View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Texas A&M University

A PROCEDURAL INTERFACE WRAPPER FOR HOUDINI ENGINE IN AUTODESK MAYA

A Thesis

by

BENJAMIN ROBERT HOUSE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	André Thomas
Committee Members,	John Keyser
	Ergun Akleman
Head of Department,	Tim McLaughlin

May 2019

Major Subject: Visualization

Copyright 2019 Benjamin Robert House

ABSTRACT

Game development studios are facing an ever-growing pressure to deliver quality content in greater quantities, making the automation of as many tasks as possible an important aspect of modern video game development. This has led to the growing popularity of integrating procedural workflows such as those offered by SideFX Software's Houdini FX into the already established development pipelines. However, the current limitations of the Houdini Engine plugin for Autodesk Maya often require developers to take extra steps when creating tools to speed up development using Houdini. This hinders the workflow for developers, who have to design their Houdini Digital Asset (HDA) tools around the limitations of the Houdini Engine plugin. Furthermore, because of the implementation of the HDA's parameter display in Maya's Attribute Editor when using the Houdini Engine Plugin, artists can easily be overloaded with too much information which can in turn hinder the workflow of any artists who are using the HDA.

The limitations of an HDA used in the Houdini Engine Plugin in Maya as a tool that is intended to improve workflow can actually frustrate and confuse the user, ultimately causing more harm than good. This led to the creation of a methodology for developers to create an automated interface wrapper for SideFX Software's Houdini Engine plugin for Autodesk Maya that easily allows for the user interface designs that have been created natively in Houdini to be used in Maya. In this way, developers will be able to create interfaces with cognitive load and user experience in mind without having to adhere to the limitations of the Houdini Engine plugin or spend extra effort to create unique wrappers manually.

ACKNOWLEDGMENTS

I would like to thank my committee members André Thomas, John Keyser, and Ergun Akleman for their continued support throughout the process of completing this thesis. I especially want to extend thanks the development team at Bluepoint Games Inc. for being the inspiration for this prototype and for providing feedback and ideas for this methodology. I would also like the thank my friends and colleagues, as well as the faculty and staff, in the Viz Lab for supporting me during my long stay at Texas A&M University during both undergrad and grad school.

Lastly, I would like to thank André Thomas in particular for his monumental role in inspiring my career path, my love for Houdini, and all things procedural. Without his guidance, determination, and infectious enthusiasm I would never have ended up where I am today. I truly cannot thank him enough for his mentorship and all that he has done for me throughout my academic and professional careers.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professors André Thomas and Ergun Akleman of the Department of Visualization and Professor John Keyser of the Department of Computer Science.

The inspiration for this thesis originated from the artists and developers at Bluepoint Games Inc.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

There were no funds raised for this thesis.

TABLE OF CONTENTS

Page

ABSTRACT ii

ACKNOWLEDGMENTS iii

CONTRIBUTORS AND FUNDING SOURCES iv

TABLE OF CONTENTS v

LIST OF FIGURES vi

1. INTRODUCTION 1

2. RELATED WORK 4

   2.1 Cognitive Load Theory 4
   2.2 Intentional Design 5

3. HOUDINI ENGINE INTERFACES 6

   3.1 Houdini Interface Wrapper 6
   3.2 Supported Functionality 10

4. METHODOLOGY 12

   4.1 Retrieving Data from Houdini 12
   4.2 Wrapper Interface Functions 13
      4.2.1 Generating the User Interface 15
      4.2.2 Initializing the Parameters 16
      4.2.3 Updating the Parameters 18
   4.3 Supporting Functions 18
      4.3.1 Formatting Mel Commands 18
         4.3.1.1 Scaling Integer Sliders 19
   4.4 Expanded Functionality 20

5. FUTURE WORK 21

6. CONCLUSIONS 22

REFERENCES 23

LIST OF FIGURES

FIGURE Page

3.1 A complex Houdini Digital Asset interface created in Houdini that demonstrates the capabilities and interface layout that can be replicated by this methodology....... 8

3.2 This is the complex Houdini Digital Asset interface for the same HDA displayed in Figure 3.1 as it is displayed with the Houdini Engine in the Attribute Properties tab in Autodesk Maya. 9

3.3 This graphic displays most of the parameter types that are supported by the interface wrapper. 10

4.1 The hierarchy for the dictionary that should be created by the HDAExport.py script and stored in the JSON file that will be parsed by a separate script in order to generate the user interface in Autodesk Maya. 14

4.2 Above is a mock-up of the interface that can be generated by this wrapper. This includes additional features that are inherently supported in Houdini Engine such as Updating, Syncing, and Reloading an HDA, as well as potential expanded features such as the ability to toggle Auto-Update similarly to the feature in Houdini and the ability to reference and manipulate multiple instances of this HDA using a single interface. 19

1. INTRODUCTION

Game development studios face increased pressure every year to build bigger worlds populated by highly detailed content, and this pressure is "reaching a boiling point" [1]. The number of person-months that go into the development of a successful commercial game has been steadily increasing since the dawn of computer games, and game developers are struggling to meet deadlines while delivering the breadth of detailed content that is now expected of them [2]. Software packages such as Autodesk Maya and Autodesk 3DS Max have long been used by artists in the video game industry as prominent 3D modeling tools and they remain some of the leading tools for creating 3D video game art to this day [3]. In recent years, there has been a growing suite of 3D modeling software packages used in video game development to help reduce costs such as the open-source 3D modeling software Blender and the procedural systems of SideFX Software's Houdini FX.

The number of development hours necessary to create 3D game art of the desired quantity and quality, combined with the looming pressure of deadlines, leaves game development studios with limited options and has traditionally led to the hire of more 3D game artists. However, this approach ultimately increases costs and comes with other practical limitations such as the lack of physical space or desired talent [1]. Thus, procedurally generated game content, which Shaker et al. defines as the algorithmic creation of game content with limited or indirect user input, is steadily gaining the attention of game development studios [2]. A systematic approach to creating 3D art for video games requires a front-loaded time investment for creating an asset, but offers a great deal of flexibility, enabling faster iterations and the reduction of repetitive tasks [4]. While not always necessary for individual 3D game art assets, there is an increasing need for tools to help artists develop faster and perform monotonous tasks in a more automated fashion so they have more time to focus on creating higher quality content.

This rules-based approach to content creation can be daunting for new artists and is an alien process to many experienced artists who have already established careers in the game development

industry. Deep rooted programs such as Autodesk Maya are significantly more familiar to most artists and the developers at Autodesk have been in collaboration with game engine developers such as Unity Technologies for years [5]. Working within these established software packages, it is increasingly common for game developers to create custom tools (programs written for a special environment that automate the execution of tasks that could alternatively be executed one-by-one by a human operator) that are are intended to ease the burden of repetitive and monotonous tasks on game artists [6]. This can be done using external tools where the developers have created a software component that adds a specific feature to an existing computer program, often referred to as a plugin, such as InstaLOD's plugin for automatic optimization of 3D game assets inside Autodesk Maya [7]. Another approach is to supplement the software package using an integrated scripting language such as Python - which can be used to write tools in Autodesk Maya without utilizing any external software package. Additionally, developers can create a wrapper - a thin layer of code which translates a software's existing interface into a compatible interface - to make an existing interface easier to use or better integrated into their own proprietary pipeline.

SideFX Software's Houdini is an increasingly popular node-based software package used to procedurally create 3D content and special effects that has already established a long history in the film and animation industry. With a growing influence in the industry, SideFX Software has created Houdini Engine - a plugin for mainstream software packages that allows developers to present the procedural workflows available in Houdini to artists in a platform that is more familiar to them. The Houdini Engine plugin is also substantially cheaper than the standalone Houdini FX software, reducing the cost of procedural development while still offering the benefits.

Despite this integration however, the Houdini Engine plugin for Maya suffers from limitations in its interface design that inhibit the capabilities of the program outside of its native interface [8]. In addition to this, the robust user interface designs that can be created by developers in Houdini are not carried over to the plugin, which can deter artists from using the plugin due to an unorganized and confusing interface. To counteract this, developers often create wrappers for specific tools created for the Houdini Engine plugin for Maya so that the tool is presented with the intended

interface and functionality. In some cases, these tools are further developed to incorporate other plugins or proprietary tools in addition to the Houdini Engine operations. However, this process can be time consuming and thus subtracts from the benefits gained by the procedural Houdini Digital Asset tools (often referred to as HDAs) that Houdini Engine has to offer.

To address these issues, we created a prototype and methodology for a procedurally generated user interface wrapper for SideFX Software's Houdini Engine plugin for Autodesk Maya. The wrapper allows developers to easily use the user interface designs they have created natively in Houdini inside Maya and integrate additional functionality using QT and the Python programming language.

2. RELATED WORK

2.1 Cognitive Load Theory

The manner in which information is presented to a user is paramount to the user's ability to absorb and utilize that information. Due to the limitations of the Houdini Engine plugin in Autodesk Maya, developers lose a lot of control over the manner in which their tools are presented to the end user. In their preeminent book on the topic, *Cognitive Load Theory*, Sweller et al. discusses that learners must process instructional information in working memory and that the load imposed on working memory by that instructional information can be divided into categories depending on its function: 'intrinsic cognitive load' and 'extraneous cognitive load'. While intrinsic cognitive load is imposed by the basic structure and necessary information that the user needs, extraneous cognitive load is imposed by the presence of unnecessary or extraneous information. In other words, though some information may be situationally useful, depending on the relevant context, extra information and information presented in a manner that requires extra work for the user increases the overall cognitive load of the user unnecessarily [9]. Because Houdini is still relatively new to the game development industry and most artists are having to learn how to use the new tools that arise from its use, it is vital that developers are able to take this concept into consideration. The Houdini Engine plugin in Maya does not support properly hidden or disabled parameters, which can lead to significant extra extraneous cognitive load on the user's working memory.

Furthermore, in their paper, *Decreasing Cognitive Load for Learners: Strategy of Web-Based Foreign Language Learning*, Zhang also discusses cognitive load theory and states that a human's cognitive capacity in working memory is limited and if it overloads, learning will be hampered, so that a high level of cognitive load can affect the performance of learning and creates a series of negative impacts [10]. Similarly, it has been shown that when a user is familiar and comfortable with a navigation structure, they function with higher levels of satisfaction and efficiency. In a related study to extend understanding of the drivers of user satisfaction with website interfaces,

Jen-Hwa Hu et al. determined that cognitive load and performance outcomes fully mediated the effect of user familiarity on user satisfaction, and that cognitive load partially mediated the impact of navigation structure [11]. These results, which support the basic theories of cognitive load, confirm the importance of purposeful decision-making in user interface design and the significance of reducing the amount of extraneous information.

2.2 Intentional Design

Acosta adds further weight to the significance of intentional design when creating the interface for tools, stating that while a user interface plays an important role, it will not produce the relevant effects if not handled in conjunction with a designed user experience. They underscore the importance of the removal of extraneous content when creating user interfaces for tools, stating that all visible content of a design should contribute to the user's objective when using the tool. The interface should not contain irrelevant information that disrupts a consistent process, or poses misleading actions to the user [12]. Steve Krug summarizes this as the "first law of usability" using the phrase "Do not make me think!", wherein it is concluded that it is easier for users to make decisions when there are less options on screen, all available options are meaningful, and additional options are within easy reach [13].

The inability for a Houdini Digital Asset's parameters to be Hidden or Disabled when using the Houdini Engine plugin in Maya, as well as the exclusion of numerous other features, does not afford design choices that take the principles of Cognitive Load Theory or Acosta and Krug's conclusions into account.

3. HOUDINI ENGINE INTERFACES

This research sought to improve the functionality of the Houdini Engine plugin for Autodesk Maya by implementing a wrapper that can procedurally generate a user interface capable of emulating the functionality that is lost in the transition from Houdini to the Houdini Engine plugin. The focus was on the functionality of the Houdini Engine plugin for Autodesk Maya in particular, as it is well known, widely used by the game development community, and was readily available for our use. The resulting wrapper was developed in such a manner that it requires a minimal amount of direct input from a user and is able to be easily integrated with additional proprietary tools. It was created using the Python scripting language and PyQT - a Python binding for the Qt cross-platform C++ framework that is used to create the interfaces for both Houdini and Maya.

In this paper we will explain the methodology used to create an interface wrapper for the Houdini Engine plugin for Autodesk Maya that reflects the interface designed in Houdini. The final implementation of the wrapper uses the following simple steps:

- Add a Python script to an HDA
- Call a function in Maya using the path to the desired HDA as an argument

No code will be published from the results of this research, however script fragments and demonstrations of functionality are shared and discussed.

3.1 Houdini Interface Wrapper

When creating a procedural tool in Houdini, which is commonly referred to as a Houdini Digital Asset, or HDA, the developer is able to create fairly robust user interfaces by modifying an HDA's Type Properties. Here, the developer can create the interactive controls in the interface refereed to as parameters. This user interface design is a fundamental part of the usability and benefits of HDAs in a production setting.

It is well known that the success of any software is dependant on its usability[14] and additional studies on website interfaces have shown that the usability of an interface depends greatly on the

visual complexity and thus the impact on a user's cognitive load [15][16][17]. As depicted in Figure 3.1, the interface created for an HDA in Houdini can be quite complex, but is able to be organized, and unnecessary parameters are able to be hidden and/or disabled. This is juxtaposed by the fixed, vertical interface of Maya's Attribute Editor that the Houdini Engine plugin utilizes seen in Figure 3.2. The horizontal tab folder structure is not respected, and hidden and disabled parameter functionality is similarly not supported. Additionally, tooltips (text that appears as a pop-up when the mouse cursor is above the parameter label) that can be used to explain unique behaviors of each parameter are not supported in the Houdini Engine plugin. Finally, due to a limitation of Autodesk Maya, the float values of any Houdini Engine parameters are clamped to 3 decimal places, which can hinder the fidelity of certain tools where a high level of accuracy is required.

The interface of the Attribute Editor in Maya is not easily editable, so developers often resort to creating a separate, custom user interface of their own or subjugate a user to an interface that lacks almost any design intent. The original purpose for this wrapper was to ease the burden on both the developer creating the tool and on the end user. It should be noted that there was no intention to analyze the user interface design capabilities of Houdini FX or Autodesk Maya, but rather to ensure that the designs created in Houdini FX can be properly translated when using the Houdini Engine plugin for Autodesk Maya.

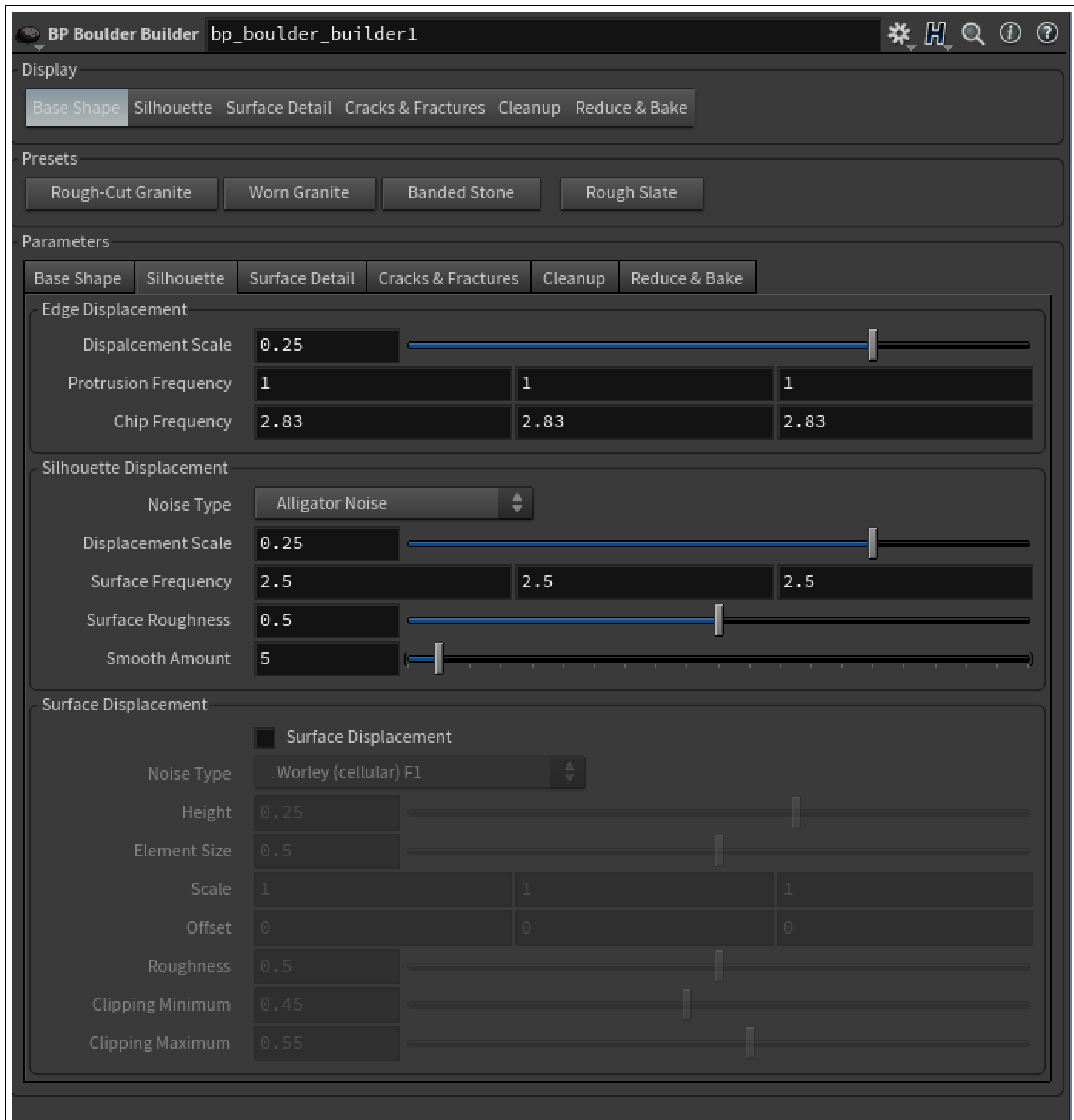

Figure 3.1: A complex Houdini Digital Asset interface created in Houdini that demonstrates the capabilities and interface layout that can be replicated by this methodology.

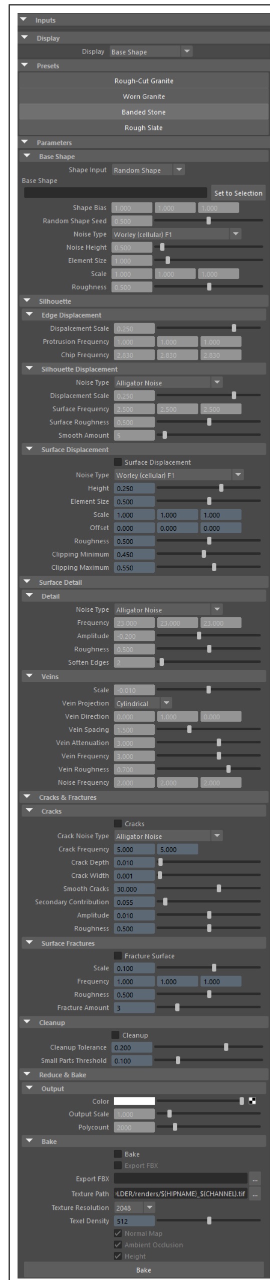

Figure 3.2: This is the complex Houdini Digital Asset interface for the same HDA displayed in Figure 3.1 as it is displayed with the Houdini Engine in the Attribute Properties tab in Autodesk Maya.

3.2 Supported Functionality

Houdini FX supports a wide array of parameter data types. For the purposes of this prototype, the following data types - most which are shown in Figure 3.3 - are supported:

- Floats, integers, & strings
- Float & integer tuples
- Toggles
- Operator paths
- Node input connections
- Single press buttons
- Single selection menus

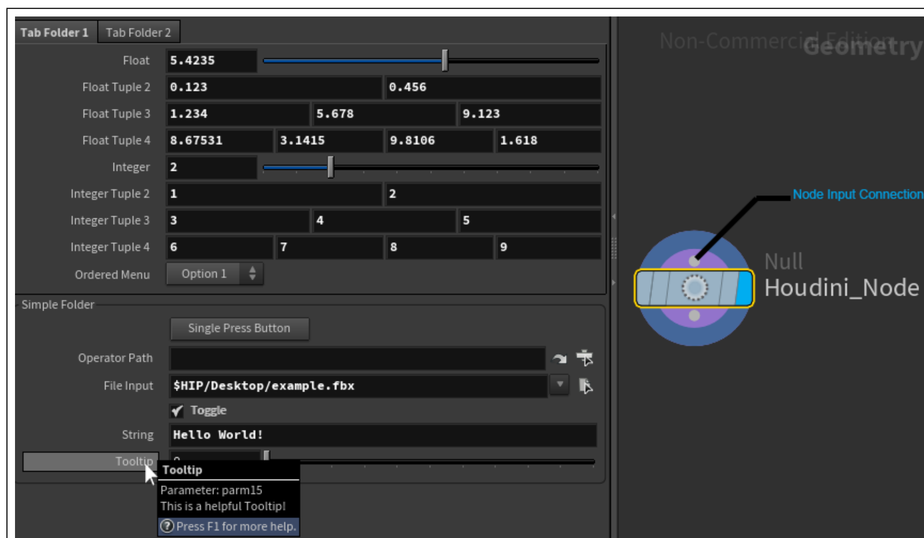

Figure 3.3: This graphic displays most of the parameter types that are supported by the interface wrapper.

Due to time constraints, the following parameter types are not supported in this prototype:

- Multiparms
- Data parms
- Separators
- Multi-press buttons & Menus
- Float and color ramps

The following interface functionalites will be supported by this prototype:

- Locked range for floats and integers
- 5 decimal places of accuracy for floats
- Dynamic "disable when"
- Dynamic "hide when"
- Simple and tab folders
- Nested folders
- Tooltips

Any parameter help text will appear as a tooltip when a user hovers overs the parameter label. Parameters are able to be locked or unlocked, and hidden or made visible based on the value(s) of other parameters.

4. METHODOLOGY

There were three distinct problems to solve in order to correctly load an HDA into Houdini, display an interface, and have it behave as expected. The first was how to collect and gather the information relevant to the HDA's user interface in a manner that is non-intrusive for the developer. The second was the means with which the interface itself will read the collected data and translate it into a packaged user interface that can easily be used for any set of parameter data exported out of Houdini. Thirdly, once generated, the interface must control the HDA in the expected manner and all of its conditional Hidden and Disabled states must be respected. In addition to these three fundamentals, the wrapper should support being easily integrated into an existing interface or expanded with additional functionalities based on personal preference or proprietary needs. These problems were addressed by 3 scripts: *HDAExport.py*, which handles collecting the interface information from Houdini, *importHDA.py*, which handles importing the HDA into Maya, and *HEWrapper.py*, which interprets the data, builds the interface, and ensures the proper behavior.

4.1 Retrieving Data from Houdini

In order to interpret the data relevant to the user interface in Houdini, a Python script - referred to as *HDAExport.py* - was created that can be added to an OnUpdate event for any given HDA. Once attached to an HDA in this way, the script will automatically run each time the HDA is modified.

Each parameter in Houdini is defined by a particular parmTemplate type and each inherits from the ParmTemplate parent class, which is a Python class used to store data that describes a parameter in an HDA's parameter interface and layout. Each parmTemplate type has a set of attributes unique to it in addition to those inherited by the parent class [8]. The *HDAExport.py* script iterates through the visible parameters in the HDA, and stores the ParmTemplate data for each as a unique Python ordered dictionary, which is an ordered collection of data values where each data value is tied to a unique key value that can be used to retrieve the associated data. This dictionary is in turn

stored into a larger ordered dictionary containing all visible parameters and their data, where the key for each parameter's data is the name of that parameter. The relevant data for each parameter's ParmTemplate component that was found to be useful for this interface wrapper was: *label*, *type*, *numComponents*, *conditionals*, *help*, *containingFolders*, *folderType*, *folderName*, *mennuLabels*, *minValue*, *maxValue*, *minIsStrict*, *maxIsStrict*, *defaultValue,* and *fileType*. If a parameter did not have a particular component, a value of None was assigned. While these values could have just been ignored, we ultimately decided to store the value as None, for consistency in the output. Ordered dictionaries were used to improve human readability and understanding should an event arise where this data needs to be modified manually.

This Python script then generates a JSON file in the same directory as the HDA file which the Python dictionary is written out to. To ensure that this system is easily iterable, the JSON file is overwritten each time the HDA is updated if it already exists. Once the Python script has been added to the OnUpdate event in the Scripts section of the HDA's Type Properties, the user will not need to take any further action to create the data necessary to generate the user interface in Autodesk Maya. This JSON file will later be processed by a separate group of Python functions in Maya to create the final interface. JSON was used for this purpose because it is a standard data interchange format that is regularly used throughout the industry.

The ordered dictionary output to the JSON file is formatted using the hierarchy shown in Figure 4.1, and will be referred to as the houData for a particular HDA. The houData contains the name of the HDA, the HDA's asset type, and an ordered dictionary of all the parameters that have been defined by the developer including any folder layouts and inherent attributes each may have. The parameters themselves are likewise defined as an ordered dictionary of their own with the parameter name being used as the dictionary key.

4.2 Wrapper Interface Functions

The wrapper functions as an alternative means of importing an HDA into Autodesk Maya using the Houdini Engine and is broken into two Python files: *importHDA.py* and *HEWrapper.py*. The former is the Python script whose main() function can take the file path to an HDA as an argument,

```
houData = {
    "hda": hda_name,
    "assetType": asset_type,
    "parms": {
        "width": {
            "label": string_value,
            "parmType": string_value,
            "numComponents": int_value,
            "conditionals": string_value,
            "help": string_value,
            "containingFolder": [
                string_value,
                ...
            ],
            "folderType": string_value,
            "folderNames": [
                string_value,
                ...
            ],
            "menuLabels": [
                string_value,
                ...
            ],
            "minValue": float_value,
            "maxValue": float_value,
            "minIsStrict": bool,
            "maxIsStrict": bool,
            "defaultValue": [
                value,
            ],
            "fileType": string_value
        }
        ...
    }
}
```

Figure 4.1: The hierarchy for the dictionary that should be created by the HDAExport.py script and stored in the JSON file that will be parsed by a separate script in order to generate the user interface in Autodesk Maya.

import that HDA into the Maya scene, and read in the associated houData file which will be used to generate the new interface. If no HDA path is given, the importHDA.main() function defaults to a file browser where a user can navigate to an HDA of their choosing. If no valid associated houData JSON file is found in the local directory, the function returns a warning. If successful, the function calls upon the library of functions in the *HEWrapper.py* file to interpret the houData and then create and operate the interface. Since this wrapper is utilizing the Houdini Engine plugin, it will also ensure that the plugin is loaded before any other operations take place.

The *HEWrapper.py* file is in essence a library of various Python functions that are used in a modular manner to interpret the JSON houData file and generate a working user interface inside Autodesk Maya. Since Autodesk Maya uses the Qt cross-platform C++ framework, the interface will be created using PyQT. The functions created for this library are separated into three categories: creating the graphical user interface or GUI, initializing the default values and parameter settings, and update functions to ensure that the information collected from the houData is properly

formatted for a given function.

4.2.1 Generating the User Interface

The first set of functions interpret the JSON houData file and iterate through each parameter key, creating a unique QT layout for each that is placed inside a designated QT layout. In QT, a widget refers to a visual user interface element such as a text box, scroll bar, label, button, etc. A layout refers to a container that arranges child widgets according to a horizontal or vertical designation. The *importHDA.py* contains a Python Window Class that uses an accompanying QT *.ui* file created in QT Designer, which is referred to as the *hda_template.ui* file. This file need only contain a single vertical QT layout, which the Window class uses as the parent layout for the QT interface generated from the houData. In this manner, a user can simply designate any empty QT layout of their choosing in any existing or custom QT interface as the container for the generated user interface. This allows the generated interface to be used alongside other external functions with minimal extra effort on the part of the user. More information on this is discussed in the *Expanded Functionality* section later on. Unique functions for each ParmTemplate type are used to create the necessary horizontal QT layout for each supported parameter type. The result of each function is referred to as a ParmWidget, or a horizontal layout containing all the QT widgets necessary for the parameter to both appear and perform in the same manner as its native Houdini counterpart.

The functions created for this purpose are: *folderWidget*, *buttonparmWidget*, *inputParmWidget*, *fileParmWidget*, *menuParmWidget*, *toggleparmWidget*, *tupleParmWidget*, *stringParmWidget*, and *sliderParmWidget*. Supporting functions were created to template repeating elements and ensure consistency with the existing Houdini interface. These are: *createLabel*, *createSlider*, and *createSpinbox*.

Each function has arguments for the parmName, houData, and container, where parmName is used as the key for the houData dictionary, and container specifies the parent widget layout, such as a folder, that should contain this ParmWidget. Folder ParmWidgets are created first in hierarchical order, followed by each of the remaining supported parameter types. Any unsupported parameter

types are ignored and a warning message is displayed to notify the user of such. The layout of each parameter type reflects the visual layout of that parameter type in Houdini. For example, most parameters should be constructed with a label on the left and a value on the right, single float and integer parameters should be accompanied by a slider, tuples should appear as adjacent horizontal values, toggles should have no left-hand label, etc. Since node input connections don't exist as interface parameters in Houdini, these parameters are generated in a folder above the designed parameters.

4.2.2 Initializing the Parameters

Once the interface has been generated, the ParmWidgets are stored in a dictionary that is used as the argument for the functions to update and modify the parameter interfaces with the parmName as the dictionary key. Each ParmWidget is comprised of a list containing their type, label, values, and any sliders, and will generally be formatted in the following manner:

$$[parameterType, labelWidget, valueWidget, sliderWidget]$$

Each ParmWidget function follows the following basic structure as depicted below:

```
def ParmWidget(self, parmName, label, container):
    # create label
    parmLabel = createLabel(self, parmName, label)

    # create the actual value component such as a double spinbox
    valueWidget = createSpinbox(self, parmName, isDouble)

    # create extra component such as slider, menu items, etc
    sliderWidget = createSlider(self, parmName)

    # create QT layout and add widgets to layout
    parmLayout =  QtWidgets.QHBoxLayout(self.parameterLayout)
    parmLayout.addWidget(parmLabel)
    parmLayout.addWidget(valueWidget)
    parmLayout.addWidget(sliderWidget)
    parmLayout.setObjectName(parmName + "_<ParmType>_pl")

    # add this ParmWidget layout to the container Layout
    if container != None:
        container.layout.addLayout(parmLayout)
```

The ParmWidget lists used in this wrapper have the following general structure with specific structures listed below. In some cases, None values are assigned where the label is unneeded,such as with the Buttons and Toggles, to maintain uniformity in the list order across all ParmWidgets. In the cases where objects or files are input by the user, such as with File, Object Merge, and Node Input parameters, an extra String value is appended to the end to store the real unicode value, while a human-readable string is displayed in the QLineEdit.

```
ParmType: ["type", label, value, extra]
Float: ["float", QLabel, QSpinBox, QSlider]
Int: ["int", QLabel, QSpinBox, QSlider]
String: ["string", QLabel, QLineEdit]
Menu: ["menu", QLabel, QComboBox]
Button: ["button", QLabel, QPushButton]
Tuple: ["tuple", QLabel, QSpinBox, ..., QSpinBox]
Toggle: ["toggle", QLabel, QCheckbox]
File: ["file", QLabel, QLineEdit, QButton, "<value>"]
Input: ["input", QLabel, QLineEdit, QButton, "<value>"]
Node Input: ["nodeInput", QLabel, QLineEdit, QButton,
            "<value>"]
```

Each ParmWidget is then used as an argument in an *initDefaults()* function, which pulls values from the associated dictionary stored in the houData JSON file. using these values, it then initializes a parameter to its default value, sets the min and max values if they are defined, and connects any associated values and inputs to their respective update function. The *initDefaults()* function essentially performs the following operations for each ParmWidget, with special operations that may be unique to each type:

- Assign default value
- Assign min & max values
- Connect interface to the associated update function

4.2.3 Updating the Parameters

Finally, there is a unique update function for each ParmWidget that ensures the input values are sent to the Houdini Engine interface and that any hidden or disabled states are respected by the interface. When any parameter value changes, any conditional arguments for that parameter will be evaluated and if the value results in any parameter changing disabled or hidden states, the wrapper will automatically sync the HDA in the Houdini Engine to ensure that the parameter state is properly reflected. Any hidden parameters will be unloaded from the Houdini Engine plugin and the newly visible parameters will be loaded into the plugin by re-syncing the HDA. Conditional statements are retrieved from the houData and translated into a Python string that can be evaluated as a logical argument.

It should be noted that re-syncing the HDA in Maya causes Houdini Engine to recook the HDA, which can slow down the HDA process. In most cases, this was found to be negligible, and when a slow down was noticeable it was an acceptable cost for the improved conditional states and usability offered by this functionality.

4.3 Supporting Functions

There were multiple supporting functions created to support the functions mentioned above, either to ensure proper formatting or consolidate repetitive code. The most notable are discussed below.

4.3.1 Formatting Mel Commands

The Houdini Engine parameters in Maya's Attribute Editor required a specially formatted Mel command to set values that may not be common sense to most, and is subject to change with future iterations of the Houdini Engine plugin, so a function to properly format input strings was created. Mimicking the "Set to Selection" functionality introduced by the Houdini Engine plugin, a Mel String Array must be constructed from the selected objects and then assigned as the parameter value in the Attribute Editor. Given a single string array of the selected objects as inputGeo, the Mel String Array was constructed as follows:

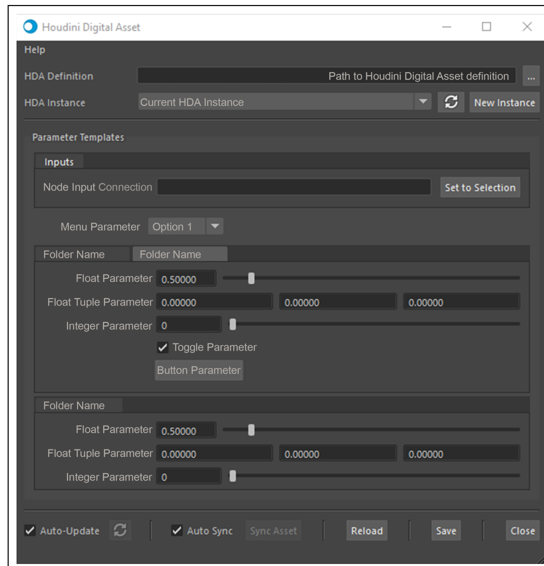

Figure 4.2: Above is a mock-up of the interface that can be generated by this wrapper. This includes additional features that are inherently supported in Houdini Engine such as Updating, Syncing, and Reloading an HDA, as well as potential expanded features such as the ability to toggle Auto-Update similarly to the feature in Houdini and the ability to reference and manipulate multiple instances of this HDA using a single interface.

```
# Resulting format: {"pCone1","pSphere1","pCube1"}
str_array="{{{}}}".format(",".join(["\"{}\"".format(s) for s in inputGeo]))
```

Once constructed, a Mel function is called to assign the value:

```
#Example of full string:
#houdiniEngine_setAssetInput hdaNode.input__node {"pCone1","pCube1","pSphere1"}
mel.eval("houdiniEngine_setAssetInput {} {}".format(hdaNode +
         "." + mayaAttributeName(parmWidget) + "__node", str_array))
```

4.3.1.1 Scaling Integer Sliders

Another notable function was a recreation of Houdini's *fit()* function, to rescale the QSlider used for float parameters, since QSliders do not support float values. This simple conversion expression is shown below:

```
newValue = (((oldValue-oldMin)*(newMax-newMin))/(oldMax-oldMin))+newMin
```

4.4 Expanded Functionality

For the purposes of this prototype, some expanded functionality was added as a means to more closely emulate the native functionality of an HDA in Houdini and to showcase the ease with which this methodology can be expanded and integrated with additional scripts. The notable expanded functionalities are:

- Implementation of Houdini's Manual & Auto Update
- Exposing the Sync Asset functionality
- Exposing the Reload HDA functionality
- Create new instances of an HDA
- Save the current HDA instance as static geometry

5. FUTURE WORK

Future development of this prototype wrapper would entail the implementation of the parameter types that were not initially supported. The most significant of these are Multi-parms and Ramps, which are commonly used parameters in Houdini. Additionally, the slow down that is induced by the implementation of the conditional Hide and Disable-When functionality should be more thoroughly examined as there may be a better alternative to re-syncing the HDA to ensure that the parameters load into Maya's Attribute Editor.

As new iterations of the Houdini Engine plugin are released by SideFX Software, this methodology may need to be altered.

6. CONCLUSIONS

Due to the ever-growing pressure on game development studios to deliver quality content in greater quantities, automating as many tasks as possible is an important aspect of modern video game development [1]. The current limitations of the Houdini Engine plugin for Autodesk Maya often require developers to take extra steps when creating tools to speed up development using Houdini. This hinders the workflow for developers, who have to design their HDA tools around the limitations of the Houdini Engine plugin. Furthermore, because of the implementation of the HDA's parameter display in Maya's Attribute Tab when using the Houdini Engine Plugin, artists can easily be overloaded with too much information which can hinder the workflow of any artists who are using the HDA. In short, the limitations of a HDA used in the Houdini Engine Plugin in Maya as a tool that is intended to improve workflow can actually frustrate and confuse the user and ultimately causing more harm than good [12].

This methodology and created prototype enables developers to create an automated interface wrapper for SideFX Software's Houdini Engine plugin for Autodesk Maya that easily enables the use of interface designs that have been created natively in Houdini. In this way, developers are able to create interfaces with cognitive load and user experience in mind without having to adhere to the limitations of the Houdini Engine plugin or spend extra effort to create unique wrappers manually. Additionally, it enables developers to easily integrate the Houdini Engine's capabilities inside Maya with other tools. By reducing the number of repetitive tasks that the developer has to perform to recreate HDA user interfaces that they have already designed in Houdini, this wrapper dramatically increases the productivity of a tool's developer. Its procedural nature also provides developers with a simple, automated processes to create iterations on these HDAs and their interfaces in Maya. This prototype methodology allows developers to remove extraneous tasks, reduce the cognitive load for themselves as well as for the end-user, and increases efficiency, ultimately promoting a more comfortable development environment [13][2].

REFERENCES

[1] K. Davidson, "Go procedural - a better way to make better games," *Gamasutra*, 2015.

[2] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games. by Noor Shaker, Julian Togelius, Mark J. Nelson.* Computational Synthesis and Creative Systems, Cham : Springer International Publishing : Imprint: Springer, 2016., 2016.

[3] P. Jamaa, "Preparing for a game dev career.," *Computer Graphics World*, vol. 30, no. 10, p. 40, 2007.

[4] M. R. Suarez, *A Procedural Approach to Computer-Aided Modeling in Nautical Archaeology.* 2016.

[5] U. Technologies, "Unity technologies collaborates with autodesk to strengthen link with autodesk 3ds max and maya.," *Business Wire (English)*, 2017.

[6] E. International, "Ecmascript 2019 language specification," 2019.

[7] I. GmbH, "Getting started with instalod for autodesk maya 2019," 2019.

[8] S. Software, "Houdini engine for maya," 2019. Available at www.sidefx.com/docs/maya.

[9] J. Sweller, P. Ayres, and S. Kalyuga, *Cognitive load theory. John Sweller, Paul Ayres, Slava Kalyuga.* Explorations in the learning sciences, instructional systems and performance technologies, New York : Springer, [2011], 2011.

[10] J. Zhang, "Decreasing cognitive load for learners: Strategy of web-based foreign language learning.," *International Education Studies*, vol. 6, no. 4, pp. 134 – 139, 2013.

[11] P. Jen-Hwa Hu, H. Han-fen, and F. Xiao, "Examining the mediating roles of cognitive load and performance outcomes in user satisfaction with a website: A field quasi-experiment.," *MIS Quarterly*, vol. 41, no. 3, pp. 975 – A11, 2017.

[12] K. Ramírez-Acosta, "Interface and user experience: important parameters for an effective design.," *Journal on March Technology*, no. suppl 1, p. 49, 2017.

[13] S. Krug, *Don't Make Me Think: A Common Sense Approach to the Web (2nd Edition)*. Thousand Oaks, CA, USA: New Riders Publishing, 2005.

[14] X. Fang and C. W. Holsapple, "An empirical study of web site navigation structures' impacts on web site usability," *Decision Support Systems*, vol. 43, no. 2, pp. 476 – 491, 2007. Emerging Issues in Collaborative Commerce.

[15] M. Chen, "Improving website structure through reducing information overload," *Decision Support Systems*, vol. 110, pp. 84 – 94, 2018.

[16] G. L. Geissler, G. M. Zinkhan, and R. T. Watson, "Web home page complexity and communication effectiveness," *Journal of the Association for Information Systems*, vol. 2, no. 1, 2001.

[17] G. L. Geissler, G. M. Zinkhan, and R. T. Watson, "The influence of home page complexity on consumer attention, attitudes, and purchase intent.," *Journal of Advertising*, vol. 35, no. 2, pp. 69 – 80, 2006.