_opt"
    bl_label="Simple Operator"
    def execute(self,context):
        scene = bpy.context.scene
        camera = bpy.ops.object.camera
        for camera in scene.objects:
            camObj = bpy.context.active_object
            camObj.data.name = camObj.name
        for ob in scene.objects:
            if ob.type == 'CAMERA' and ob.name.startswith("EXP_"):
                ob.select = True
            else:
                ob.select = False
        for ob in bpy.context.selected_objects:
            bpy.ops.object.select_all(action='DESELECT')
            ob.select = True
            bpy.context.scene.objects.active = ob
            camera = bpy.ops.object.camera
            for camera in scene.objects:
```

```
                camObj = bpy.context.active_object
                camObj.data.name = camObj.name
            path = bpy.path.abspath('//export_from_blender//DAE\\')
            if not os.path.exists(path):
                 os.makedirs(path)
            anim_info = []
            camera = bpy.data.cameras[ob.name]
            for ob in bpy.context.selected_objects:
                bpy.ops.nla.bake(frame_start=scene.frame_start,

            frame_end=scene.frame_end,
                        step=1,
                        only_selected=True,
                        visual_keying=True,
                        clear_constraints=True,
                        clear_parents=True,
                        use_current_action=False,
                        bake_types={'OBJECT'})
                def get_anim_info(obj):
                    return "{}  {}  {}  {}  {}  {}  {}  {}  {}".for-
mat(scene.frame_current,

            round(obj.location.x, 4),
            round(obj.location.y, 4),
            round(obj.location.z, 4),
            round(obj.rotation_euler.x*57.2957795, 4),
            round(obj.rotation_euler.y*57.2957795, 4),
            round(obj.rotation_euler.z*57.2957795, 4),
            round(camera.lens, 4),
            round(camera.sensor_width, 4))
                for frame in range(scene.frame_start, scene.frame_end + 1):
                     scene.frame_set(frame)
                     anim_info.append(get_anim_info(ob))
                     with open(path + ob.name + '.chan', "w") as f:
                            f.write("\n".join(anim_info))
        return{'FINISHED'}

class Main_File(bpy.types.Operator):
    bl_idname="dh.main_file"
    bl_label="Simple Operator"
    def execute(self,context):
        bpy.ops.wm.open_main-
file(filepath="//ocean/work/4895_rolli/3d/anim/000_000_base.blend")
        return{'FINISHED'}

class Project_CreateOperator(bpy.types.Operator):
    bl_idname="dh.prj_create"
    bl_label="Simple Operator"
    bpy.types.Scene.scene_start = bpy.props.IntProperty(
        name = "Scene Start",
            default = 1,
            min = 1,
            )
    bpy.types.Scene.shot_start = bpy.props.IntProperty(
        name = "Shot Start",
            default = 1,
            min = 1,
            )
    bpy.types.Scene.scene_end = bpy.props.IntProperty(
        name = "Scene End",
            default = 1,
            min = 1,
            )
    bpy.types.Scene.shot_end = bpy.props.IntProperty(
        name = "Shot End",
```

```
            default = 1,
            min = 1,
            )
    bpy.types.Scene.curr_date = bpy.props.StringProperty(
        name = "Date",
            default = (time.strftime("%Y%m%d")[2:]),
            )
    def execute(self,context):
        scene = bpy.context.scene
        a = scene.scene_start
        b = scene.scene_end
        c = scene.shot_start
        d = scene.shot_end
        e = scene.curr_date
        s = 'scene_'
        src = "//filepath_to_000_000_base.blend"
        for x in range (a,b+1):
            path2 = os.path.join("//filepath_to_3d/anim/",s + str("%03d" % x))
            os.makedirs(path2, exist_ok=True)
            for y in range (c,d+1):
                path3 = "//filepath_to_syntheyes/" + str("%03d" % x) +'/ex-
port/' + 'for_blender/' + 'blendSetup_'+ str("%03d" % x) + '_' + str("%03d" %
y) + '.py'
                if os.path.exists(path3):
                    path4 = os.path.join("//filepath_to_3d/anim/" + s +
str("%03d" % x), str("%03d" % x) + '_' + str("%03d" % y))
                    os.makedirs(path4, exist_ok=True)
                    dst = path4 + '/' + str("%03d" % x) + '_' + str("%03d" %
y) + '_' + e + '.blend'
                    shutil.copyfile(src, dst)
                    ghj = ("\"%s\""%dst)
                    foo = ghj.replace('/','\\')
                    pathscr = "\\\\filepath_to_syntheyes\\" + str("%03d" % x)
+'/export/'  + 'for_blender/' + 'blendSetup_'+ str("%03d" % x) + '_' +
str("%03d" % y) + '.py'
                    qwe = ("\"%s\""%pathscr)
                    moo = qwe.replace('/','\\')
                    blender = "C:/Program Files/Blender Founda-
tion/Blender/blender.exe"
                    saver = "\\filepath_to_workfiles\batch_exp_mdd\ap-
pend_script.py"
                    print (moo)
                    subprocess.call([blender,'--background', foo, '--python',
moo], shell=True)
        return{'FINISHED'}


def register():
    bpy.utils.register_module(__name__)

def unregister():
    bpy.utils.unregister_module(__name__)


if __name__=='__main__':
    bpy.utils.register_module(__name__)
```