EFFECTS AND PIPELINE TOOLS FOR HOUDINI AND MAYA USING PYTHON

Masters Thesis

Mihail Temelkov

N.C.C.A. Bournemouth University

21 August 2009

Contents

Chapter 1

Introduction

In essence, my project is about creating tools for Houdini and Maya using Python. Python is one of the most popular programming languages in the CG industry. It is known for its clear syntax, low learning curve and comprehensive library of modules. It supports both functional and object-oriented programming and is popular for rapid application development, prototyping and tasks for which speed is not the most important factor. It allows quick improvements in efficiency, which can be highly valuable in deadline driven environments.

This project was a good opportunity to increase my knowledge of the Python API's available in Houdini and Maya. Also, it involved creating a user interface using PyQt, so this was another area where I improved my skills.

The project also addressed some frequently discussed problems. One of these problems is how to easily create a particle flow in Houdini. In Maya this problem is handled by the Create Curve Flow effect tool. At present Houdini does not have such a tool.

Another common task is transferring cameras between Houdini and Maya. During a recent project I encountered this problem and discovered that the available solutions involve many steps and are error-prone.

Finally, my goal was to create tools that can benefit anyone who uses Houdini or Maya.

Chapter 2

Previous Work

2.1 Houdini Tools

Beginning with Houdini 9 all shelf tools are written in Python. In addition, since 2007 Side Effects Software has encouraged Houdini users to adopt Python by releasing a specification on creating Python tools for Houdini, as well as two Masterclasses in the form of video lectures.

2.2 Houdini Exchange Tools

Still, there are just a few user-developed Python tools. There are two downloadable shelves available on Houdini Exchange which use Python: "python manipulate", which has three tools, and "Camera from Rhino", which has a single tool. The code for these shelf tools is embedded, so it is easy to analyze it. However, these tools are quite simple. Also, neither of these shelf tools have user interfaces, so they cannot be used to learn how to created custom user interfaces for Houdini.

2.3 Camera Transfer Tools

There are no widely-available tools, which automatically transfer cameras between Houdini and Maya. There is an online tutorial on SCAD's website, which describes the steps required to manually transfer a Maya camera to Houdini, but it requires running a MEL (Maya Embedded Language) script, manually creating nodes in Houdini and manually transferring camera properties such as aperture from Maya to Houdini. The process can be quite time-consuming and error-prone for users who are new to either application.

Chapter 3

Technical Background

One of the goals of my master's project was to be able to "port" an existing Maya effects tool to Houdini. A lot of users new to Houdini come from a Maya background and frequently get intimidated trying to master a paradigm, which is quite different from what they are used to. In view of that, it would be beneficial to recreate the look and feel of a Maya effect, while retaining the flexibility offered by Houdini. The *Create Curve Flow* effect in Maya was chosen as a suitable candidate to "port", because it has a wide variety of applications, while being relatively simple to implement in Houdini. In addition, its Maya user interface is straightforward and contains parameters, almost all of which have direct counterparts in Houdini – the notable exception being "goal weight".

Python was chosen as the language to be used to "migrate" the effects tool, because of its ease of use, portability and the fact that it is supported by both applications. Since version 9.5 Houdini had added really strong support for Python via the Houdini Object Model (HOM). HOM is a fully object-oriented implementation, which offers more flexibility than HScript. Maya too offers support for Python since version 8.5, even though its implementation is closer to being just a wrapper for MEL. Since the goal was to create a tool that can be distributed to the Houdini community, portability was very important. Python met that criterion as well – when a Python module is imported, the source code is compiled automatically. The compiled version can then be used on machines running different operating systems without any further need for have a copy of the source code available for each environment.

Since Houdini does not have a user interface toolkit, external toolkits were evaluated. The two most popular UI toolkits for Python - wxPython and PyQt, were considered. PyQt was chosen because of its excellent documentation and the existence of an intuitive visual designer.

PyQt does place certain limitations however. One such limitation is the fact that PyQt widgets do not support float values. Since the user interface of Maya's *Create Curve Flow* effects tool features several controls allowing float values, it was important to overcome this limitation.

Another limitation was imposed by PyQt's "signals and slots" mechanism, which requires that the data type of the value emitted by the signal (event) must be matched by the data type of the parameter passed to the slot (event handler). This meant that it would not be possible to link directly a text field (known as a 'line edit widget' in PyQt) and a horizontal slider. Again, it was important to preserve the look and feel of the Maya user interface, and a workaround was necessary.

In contrast, the goal of the camera transfer tools was not to match the look and feel of Maya's interface, but to make sure that camera data is transferred accurately between Houdini and Maya. At the same time, it was preferable to give users flexibility over the range of animation they could transfer.

Any time data is transferred between different applications variables like units and scene settings become important factors. For example, transferring camera aperture between Maya and Houdini requires a "conversion" – Maya's aperture data needs to be scaled by a factor of 25.4 to derive the aperture in Houdini. In the course of doing research for this conversion process, I came upon information that led me to believe that Maya does not handle vertical camera aperture accurately. The rationale is explained at the end of Appendix C, chapter 7: Importing a Houdini camera).

Still another kind of problem I encountered was due to the fact that it is difficult to determine which Houdini channels are animated using the Python API. Originally I had relied on methods, which return information about channel keyframes. Since this approach does not take into account channels driven by CHOPs, I tried using the method hou.Parm.isTimeDependent(). However, while testing in different versions of Houdini, I discovered that it did not behave correctly in older versions, and even in the later versions it would produce inconsistent results between keyframed and CHOPs-driven channels. As a result, I developed my own algorithm of determining whether a channel is animated or not. It led to some efficiency gains and made the data transfer process more robust.

Chapter 4

Solution, self-evaluation, future work

4.1 Particle tools

Note that this section contains a summary of the solutions I have developed. For the full versions of these solutions, including a number of screenshots, please refer to appendixes C, D, E and F, which are user guides for the various tools.

The first tool that was implemented was the particle Curve Flow tool for Houdini. The original goal was to have its interface look and behave like the interface of Maya's *Create Curve Flow* tool. The two problems posed by PyQt were solved by sub-classing class QLineEdit in a custom module called *mt_widgets* (please refer to Appendix B, Python Module List, for a listing of the module's methods, or to the included source code). The solution was not difficult to implement and I could have done it sooner – instead I had spent too much time trying to find a solution using PyQt's API.

After ensuring that the Houdini particle flow tool's user interface matches exactly the user interface of the original tool in Maya, the next goal was to match the resulting look. Almost all Maya particle parameters – emission rate, lifespan, etc., have direct counterparts in Houdini and it was relatively easy to match roughly the look of the curve flow effect. Matching the output of the two tools exactly became more challenging. While the Maya tool creates expressions to control the particle motion, the most intuitive way to create the effect in Houdini is using an Attractor POP, which is influenced by forces. Balancing these forces to create an acceptable result can be time-consuming, adding another constraint – to match exactly the Maya look, can be too burdensome. The decision was made to abandon the intention to match exactly the Maya look of the effect and instead to leverage Houdini's strengths – instead of going for a specific look to add extra controls which would allow the user to create a wide variety of effects.

In addition, due to Houdini's strong support for proceduralism, the restrictions imposed by the Maya tool do not need to exist in the Houdini variant. For example, in

Maya once a curve flow is created, the user cannot change the number of segments or the number of sub-segments. In Houdini, varying these parameters is preferable, because it allows interactive experimentation with the geometry along which the particles flow. (Please refer to Appendix D and Appendix E, section *Modifying an existing setup*).

The emphasis on flexibility led to the decision to split the original curve flow tool into two tools – a curve flow variant and a surface flow variant. Instead of using an Attractor POP, the surface flow setup uses a Creep SOP to transport the particles along the surface. This variant allowed achieving some elusive goals. For example, using an expression can ensure that the particles will traverse the surface in a precise amount of time. The surface flow variant also allowed closely approximating the effect of particle goal weight. Neither of these was easily achievable by the curve flow variant.

Creating two particle flow tools increased the research and development time, but it also forced code reuse and the segregating of frequently used functions into "utilities" modules (please refer to Appendix B for a listing of modules). Houdini-specific operations like displaying a helpcard in Houdini's help browser were separated in a module called *mt_hou_utils*, whereas application-agnostic functions were separated in a module called *mt_generic_utils*. The same approach was used later during the development of the camera transfer tools, when the *mt_maya_utils* module was added.

In terms of using Houdini's functionality to make the particle tools more flexible, various approaches were used – channel references, expressions, Switch SOPs controlled by spare parameters existing at the geometry object level, toggle controls activating/deactivating other controls.

In some cases adding flexibility required adding extra logic to clean up redundant nodes. For example, before a new flow setup is created, a new Null node is attached to the guide curve. That Null node is then referenced by the new particle flow network. Extra logic was added to find and delete "orphaned" Null nodes to avoid clutter and potential errors. The same logic was used later in one of the camera transfer tools to delete orphaned CHOP networks.

The main goal of the flow tools – allowing users to create a particle flow easily, was achieved. In addition to listening for the stand 'click' behavior, Houdini allows to

distinguish between clicking and ctrl-clicking the tool's icon. As a result, the tools can automate the flow setup process completely (when the tool's icon is ctrl-clicked).

The two variants work as expected, even if they don't create the same flow as Maya's tool would create given the same parameters. The curve variant is less predictable, as it is difficult to balance the forces, which affect the particles. My expectation is that compared to the surface flow variant it would take a greater number of additional flow shaping/controlling nodes to produce directable results.

Prior to deciding to create the camera transfer tools, I tried to create particle fluid versions of the curve/surface flow tools. I could not find a intuitive, easy to set up and relative quick solution. Once Houdini's particle fluids become faster and more directable, it would be useful to create fluid flow tools. A good benchmark is Realflow and it's D-spline tool.

4.2 Camera Transfer tools (please refer to Appendix C):

The goal of automating the camera transfer process was to make it easier, faster and more robust. Besides scripting all the steps described in the tutorial that inspired these tools, I added data validation for a variety of scenarios. The bulk of the time was spent testing various use cases on different Houdini builds. It became apparent that tests should first be done on the earliest Houdini builds, because they have the least Python support and certain methods don't behave as expected on the earlier builds. For example hou.Parm.isTimeDependent() produced different results in the different builds given the same data. This realization forced me to come up with my own algorithm of determining whether a channel is animated or not. It simply counts the number of unique values over a given frame range. If the number of unique values equals one, the channel can be considered static. The same algorithm was used to determine the number of animated channels in Maya, during the process of exporting a camera. The method used originally was to check if a channel is "connected", but it turned out that in some cases – e.g. animation driven by a motion path, that is not a guarantee that the channel's values will change over time. In summary, while trying to find an error-proof method of separating static from animated channels, I ended up making efficiency gains.

A number of other enhancements were made, like using a variety of message types – errors, warnings or purely informative messages.

Not all possible data validation use cases were tested, this is one are where there is room for improvement.