OPEN RIGGING THROUGH XML: CHARACTER SETUP UTILIZING METADATA AND NODE BASED EDITING

A Thesis

by

LOGAN SCOTT KELLY

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

MASTER OF SCIENCE

Chair of Committee, Tim McLaughlin
Committee Members, John Keyser
Philip Galanter
Head of Department, Tim McLaughlin

May 2014

Major Subject: Visualization

Copyright Logan Scott Kelly 2014

ABSTRACT

Modular rigging systems exist to automate many of the labor intensive tasks associated with setting up character motion and control systems for 3D animation production. In this paper, a modular rigging system is described that encodes rig definitions in extensible markup language (XML). This method provides for version control along with the construct of a metadata node network facilitating easy propagation of changes to existing rigs. A node based interface is also provided for easily authoring rig definition files. The interface presented to the user is sufficiently simple that a user with minimal knowledge of rigging can construct a variety of complex rigs. By providing a node-based interface and rig definition format that utilizes version control, this method makes available capabilities that are currently not present in other open rigging systems.

TABLE OF CONTENTS

Page

ABSTRACT ii

TABLE OF CONTENTS iii

LIST OF FIGURES v

1 INTRODUCTION 1

2 RELATED WORK 3

2.1 Modular Rigging 3
2.2 Semantic Networks 3
2.3 Rig Definitions and Data Exchange Formats 5
2.4 Node Based Interfaces 6

3 METHODOLOGY 8

3.1 Rig Definition Files 8
3.2 Rig Instantiation and Updating 10
3.3 Rig Authoring 13

4 IMPLEMENTATION 16

4.1 Using XML for Rig Definitions 16
4.2 Using C++ and Maya API to Load, Update, and Remove Rigs 18
4.2.1 Constructing the Metadata Node Network 19
4.2.2 Commands for Loading, Updating, and Removing Rigs 20
4.3 Using PyQt for the Rig Manager and Node Editor 21
4.3.1 Rig Manager 22
4.3.2 Node Editor 26

5 DISCUSSION 32

6 FUTURE WORK 34

REFERENCES 36

Page

APPENDIX A 38

APPENDIX B 41

APPENDIX C 49

APPENDIX D 51

APPENDIX E 53

APPENDIX F 54

LIST OF FIGURES

FIGURE		Page
1	An example node graph of a character, with representations of the same graph in XML and JSON.	9
2	A possible Metadata Node Network present in a scene.	11
3	Concept design for the Rig Manager interface.	12
4	Concept design for the Node Editor.	15
5	An example MetaDataManagerNode connected to two MetaRootNodes in Maya.	20
6	The drop down menu provided for accessing the Rig Manager.	22
7	The Rig Manager interface for an empty scene.	23
8	The Rig Manager interface and "Rig Edit Mode" label on the viewport after clicking the "Create Rig" button.	24
9	The Rig Manager interface after a rig instance has been loaded into a scene.	24
10	The Rig Manager interface after a different version is selected from the Version drop down box.	26
11	The Node Editor interface is empty when the user begins creating a new rig definition.	27
12	A Rig Node has been added and selected in the Node Network Panel, and settings for the node are displayed in the Node Details Panel.	28
13	A Geometry Node attached to a Rig Node.	29
14	A Global Node attached to a Rig Node.	30
15	A Spine Node added to the node graph for a rig.	31
16	Attributes of a MetaDataManagerNode listed in Maya.	41
17	Attributes of a MetaRootNode listed in Maya.	42
18	Attributes of a MDGlobalNode listed in Maya.	43
19	Attributes of a MDHipNode listed in Maya.	45
20	Attributes of a MDSpineNode listed in Maya.	46

1 INTRODUCTION

Rigging is an essential step in the production pipeline of 3D character animation studios. It is the process that lies between modeling of character forms and animation of character performance. A rig is the architecture required for animating the motion of a character [O'Neill, 2008]. The process of designing and creating a rig is complex, time-consuming and labor-intensive.

In order to minimize both time and effort, procedural methods are used to automate portions of the rig construction and revision. Special types of rig scripts, called modular rigging systems, make the process easier by providing object-oriented components that define specific parts of a rig which can be connected together. However, many of these modular rigging systems do not provide support for revising rigs after they are already being used in animated productions. Additionally, often there is no persistent encoding of the rig definition outside of its use in a scene description file, which makes the utilization of version control tools for rigs difficult at best.

This paper presents a modular rigging system that utilizes rig definition files to support the process of designing rigs and enable rig updates while a rig is being used. Only the data relevant to a specific rig is contained in its own definition file. Therefore, managing the definition files with version control software is easily accomplished. The system also utilizes rig versions in order to determine if existing rigs are current or out of date. Every rig has a Metadata Node Network attached to each object that comprises the rig in the scene so that updates can be easily pushed from an associated definition file. A node-based interface for authoring rig definitions in a special editing mode also makes the rig creation process extremely simple.

This system is implemented as plug-in for the Autodesk Maya 3D software package. It utilizes the Maya C++ API in order to provide functions for loading and updating rigs from definition files, as well as defining metadata nodes used in Metadata Node Networks. XML (Extensible Markup Language) is used for encoding the rig definition files. Python and PyQt are used for creating the node-based interface for authoring rigs, as well as for the Rig Manager that lists all rigs and switches the user into and out of

rig editing mode.

The resulting system provides powerful functionality for managing and editing rigs that is not typically found in systems outside of a studio's proprietary tool set. The system's node-based interface also presents rig construction in a simple yet flexible way that only exposes the information pertinent to the user and hides complexity that typically makes rig construction difficult. Additionally, the use of open formats and non-proprietary software provides a framework that others should be able to expand upon easily.

2 RELATED WORK

2.1 Modular Rigging

In response to the increasing use of digital characters in films, games, and episodic media, a variety of modular rigging systems have been developed in recent years. One of the earliest publicized systems is Industrial Light & Magic's Block Party. A key feature of Block Party is that rig skeletons and controls can be shared across characters. Additionally, through a simple plug and socket mechanism Block Party allows mixing and matching of individual components of rigs to create new rigs. For example, the centaurs in "The Chronicles of Narnia" were a combination of the upper torso of a human rig and a horse rig excluding its head. In order to share joint positions and deformations across characters with similar body forms, a reference mesh associated with every component is deformed to match the shape of the new model. This deformed mesh is used as a volume guide in order to transfer the joint positions and deformations, resulting in production-ready deformations [Smith and White, 2006].

A similar system was developed by Blue Sky Studios to manage their procedural character rigs. However, where Block Party emphasized the use of volume guides as the foundation of their system, Blue Sky's system is based on the idea of separating the template for a character rig from the character itself. A character template is a network of nodes that defines all of a character's components' attributes, functionality, and relationships to other components. Rig templates can be developed in parallel with artists actually using the rig in its current state for animation. The rig templates would then be able to deploy updates to the current rigs used in shots automatically through a series of rig component script nodes (or rcSNs) that could build up or tear down a particular component. All rcSNs are attached to a top level rig component manager that controls the current state of all rcSNs in the scene[Malvarez et al., 2008].

2.2 Semantic Networks

An important concept to understand for the next modular rigging system that will be described is the idea of a Semantic Network. A Semantic Network is a graph that

represents knowledge in interconnected nodes and arcs. Semantic networks have existed long before computers, and have been used in philosophy, psychology, and linguistics. One of the earliest recorded semantic networks was created by the Greek philosopher Porphyry to comment on Aristotle's categories[Shapiro, 1992].

Semantic Networks have also been used in computer technologies for artificial intelligence, and also for traversal of the internet. In the case of internet traversal, search engines utilize semantic networks by harvesting metadata embedded in web pages in order to build its own Semantic Network to relate similar websites together. metadata includes information about what type of content is on a web page, and can be easily accessed by a web crawling agent that indexes websites. Relations between websites can then be accessed to return search results when a user performs a query with the search engine [Berners-Lee et al., 2001].

The video game company Bungie has used similar concepts of metadata and semantic traversal of node networks in their modular rigging system. This system is built on top of Maya, a commercial 3D software package. metadata is a term that means data about data, and semantic in this context means to describe what something is rather than what it is composed of through the utilization of metadata. For example, a character's arm component in a rig could be described as being composed of three joints, an IK solver, expression nodes, and 2D curves used as controller objects. But this wouldn't be a very useful description from a rigging perspective because it does not tell anyone that these objects are part of the character's arm. This description corresponds to an explicit description, and arm is the semantic description. The word arm corresponds to the metadata attached to the rig component [Hunt, 2009].

Bungie's rigging system allows for semantic traversal of the rig by constructing what they refer to as a Metadata Node Network. MetaNodes are additional nodes created in Maya's Dependency Graph that attach to each other, and also to important rig components. These are highly similar to Blue Sky's rcSNs, except that MetaNodes store additional metadata information to enable semantic traversal of the rig for scripting purposes.

2.3 Rig Definitions and Data Exchange Formats

While the previously described approaches to modular rigging have many powerful features, all of them are missing a rig definition file separate from the scene containing the rig. There are several reasons why this would be a desirable feature to have. First of all, creating a separate definition file that is agnostic to a particular software package allows the possibility of sharing the same rig across several software packages. This would ease the hand-off between different stages of the production pipeline where different software packages may be used. For example, animating in Maya but doing effects in Houdini. Another reason for creating a rig definition file is that it can describe a rig at a more abstracted level, containing only the data necessary to create the rig. This makes it easier to manage the rig data using version control systems, and also to update the rig definition when changes are necessary.

There are several options to choose from when deciding upon a particular format for a rig definition file. One option is to create a unique format just for use with rig definitions. While this option allows the greatest degree of freedom, it also requires the greatest amount of effort and makes adoption by others more difficult. Everyone using the definition format or expanding upon it would need to learn how it is structured from the ground up, and also write a parser from scratch in order to extract data from it into their own software or tools. A successful example of this approach is the RenderMan Interface Bytestream (RIB) file format for specifying RenderMan scene descriptions [Apodaca and Gritz, 1999].

Another option is to use a pre-existing data exchange format, and follow its formatting conventions. There are more varieties of formats than can possibly be listed here, but two of the most prominent are JSON and XML. JSON stands for JavaScript Object Notation and is a text-based open standard. The specification for JSON was written by Douglas Crockford, a computer programmer with extensive experience in JavaScript. It originated as a format to exchange data between JavaScript executed in web browsers and web servers. Despite its name, JSON is language-independent [Crockford, 2006]. XML stands for Extensible Markup Language and is a format traditionally used in web technologies for distributing arbitrarily structured documents

and data. The specification for XML was produced by the W3C, or World Wide Web Consortium which is the primary standards organization for the World Wide Web [Bray et al., 2008]. XML is also widely adopted, and has been used as the underlying format for many document types such as Microsoft Office files [Ngo, 2006]. While the choice between using JSON or XML is somewhat arbitrary, JSON tends to be more data-oriented and less verbose and XML is more documented-oriented and has more formatting relative to a similar amount of data. Both formats are human-readable, open standards with parsers commonly available in many languages.

2.4 Node Based Interfaces

As demonstrated by Side Effects Software's Houdini, node-based editing is a simple and powerful method for creating and managing 3D content. While Maya's Dependency Graph represents the scene as a interconnected node network, it is not intended as the primary interface for creating new content or even modifying connections between nodes.

In order to create a node interface that is editable within Maya, Tim Withers used PyQt for his thesis work at Texas A&M. PyQt is a Python binding of the Qt framework which is commonly used for developing graphical user interfaces. Qt is far more flexible and powerful than the default GUI framework provided by Maya, enabling the development of custom node-based interfaces which would not have previously been possible. Tim Withers's Node Editor interface as it was originally created is only intended for use with authoring particle systems. It replaces the default menu interface for particle systems, which many users of Maya see as cumbersome and non-intuitive [Withers, 2012].

This node-based interface can be re-purposed for other tasks where a node-based approach would be beneficial. Since Tim Wither's made his custom interface code open source, anyone can download the code and modify it freely for their own purposes. All that is required is to remove the underlying commands executed by the GUI widget elements in his interface and replace them with code that performs the desired actions. For example, the node-based interface can be used to replace the task of authoring rig

definition files in a text editor so that the user never needs to learn the rig definition format.

3 METHODOLOGY

The rigging system described in this thesis calls for three different subsystems in order to provide all of its essential features. The first subsystem is a rig definition file format that contains all of the data and metadata necessary to build a rig. The second subsystem is a mechanism for instantiating rigs based upon a rig definition file, and a method for propagating changes to pre-existing rigs when its corresponding rig definition file is updated. The third and final subsystem is an interface for authoring rig definition files.

3.1 Rig Definition Files

For the rig definition file format, a previously established data exchange format such as JSON or XML is preferred. Both formats have several useful properties that assist in implementing the rig definition file. These formats have been widely publicized and used, therefore many individuals will already be familiar with the basic syntax. They are implemented in a human-readable text-based format that makes debugging easier, as well as allowing the definitions to be managed with version control systems such as Git.

The hierarchical nested structure of these formats works well for representing the composition of a rig at an abstracted level. For any modular rigging system, the composition of a rig can be mapped by a graph of interconnected nodes. Each of these nodes represents a component, which is a discrete unit of a rig corresponding to a body form. An example component would be an arm, leg, or spine. While more complex components could be envisioned, for the purposes of this system an organic biped body form will be represented. Every connection between two components represents a relationship between the components. While there could be many different kinds of relationships between components, our system always assumes that a parent-child relationship is being established between the connected components. For the problem domain of an organic bipedal character, this will be sufficient for the rig representation.

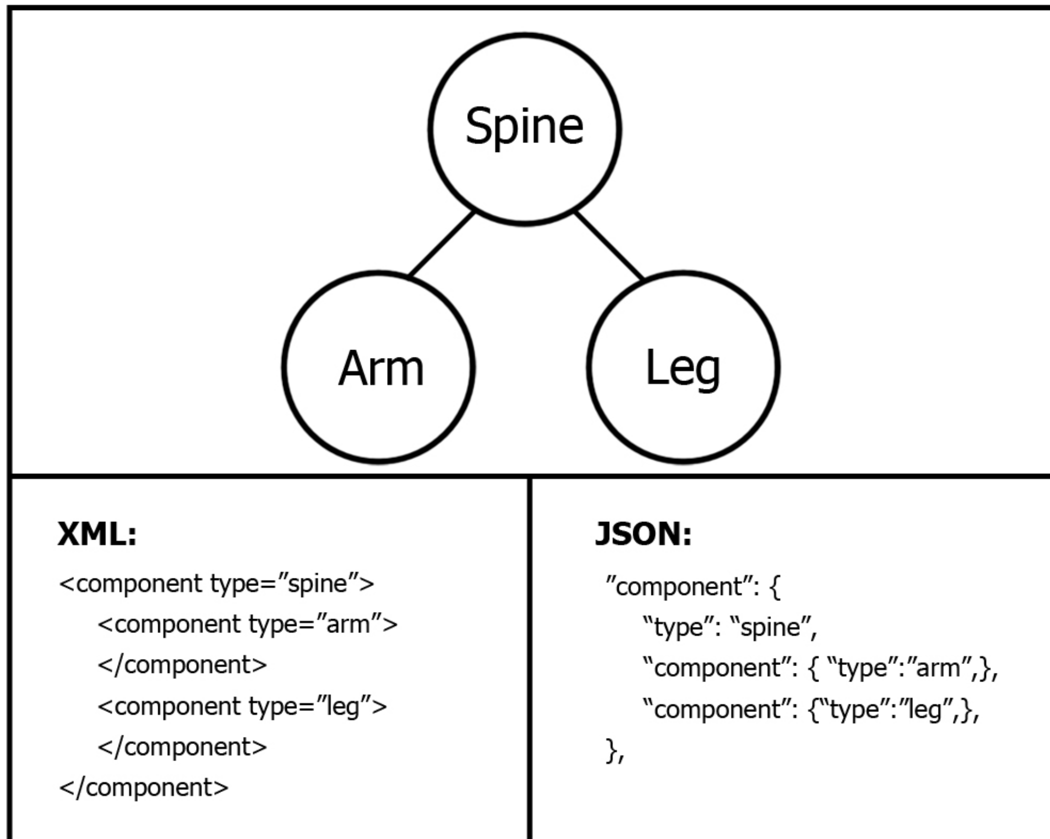


Figure 1: An example node graph of a character, with representations of the same graph in XML and JSON.

Graphical notations and linear notations can express equivalent information [Shapiro, 1992]. As a result, the node graph of the rig can be represented in our chosen data exchange format. Both JSON and XML have the capacity to represent objects with attached attributes that have other objects nested within them. This naturally leads to the idea that each object could represent a component in the graph, and each parent-child connection between two components can be represented implicitly by nesting an object inside of its parent object. Useful metadata associated with a given component can be contained inside of the attributes associated with its given object. An example of nesting objects inside of one another to represent parent-child connections can be seen in Figure 1.

There are a few scenarios that the previously defined rig definition system would have difficulty representing. One such scenario is to imagine a body form in which two limbs extend out from a central body and then re-converge into a third limb that is attached to both limbs. Since the third limb would need to be nested under both of the objects, this would be impossible to represent under the previously established conventions without adding some sort of support for instancing the object for the third limb under multiple objects. Because this would represent a non-bipedal form, the system established under this thesis disallows this scenario.

Another scenario that could not be represented with the present rig definition system is one in which components can have dynamic parenting relationships with other components. One could imagine a body type where a limb could be detached and re-attached elsewhere, or the parenting relationship with a component's parent could be temporarily inverted so that the parent could be made to follow the child. Since the structure of the rig definition file is underpinned by the assumption that parenting relationships are static, these scenarios could not be easily represented by the definition file without significant alterations to the format.

3.2 Rig Instantiation and Updating

The rig definition file contains all of the information that is needed to build a rig, but this file is not useful until instances of the rig can be used within the 3D scene of a given software package. The relationship between the rig definition file and the rig instances is analagous to genotypes and phenotypes in biology. Genotypes are the internally coded information responsible for biological traits, and the phenotypes are the outward physical manifestations of that information [Lewontin, 2011].

The *Load* and *Update* functions of the rigging system are responsible for creating all of the joints, controllers, constraints, models, and skin weighting to implement the rig definition as a usable rig instance. Each rig instance is attached to a Metadata Node Network. This Metadata Node Network is similar to Bungie's modular rigging system [Hunt, 2009]. The Metadata Node Network is essential for updating and removing rig instances as may be necessary in the future.

A good way to understand the Metadata Node Network is from a top down perspective. The top level metadata node, that all other nodes are ultimately connected to, is the rig manager node. Only one rig manager node exists in a given scene that implements the metadata system. Attached directly to the rig manager node are the metadata rig nodes corresponding to each rig instance in the scene. These metadata rig nodes contain a reference to the rig definition file used to define the given rig instance. This reference takes the form of a file path that the rig updating mechanism uses to open the rig definition and check for updates.

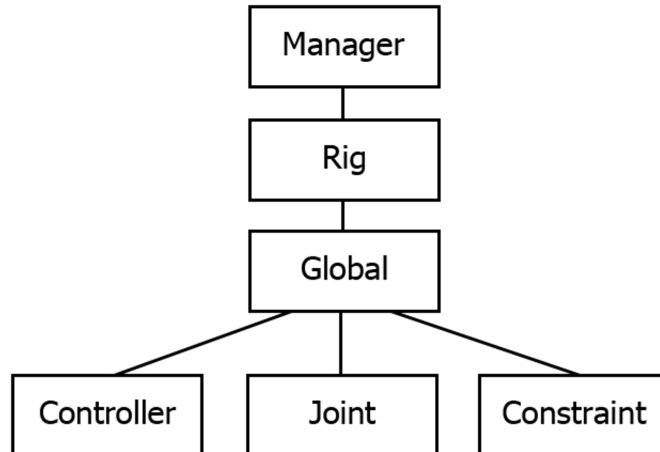


Figure 2: A possible Metadata Node Network present in a scene.

Attached beneath the metadata rig node is the metadata node for the root component of the rig. Typically this is a global component, which provides a controller that is able to move, rotate, and scale in global space the rest of the components of the rig which are attached to it. The metadata node for the root component can then have any number of metadata nodes for child components attached to it. These metadata

nodes may then also have metadata nodes for components attached to them, and so on in a recursive manner (see Figure 2). The connections of these nodes exactly mirror the nesting of objects in the rig definition file.

Each metadata node for a rig component has connections to every object in the scene which is part of that component. This allows the *Update* and *Remove* functions of the rigging system to find the correct objects to change for updates, or remove when the rig is removed.

The functionality for creating, updating, and removing rigs can be implemented either as commands through a plugin for the given 3D software package, or as procedures in a scripting language supported by the software. As long as the software package exposes the ability to script the creation of all objects necessary to create a rig, and to create persistent custom nodes for the metadata network, both a plugin or scripting approach should work.

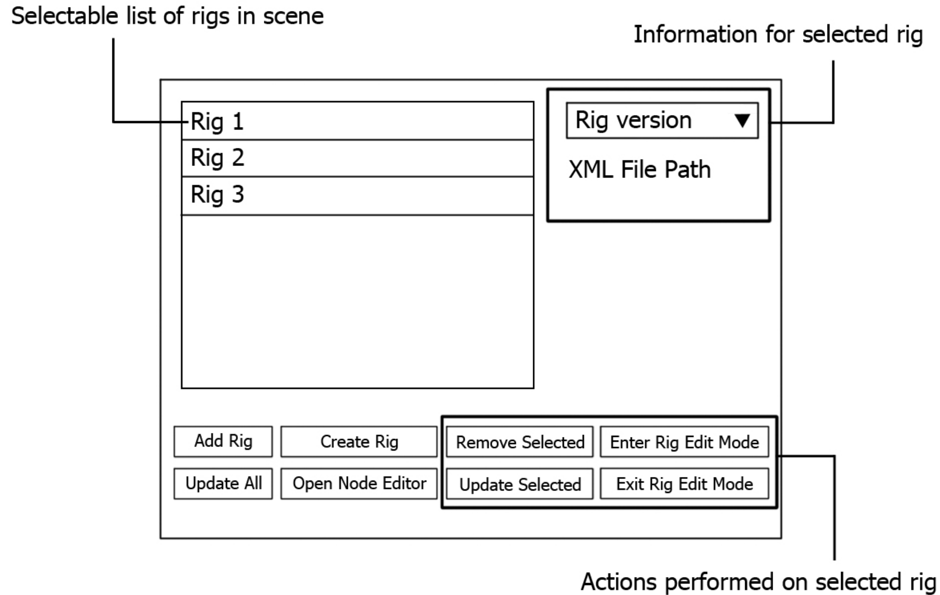


Figure 3: Concept design for the Rig Manager interface.

The *Load*, *Update*, and *Remove* functionality for rigs is made accessible to the user through the Rig Manager interface. The interface provides a list of all rigs present in the scene, and allows various operations to be performed on a selected rig. If no rig is selected, a new rig may be added to the scene based upon a rig definition file. A conceptual design for the Rig Manager is shown in Figure 3.

The Rig Manager allows users to edit an existing rig definition file by selecting a rig and entering *Rig Edit Mode*. While in *Rig Edit Mode*, an instance of the rig is displayed in its default configuration. This means that all controllers have zero local translation and rotation, local scale values of one, and no attached animation curves. Any changes to the positions of controllers and joints on this rig are saved into the positions in the rig definition file. During *Rig Edit Mode* the user can open the Node Editor, which shows a node-based representation of the rig. The Node Editor interface is discussed in more detail in the next section.

3.3 Rig Authoring

It is necessary for developers to understand the rig definition file format in order to extend the modular rigging system. However, most users of the system should not need to understand how to author new rig definition files in a text editor. Instead, a graphical user interface (GUI) is provided that is capable of generating new rig definition files based upon user interactions. A GUI lowers the initial difficulty for new users to begin creating rigs with the system. It also increases the speed with which new rig definitions can be authored. As an additional benefit, the GUI guarantees that the rig definition files produced are of valid structure, and disallows user interactions that would result in an invalid rig construction.

Node-based GUI systems have been used in software packages such as Houdini. Node-based interfaces have proven efficient for tools that need to rapidly generate many variations of a product and to easily perform changes earlier in the creation of the product. Both of these attributes are useful to have for a rigging system.

The rig definition file can be represented as a graph of connected nodes, making a node-based GUI fit well with the existing system. The *Update* functionality provided by

the rigging system guarantees that any operations performed in the node-based interface will be reflected correctly on the rig. A node-based interface is a different approach from many other software packages such as Autodesk Maya, which require that the user access operations from lists of menus that modify a scene graph without the user knowing exactly what connections are being made. Because menu-based systems require the user to memorize the order in which operations must be performed, errors can occur when the user does not remember the correct order[Withers, 2012].

The menu-based approach requires the user to maintain a mental model of the state of their scene before applying an operation to do it correctly. A node-based approach is made more simple by representing the state of a scene directly to the user as a graph. The node-based interface can disallow incorrect operations that could occur with the menu-based approach by providing connection points that allow only correct connections. Since insertions into the node graph are one of the primary operations that a user can perform, making changes to something that has already been constructed is easier than with the menu-based approach. When a new node is inserted, changes can be propagated through every node connected below the inserted node. To perform the same action with the menu-based approach, many previously applied operations need to be undone, and then new correct operations applied in order to properly modify the scene state. Modifying something that has already been made is often too time-consuming or complicated, and the user is better off starting from scratch.

A node-based interface for authoring a rig definition file can mirror the construction of the Metadata Node Network which supports rig updates as well as the hierarchical structure of objects in the definition file. Nodes can be created in a node network panel by selecting from a pre-defined list. Selected nodes can then be connected together to represent parenting relationships, disconnected and reconnected if parent ordering needs to be changed, or deleted if a node is no longer necessary. Attributes for the component represented by the node can be modified in a details panel with widgets for all the relevant settings. The general layout for the node-based interface, called the Node Editor, is represented in Figure 4.

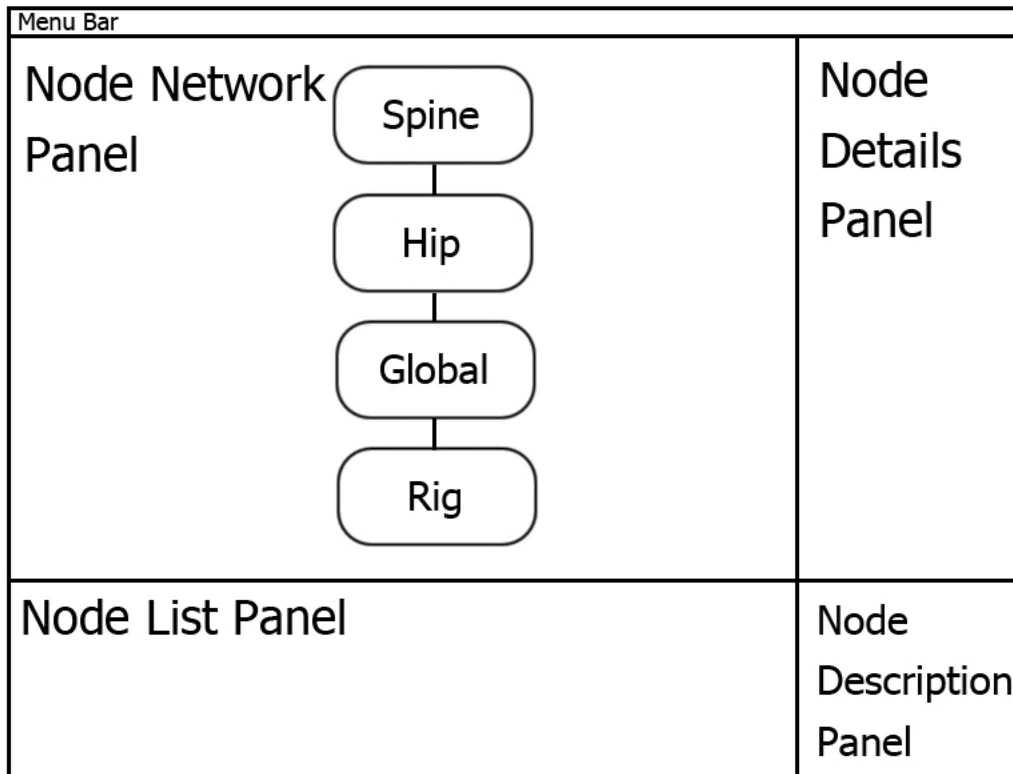


Figure 4: Concept design for the Node Editor.

Following all actions, the interface regenerates the rig definition file based upon the changes in the GUI widgets. The new rig definition file is used immediately to update the rig instance represented in the 3D scene during *Rig Edit Mode*.

4 IMPLEMENTATION

The modular rigging system described in this paper has been implemented in the 3D software package Autodesk Maya. The particular programming languages, software, and APIs used to implement this system are not the only choices that could be made. The rig definition format is designed so that as a possible future extension, rigs could be shared across varying software packages and applications. Autodesk Maya was chosen because of its proven capability and widespread use in the animation industry. For the rig definition format, XML was chosen in preference to JSON because of the abundance of XML parsers in Python and C++. The custom nodes for the Metadata Node Network as well as the *Load*, *Update*, and *Remove* functions for the rigs were all written as a Maya C++ API plugin. The node-based interface and Rig Manager interface were created using Python and PyQt. For reference, an overview of the architecture in a design diagram is included in Appendix .

4.1 Using XML for Rig Definitions

The data exchange format chosen for the rig definition files is XML. The XML format supports hierarchies of nested objects with attached attributes, and our particular rig definition format takes full advantage of this. A full listing of all attributes for every element in the rig definition file can be found in Appendix . The top level object, or *element* as it is typically referred to in XML, is the **Rig** element. An example **Rig** element looks like this:

```
<rig guiLocation="247.0,461.0" name="test" version="1.2">
```

The **Rig** element can have two types of child elements contained within it. The first type is a **Geo** element, which is used to contain data about a piece of geometry that is loaded and attached to the rig. A **Geo** element may look like this:

```
<geo file="./assets/TestSphere.ma" guiLocation="138.0,310.0" name="
    pSphere1" />
```

The other type of element a **Rig** element should contain is a **Component** element. This element is used to define every type of rig component used by the rigging system. The particular type of component and its characteristics are defined by the attributes on the **Component** element and additional child elements contained by the **Component** element, such as **Location** elements. Only one **Component** element should be contained by the **Rig** element, since logically there should be a top level component from which every other component should be ultimately attached. In almost every case, the top level **Component** element should be a global component.

A global component is typically the top level component of a rig because it contains a controller object with no joints or other objects. This controller object can be used for translating, rotating, and scaling the entire rig in world space. A global component element may look like this:

```
<component color="yellow" guiLocation="362.0,331.0" icon="FourArrowCircle
    " name="global" rigId="08495011515e4c1d926513662cc869dd" type="global"
     version="1.1">
```

A global component always has one **Location** element as a child element. This element identifies the position, rotation, and scale for a global component's controller. Here is an example:

```
<location localX="0.0" localY="0.0" localZ="0.0" rotateX="45.0" rotateY="
    0.0" rotateZ="0.0" scaleX="1.0" scaleY="1.0" scaleZ="1.0" />
```

Other **Component** elements can also be contained by the global component element. Typically these component elements are of different types, since it would not be especially useful to create a rig with nothing but global components. For a biped rig, the global component has a hip component as a child element. The hip component element is identical to the global component element, except that its type is set to "hip". The difference between a hip component and a global component is that a hip component contains a joint object, while a global component does not. The position of the joint object uses the data in the **Location** child element.

Another type of component element that is usually contained by a hip component element is a spine. A spine component represents a chain of joints that can be controlled with forward kinematics (FK), inverse kinematics that uses a spline curve (spline IK), or spline IK that will stretch to reach a target position (stretchy spline IK).

A **ShoulderControl** child element can be included within the spine component element if the "splineIK" or "stretchySplineIK" options are selected for the **KinematicType** attribute. This element is used to define the position and appearance of the controller that positions the target of the IK solver. This target is usually placed at the shoulders of the character, therefore the controller is referred to as the shoulder controller.

The spine component also contains multiple **Location** child elements. These **Location** elements are nested beneath one another to represent parenting relationships between joints and controllers. At least three **Location** elements must be contained by a spine component element to build a valid spine component. There is no upper limit on the number of joints and controllers a spine component can contain.

4.2 Using C++ and Maya API to Load, Update, and Remove Rigs

Maya allows its functionality to be extended with scripts in Maya Embedded Language (MEL) and Python. Compiled plugins written in C++ can also be created [Gould, 2003]. The functionality of the rigging system to load, update, and remove rigs based upon rig definition files is provided by a Maya plugin written in C++. C++ was chosen over MEL or Python for many reasons. MEL could not be used because it does not provide access to the Maya API. The Maya API is necessary for the rigging system because it provides the capability to create custom nodes in Maya's Dependency Graph, which are used to implement the Metadata Node Network. Without this network, updating and removing rigs would be impossible.

Python provides access to the Maya API as well, but C++ is used instead because updates to rigs are executed much more quickly. Since rig updates are executed constantly while in Rig Edit Mode, the execution speed of C++ is advantageous. While a Python implementation might be easier for new developers to use, such an

implementation is left as a possible future extension to the system.

4.2.1 Constructing the Metadata Node Network

Maya's Dependency Graph (DG) is the fundamental structure by which every scene construction operation in Maya is accomplished. Each DG node has a set of input connections that provide different types of data. Every DG node has a compute function that can perform operations on this data, and then produce results to various output connections. These output connections can then be connected to other DG nodes. A subset of the Dependency Graph is the Directed Acyclic Graph, or DAG. The DAG is different from the DG because no child node in the DAG can be its own parent. The DAG is used for objects that have a 3D position associated with them in the scene, such as geometry or groups. Because DAG nodes are a subset of DG nodes, DAG nodes can receive connections from DG nodes [Gould, 2003]. This is an essential property for the rigging system, because a Metadata Node Network can be constructed from DG nodes which are connected to DAG nodes comprising a rig (such as joints and controllers).

The C++ plugin provides custom nodes for use in the Dependency Graph. Each custom node defines a particular type of metadata node for use in the Metadata Node Network. Each type of node has a unique set of attributes that are attached to other nodes relevant to that type of metadata node. The compute functions of the nodes are not used, as the nodes are only used so that their connections can be traversed.

The first custom node provided by the plugin is the **MetadataManagerNode**. This type of node is the top level node in the Metadata Node Network, to which all other metadata nodes are ultimately attached (see Figure 5). The next custom node provided is the **MetaRootNode**. There is one **MetaRootNode** in the scene for each rig loaded from a definition file.

The plugin also provides metadata nodes for every type of component available in the system. One such metadata node is for global components, and is called an **MDGlobalNode**. Another metadata component node is provided for hip components, called a **MDHipNode**. A metadata component node is also available for spine components, and is called a **MDSpineNode**. For a full listing of all the attributes used

by each type of metadata node, see Appendix .

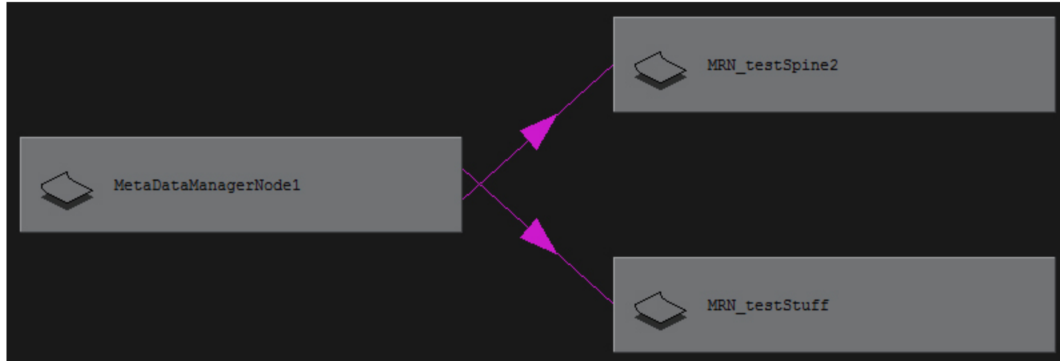


Figure 5: An example MetaDataManagerNode connected to two MetaRootNodes in Maya.

4.2.2 Commands for Loading, Updating, and Removing Rigs

The C++ plugin also provides a series of commands for performing tasks important for the rigging system to function. All of the commands utilize a set of C++ classes which parse rig definition XML files into guide classes, build up rigs based upon the guide information, update and remove rigs, and perform Metadata Node Network traversal. An overview of these classes can be found in Appendix .

A series of commands are provided by the plugin, and are callable by the user and other scripts as MEL commands. A complete listing of all the parameters available for each command can be found in Appendix .

The first command provided by the plugin is **updateMetadataManager**. This command is used to update either all of the rigs currently loaded into the scene, or individual rigs.

The **listMetaRoots** command returns a list of all the **MetaRootNodes** currently loaded into the scene. This command is used to populate the widget in the **Rig Manager** that lists all of the rigs in a scene.

The **loadRig** command is used to load a rig instance into the scene using a rig definition file. When the rig is built, it also constructs the Metadata Node Network associated with the rig and attaches all objects to it.

The **removeRig** command removes an individual rig instance from the scene. All nodes connected to the rig's Metadata Node Network are deleted, and the rig's Metadata Node Network is also deleted.

getMetaNodeConnection is used to perform traversal of the Metadata Node Network by following connections on a metadata node. It returns the name or DAG path of the node connected to a particular attribute on a metadata node.

getMetaChildById also performs traversal of the Metadata Node Network, but it only searches the **metaChildren** attribute of metadata nodes. From the list of child metadata nodes, it returns the name of the metadata node that matches a provided rigId.

4.3 Using PyQt for the Rig Manager and Node Editor

PyQt is a binding of the Qt application framework for Python. Beginning with Maya 2011, PyQt has been available for developers to use to create custom GUIs in Maya. While MEL provides a limited selection of widgets for use in custom GUIs, the flexibility of PyQt allows a greater range of custom GUIs to be created. Because of its versatility, Tim Withers chose to use PyQt for the development of his node-based GUI for authoring particle systems [Withers, 2012]. Since Tim Withers provided the source code for a node-based interface system that was proven to work, I chose to use the source code for his project as the basis for the Node Editor interface.

Since the Node Editor interface used PyQt, it was natural to also use PyQt to create the Rig Manager interface. The program Qt Designer was used for designing the Rig Manager interface. Qt Designer is a program that is available from the creators of Qt for authoring Qt interfaces in a WYSIWYG (what-you-see-is-what-you-get) editor [Digia, 2013]. A WYSIWYG editor allows a user to visually design an interface instead of writing code. Using Qt Designer, python scripts for creating an interface can be automatically generated, and the signals emitted by Qt widget elements can be

connected to slots in Python code. Signals are emitted by Qt widgets whenever the user interacts with a widget. Slots either can be signals on other Qt widgets or function callbacks in Python.

4.3.1 Rig Manager

The Rig Manager allows users of the rigging system to load, update, and remove rigs from their scene. It also provides access to **Rig Edit Mode** where the user can modify or create new rigs using the Node Editor. In order to access the Rig Manager, a drop down menu is added to the list of menus at the top of Maya's interface. This drop down menu (as seen in) is created by a MEL procedure that is called when the plugin is loaded. A screenshot of this interface can be seen in Figure 6.

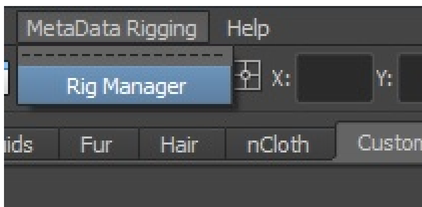


Figure 6: The drop down menu provided for accessing the Rig Manager.

The plugin also provides metadata nodes for every type of component available in the system. One such metadata node is for global components, and is called an **MDGlobalNode**. Another metadata component node is provided for hip components, called a **MDHipNode**. A metadata component node is also available for spine components, and is called a **MDSpineNode**. For a full listing of all the attributes used by each type of metadata node, see Appendix .

Selecting the Rig Manager from the menu displays the Rig Manager interface to the user. When the Rig Manager is displayed in a new scene with no rigs currently present, only two options are available to the user: add a rig to the scene from a definition file, or create a new rig in **Rig Edit Mode**. All other buttons are disabled until one of the

two actions are taken. The Rig Manager for an empty scene is shown in Figure 7.

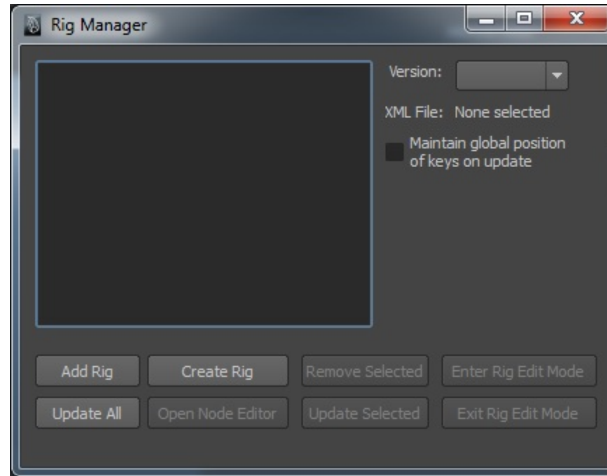


Figure 7: The Rig Manager interface for an empty scene.

When the user clicks the "Add Rig" button, a file dialog is shown allowing the user to select a rig definition file from the file system. These are typically stored within a subdirectory of the Maya project called "rigDefinitions". When a rig definition file is selected by the user, the **loadRig** command is called with the file path provided as a parameter, resulting in an instance of the rig being loaded into the scene.

If the user clicks the "Create Rig" button, **Rig Edit Mode** is immediately entered and the user can begin creating a new rig using the Node Editor. Entering **Rig Edit Mode** involves saving the current scene, and creating a new empty scene where interactions with an instance of the rig result in changes to its definition file. A text label that says "Rig Edit Mode" is also added to the viewport of the 3D scene to make it clear to users that they are operating in **Rig Edit Mode** (see Figure 8). While in **Rig Edit Mode**, only two buttons can be selected in the Rig Manager interface: the "Open Node Editor" button, and the "Exit Rig Edit Mode" button. The "Open Node

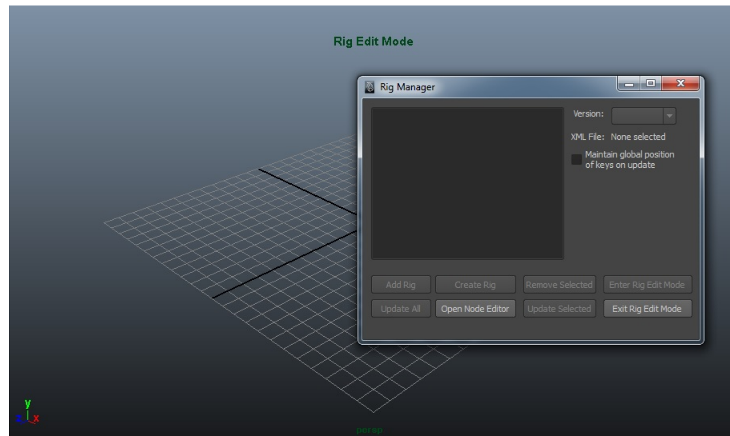


Figure 8: The Rig Manager interface and "Rig Edit Mode" label on the viewport after clicking the "Create Rig" button.

Editor" button will display the Node Editor interface, which is discussed in the next section. The "Exit Rig Edit Mode" leaves **Rig Edit Mode**, unloading the current scene and returning to the last scene loaded before the "Create Rig" button was pressed.

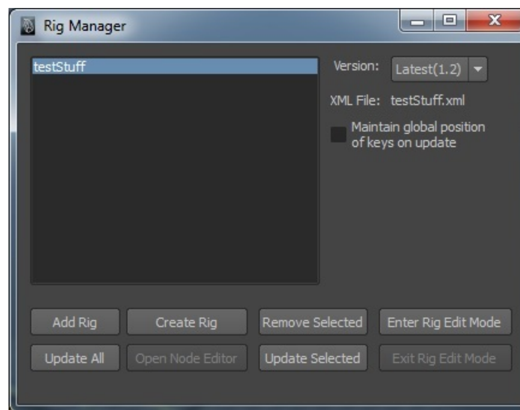


Figure 9: The Rig Manager interface after a rig instance has been loaded into a scene.

Once an instance of a rig has been loaded into a scene, it is added to the list of rigs in the upper left of the Rig Manager panel (see Figure 9). When a rig is selected from this list, information on the right side of the panel is populated for this rig. Several buttons at the bottom of the Rig Manager panel are also enabled for performing actions on the rig selected from the list. These buttons include:

Remove Selected removes the selected rig from the scene by calling the **removeRig** command with the name of the **MetaRootNode** for the rig instance provided as a parameter.

Update Selected updates only the selected rig with the **updateMetadataManager** command. If the "Maintain global position of keys on update" option box is checked, it calls the command with the **globalPos** parameter.

Enter Rig Edit Mode enters **Rig Edit Mode** and loads an instance of the selected rig into the scene for editing. When the Node Editor is displayed, it shows the node graph for the selected rig.

The drop down box for Version is also populated with a list of all versions of the rig definition file available. The version selected in the drop down box is the version currently loaded for the selected rig instance. If a different version is selected, the “Update Selected” button will be displayed in red until the rig is updated (see Figure 10).

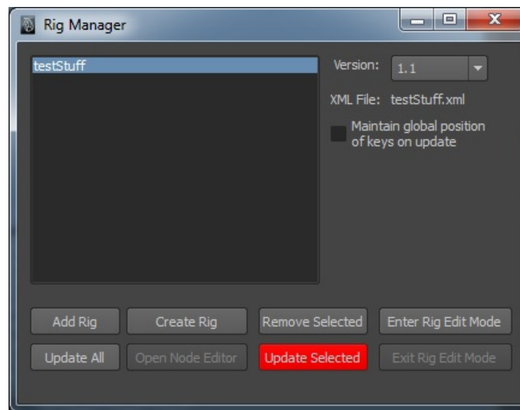


Figure 10: The Rig Manager interface after a different version is selected from the Version drop down box.

4.3.2 Node Editor

While some graphical assets and the general structure of Tim Withers’ interface are re-used in the Node Editor, the underlying commands executed by the interface have been replaced by code that interfaces with the rest of the rigging system [Withers, 2012]. The selection of nodes available in the Node List Panel is different, as is the information presented in the Node Details Panel and the Node Description Panel when nodes are selected.

If the user opens the Node Editor interface when creating a new rig, no nodes are shown in the Node Network Panel because no definition file currently exists (see Figure 11). The user can begin authoring a new rig definition by dragging different node types

from the Node List Panel into the Node Network Panel. Descriptions of each type of node are shown in the Node Description Panel when a node type is selected from the Node List Panel.

As the user begins placing nodes into the Node Network Panel, a temporary rig definition file is created. This is used to update a rig instance in the 3D scene as changes are made. Nodes added in the Node Network Panel correspond to different types of elements in the rig definition file. The color of the graphic used to represent each node corresponds to a different type of element in the rig definition file. For instance, red is used on all nodes that represent component elements.

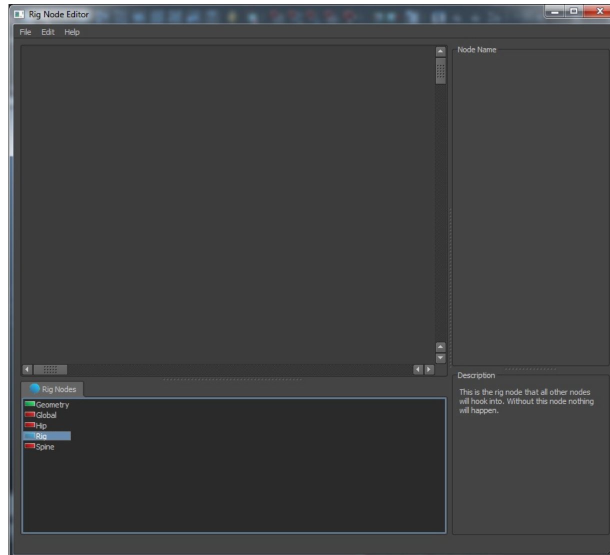


Figure 11: The Node Editor interface is empty when the user begins creating a new rig definition.

The first node that must be added to the Node Network Panel to begin creating a rig is the **Rig** node, and corresponds to the **Rig** element in XML. When a Rig node is selected in the Node Network Panel, information and parameters for the node are displayed in the Node Details Panel (see Figure 12). For a **Rig** node, this includes the current version of the rig definition file being authored and a text box for setting the

name of the rig.

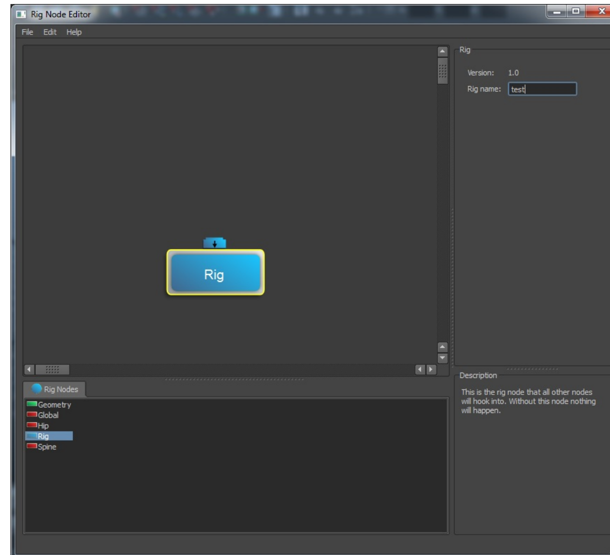


Figure 12: A Rig Node has been added and selected in the Node Network Panel, and settings for the node are displayed in the Node Details Panel.

As nodes are added to the Node Network Panel, the changes that the nodes represent will not be reflected in the scene until they are attached to the node graph for the rig. This is done by attaching an output connection from a node to an input connection on another node. A **Geometry** node is used to add a piece of geometry to a rig, and corresponds to the **Geo** element in XML. In the Node Details Panel for the **Geometry** node, a button is provided that will call a File Dialog for selecting a Maya ascii file containing geometry. When a file is selected, a list box is populated with the name of every geometry object present in the file. The geometry selected from the list will the be the geometry used by the rig (see Figure 13).

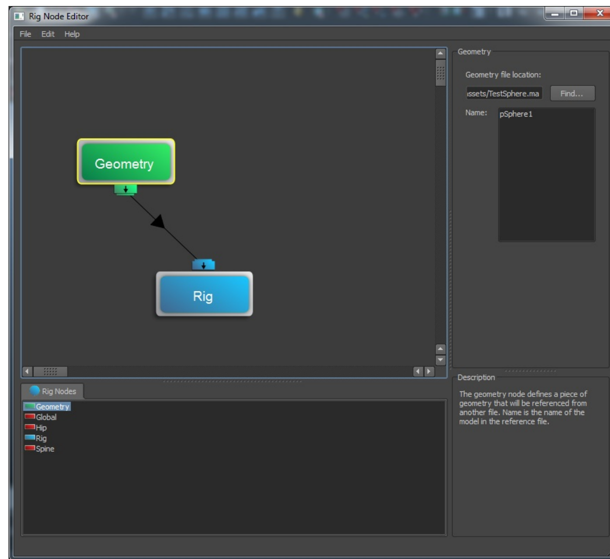


Figure 13: A Geometry Node attached to a Rig Node.

The **Global** node corresponds to a **Component** element of type “Global” in XML. The Node Details Panel for the **Global** node allows the user to set the name of the global component as well as the icon and color of the controller for the global component. The drop down boxes are populated with a set of pre-determined options created in Python functions (see Figure 14).

After a **Global** node is connected to the graph, a controller for the global component is created for the rig instance shown in the 3D scene. When this controller is moved, its position in world space is used for the **Location** element of the global component in the XML file. The rig definition file is automatically updated by this change in position by using a **scriptJob**. A **scriptJob** is a feature of Maya that allows a user-defined function to be called whenever the values of a channel on an object are modified. In this case, the function causes the rig definition file to be regenerated from the node graph in the Node Editor.

The **Hip** node represents a **Component** element of type “Hip” in an XML file. The settings for the **Hip** node in the Node Details Panel are identical to those available for the **Global** node. When a **Hip** node is attached to the node graph, both a controller

and joint object are created in the 3D scene for the hip component. Since the joint is parent constrained to the controller, the controller is used to update the **Location** element of the hip component.

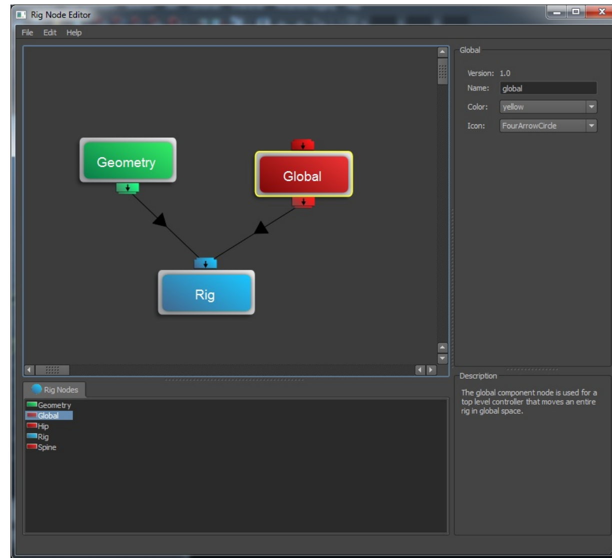


Figure 14: A Global Node attached to a Rig Node.

The **Spine** node corresponds to a **Component** element of type "Spine" in XML. There are many settings available for the **Spine** node in the Node Details Panel. The "FK Color" and "FK Icon" widgets set the icon and color used for the FK controllers of the spine. The "Kin. Type" widget allows the user to change among the three Kinematic Types available for spine components. If a IK Kinematic Type is selected, additional widgets are added for setting the icon and color of the shoulder controller. The index of the joint in the parent component to which the spine is attached can also be set using the "Parent Number" widget. By default a spine is built with three joints, but more can be added using the "Add Joint" button. If more than three joints are present, joints can be be removed using the "Remove Joint" button (see Figure 15).

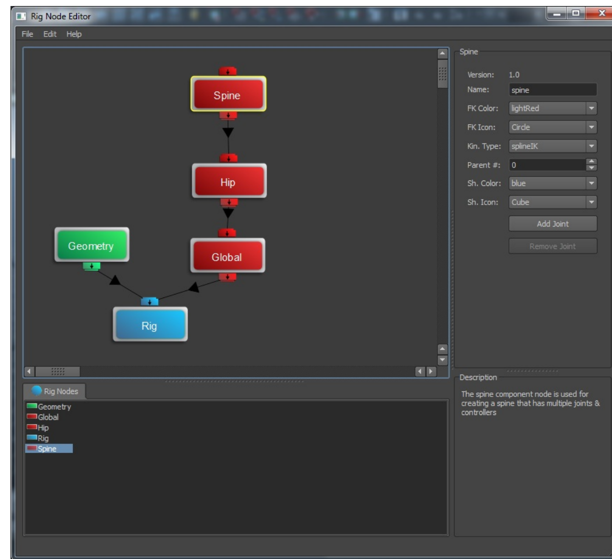


Figure 15: A Spine Node added to the node graph for a rig.

Once the user is done creating or modifying a rig definition, a new XML file can be saved using the File menu at the top of the Node Editor. If a rig definition file already exists for this rig, the rig version is incremented.

5 DISCUSSION

This system aids scene organization by providing mechanisms for managing different versions of rigs and loading them into scenes. The Metadata Node Network makes it possible to update rigs that have already been deployed in scenes and provides a method to help preserve animation keys despite changes to the rig. This would be difficult to do with Maya's default file referencing system. The node-based interface also greatly accelerates rig design and hides most of the more technical aspects of rigging from the user. Additionally, it entirely eliminates the need to script rigs designed with the system.

In the absence of this rigging system, users who wish to create a rig either need to manually create all of the connections in Maya to build a functional rig, or write a script using Python or MEL that will build the rig for them. They could also utilize another free rigging system, but none of the systems presently available provide a node-based interface for customizing the rig and easily modifying the relationships between the rig's components. Additionally, such systems do not provide any mechanism for easily updating existing rigs that have been created. Generally these rigging systems only provide a user interface that sits on top of a Python or MEL script which can only create new rigs.

The greatest weakness of the system remains the limited number of components that are available to the user for constructing rigs. In order to create a complete character rig, many more features such as blend shapes for facial performance must also be supported by new types of components. For characters with extremely unusual performance characteristics, the system may also prove insufficient. An example of this is a character who has limbs that can be detached and re-attached to different locations on the body. This type of problem is much more difficult to solve, as it undermines basic assumptions about the architecture of the system such as parenting relationships between components.

The rigging system fills a gap as an open source node-based rigging system. While many free or open source rigging systems exist for Maya, none of those currently

available provide the node-based interface or version capabilities implemented within this system. This provides for small studios and individuals capabilities typically not found outside of proprietary tools at large animation studios.

The software-agnostic rig definition format used by this system also solves pipeline issues for studios. Some of these issues include sharing data among many different software packages, or sharing data between facilities at different locations. The rig definition files can be managed with existing version control systems such as Git, which benefits 3D animated productions that choose to use such systems. While the rigging system is not a complete pipeline solution, its rig definition approach could be integrated into larger pipeline packages.

6 FUTURE WORK

As previously mentioned, the system would greatly benefit by the addition of a larger variety of components. The more components made available in the system, the greater the variety of rigs that can be constructed by the user. Some components that come to mind include a component for rigging faces, and components for quadrupedal legs and spines. Non-organic types of components could also be supported, such as common types of machine components. The system could also integrate with a modeling and rigging tool such as the one created by Chris Wheeler in his thesis work [Wheeler, 2009].

The rig definition format could be expanded to include support for unusual rig performance characteristics, such as dynamic parenting relationships between components. Many different types of relationships between components could be supported, such as components that are only constrained by orientation to their parent components.

Instancing of rig components is an important feature that could be supported by the definition format. This would make it easier to re-use the same component or set of components within different parts of the same rig, or across several rigs. The rig definition format could support referencing pieces of a rig, where shared components are stored in their own separate files. Thus changes to those components can more easily be shared and propagated.

A feature that was considered, but dropped early in development due to time constraints, is the addition of a picker interface for selecting controller objects in a 2D control panel. Such a picker interface could either be designed in a WYSIWYG drag-and-drop environment, or procedurally generated by an automated system that determines how to organize and place the 2D elements in the interface.

In order to take advantage of the software-agnostic rig definition format, implementations of the rigging system in other software packages such as Houdini, 3D Studio Max, or Softimage should also be created. Assuming that such implementations are possible, the same rig could be used seamlessly in all three packages. Additionally, project management tools could be written that use the information in the rig definition

files in order to track development progress on rigs. Such tools could be written in HTML5 and WebGL so that rigs could be viewed in 3D in a web browser without one ever having to own or open a standalone 3D animation package.

While the functions for loading, updating, and removing rigs were written as a C++ plugin for performance reasons, an alternative implementation in Python is possible. A Python implementation would make it easier for new developers to begin modifying the system and more quickly to begin implementing changes.

The XML rig definition format could also benefit from the creation of a schema. A schema is a file used by a XML document that lists all valid elements, what attributes each element has, and how elements can be structured in relation to one another in the document. If a schema is provided, any given XML document can be validated against the schema to determine if it is validly structured. This helps to catch errors when an XML document is generated and enforces consistency for a particular document type.

Once this document is published, the full source code for the rigging system will be hosted on the website GitHub, which has been used to share codebases for many large software development projects. It is the author's hope that by his doing so, others will be encouraged to learn from the techniques implemented in this rigging system, and that they may contribute with their own improvements to the project.

REFERENCES

Anthony A. Apodaca and Larry Gritz. *Advanced Renderman: Creating CGI for Motion Pictures*. Morgan Kaufmann, San Francisco, CA, 1999.

Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001. URL http://www.sciam.com/article.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21.

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (xml) 1.0 (fifth edition). Technical report, World Wide Web Consortium, November 2008. URL http://www.w3.org/TR/REC-xml/.

Douglas Crockford. The application/json media type for javascript object notation (json). Technical report, Internet Engineering Task Force, July 2006. URL http://www.ietf.org/rfc/rfc4627.txt. RFC 4627.

Digia. *Qt Designer Manual*, 2013. URL http://qt-project.org/doc/qt-4.8/designer-manual.html.

David Gould. *Complete Maya Programming (Volume I): An Extensive Guide to MEL and the C++ API*. Morgan Kaufmann Publishers, San Francisco, CA, 2003.

David Hunt. Modular procedural rigging. In *Game Developer's Conference 2009*, San Francisco, CA, 2009. URL http://www.gdcvault.com/play/1708/Modular_Procedural_Rigging.

Paul Leach. A universally unique identifier (uuid) urn namespace. Technical report, Internet Engineering Task Force, July 2005. URL http://www.ietf.org/rfc/rfc4122.txt. RFC 4122.

Richard Lewontin. The genotype/phenotype distinction. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2011 edition, 2011. URL http://plato.stanford.edu/archives/sum2011/entries/genotype-phenotype/.

Erik Malvarez, Scott Sharp, and Stephen Unterfranz. A new approach to procedural character rigs. In *ACM SIGGRAPH 2008 Talks*, SIGGRAPH '08, pages 47:1–47:1, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-343-3. doi: 10.1145/1401032.1401093. URL http://doi.acm.org/10.1145/1401032.1401093.

Tom Ngo. Office open xml overview. Technical report, European Computer Manufacturers Association, 2006. URL http://www.ecma-international.org/news/TC45_current_work/OpenXML%20White%20Paper.pdf.

Rob O'Neill. *Digital Character Development: Theory And Practice*. CRC Press, Burlington, MA, 2008.

Stuart C. Shapiro. Semantic networks. In *Encyclopedia of Artificial Intelligence*. Wiley, 1992. URL http://www.jfsowa.com/pubs/semnet.htm.

Jason Smith and Jeff White. Blockparty: Modular rigging encoded in a geometric volume. In *ACM SIGGRAPH 2006 Sketches*, SIGGRAPH '06, New York, NY, USA, 2006. ACM. ISBN 1-59593-364-6. doi: 10.1145/1179849.1179993. URL http://doi.acm.org/10.1145/1179849.1179993.

Chris Wheeler. An automated system for the creation of articulated mechanical parts. Master's thesis, Texas A&M, College Station, TX, 2009.

Tim Withers. Developing a framework for a new visual-based interface design in autodesk maya. Master's thesis, Texas A&M, College Station, TX, 2012.

APPENDIX A

XML ELEMENT ATTRIBUTES

Rig Element Attributes
Name is used by the rig *Load* and *Update* functions when creating objects in the scene. It is the prefix string that is always at the front of every object's name in the scene when it is created. This makes it easier to identify which objects are part of which rig when viewed in the 3D scene.
Version is the version number associated with this rig definition file. It is used to determine if a rig instance is currently up to date with its rig definition.
GuiLocation is a location in 2D screen coordinates for positioning the rig node representing this element in the Node Network Panel of the Node Editor.

Geo Element Attributes
File is a project relative file path which points to the Maya scene file containing the geometry to load. Since the file path is project relative, a project directory must be currently set in Maya's configuration options in order for the geometry to be correctly loaded.
GuiLocation is identical to the same attribute on the **Rig** element, except that it is used for the geometry node corresponding to this element.
Name is used to identify a piece of geometry contained within the geometry Maya scene file.

Component Element Attributes for Global
Name is used as the second prefix for an object's name when created in the scene, appearing after the rig name from the **Rig** element in order to identify which rig and component of which the object is a part.
Version is used by the *Update* function when it is determining whether a particular component needs to be updated or not.
Type defines what type of component is defined by this **Component** element. In the case of a global component it will be equal to "global".
RigId is an identifier string associated with this component which is created randomly from a version 4 UUID (Universally Unique Identifier). UUIDs are designed such that when one is randomly generated, the chances that two UUIDs are identical is statistically unlikely [Leach, 2005]. The rigId is used internally by the rigging system to match component elements in a rig definition with their corresponding metadata nodes in the scene when calling the *Update* function.
Icon identifies which pre-defined shape to use for the global component's controller object. An extensive list of controller shapes is defined by the rigging system, which allows the user to pick one by name instead of having to create a shape from scratch.
Color determines which of the predefined colors provided by Maya to use for the global component's controller object.

The spine component type differs in a significant number of ways from the global and hip component types. It does not use the **Icon** attribute, and some of the attributes used in the global and hip are used in different ways by the spine component. Additional attributes are also added that the global and hip components do not use. The differences in the spine component type's attributes are shown below.

Component Element Attributes for Spine
Color is used to set the color on all of the FK controllers present in the spine component.
FkIcon is the controller shape used by all FK controllers.
KinematicType defines what type of kinematic solution will be used to determine joint positions by the spine component. The three options available are "FK", "splineIK", or "stretchySplineIK". If "splineIK" or "stretchySplineIK" are used, then the **ShoulderControl** child element must be present within this spine component element.
ParentJoint is the index number of the joint and controller in the parent component of this spine to which the top level joint and controller of this spine will be attached.

ShoulderControl Element Attributes
Color is the color of the shoulder controller.
Icon is identical to the global and hip component's **Icon** attribute, but is used for the shoulder controller.
LocalX, **LocalY**, **LocalZ**, **RotateX**, **RotateY**, **RotateZ**, **ScaleX**, **ScaleY**, and **ScaleZ**. These are used for purposes identical to those of the **Location** element but are for the shoulder controller.

APPENDIX B

METADATA NODE ATTRIBUTES

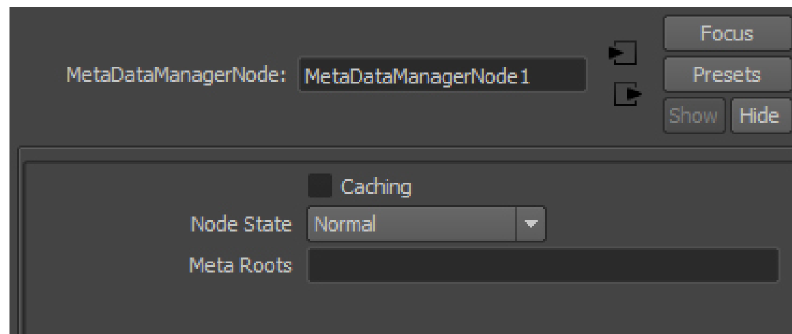


Figure 16: Attributes of a MetaDataManagerNode listed in Maya.

MetadataManagerNode Attributes
MetaRoots attribute is attached to the **MetaParent** attribute of **MetaRootNodes**.

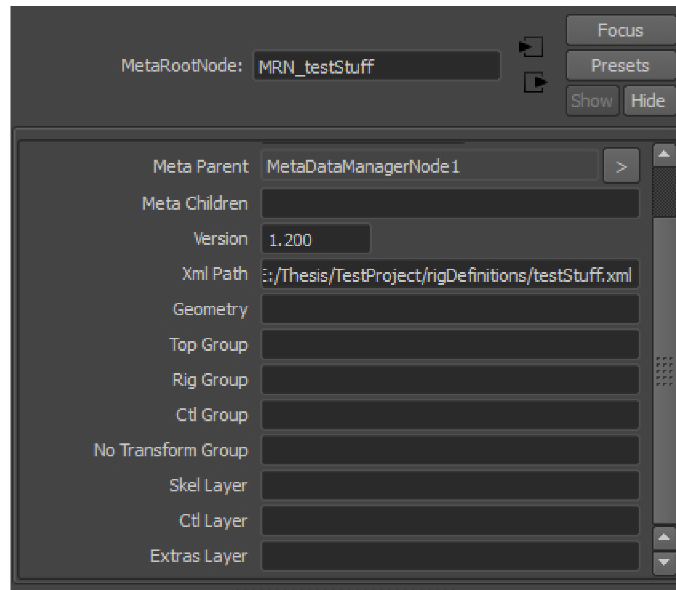


Figure 17: Attributes of a MetaRootNode listed in Maya.

MetaRootNode Attributes
MetaParent is always connected to the **MetaRoots** attribute of the **MetadataManagerNode**.
MetaChildren is attached to one or more **MetaParent** attributes on component metadata nodes.
Version holds the current version number for the rig associated with this **MetaRootNode**. It is drawn from the rig definition file the when it is loaded or updated.
XmlPath is the file path to the XML rig definition file associated with this rig.
Geometry is connected to the **MetaParent** attribute on the geometry node associated with this rig.
TopGroup is connected to the **MetaParent** attribute on the top level group for this rig. This group is used for organizing the scene Outliner, with every other DAG node of the rig as a child of this group node.
RigGroup is connected to the **MetaParent** attribute on the rig group for this rig. The rig group is a child of the top group, and contains all of the joints, geometry, constraints, end effectors, and IK handles associated with the rig.

CtlGroup is connected to the **MetaParent** attribute on the control group for this rig. The control group is a child of the top group, and contains all of the controller objects and clusters associated with the rig.
NoTransformGroup is connected to the **MetaParent** attribute on the NoTransform group for this rig. This group contains certain objects in a rig that should never be moved from their default transformation positions.
SkelLayer is connected to the **MetaParent** attribute on the display layer which contains all of the joints in this rig.
CtlLayer is connected to the **MetaParent** attribute on the display layer which contains all of the controllers in this rig.
ExtrasLayer is connected to the **MetaParent** attribute on the display layer which contains miscellaneous extra objects which the user typically does not need to see, such as clusters and IK handles.

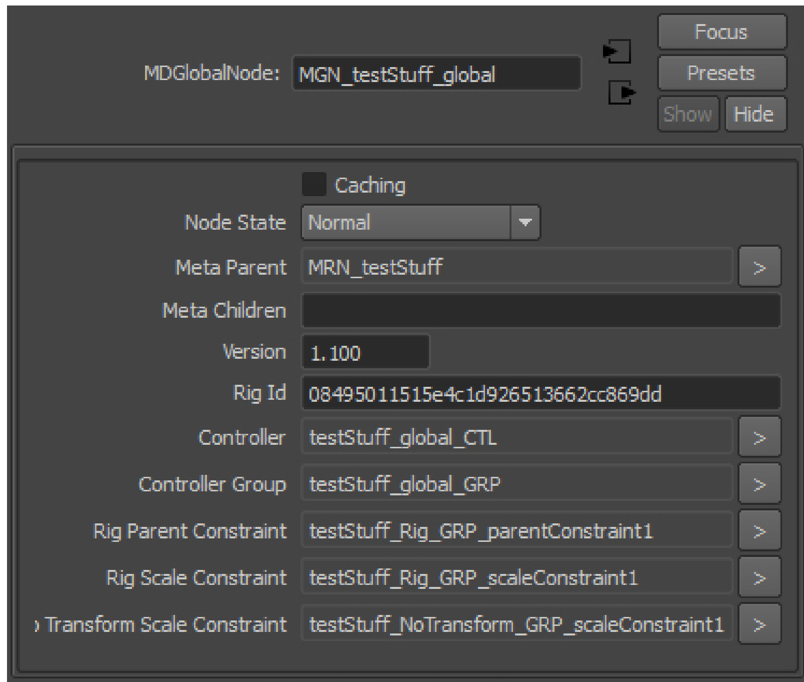


Figure 18: Attributes of a MDGlobalNode listed in Maya.

MDGlobalNode Attributes
MetaParent is connected to the **MetaChildren** attribute of another metadata node such as a metadata component node, or a **MetaRootNode**. The metadata node to which it is connected is identical to the parenting relationships in the rig definition file.
MetaChildren is connected to the **MetaParent** attribute of another metadata component node.
Version holds the current version number of this component as defined in the rig definition file when it was last loaded or updated. It is used by the *Update* function determine if a component needs to be updated.
RigId holds the UUID number for this component as defined in the rig definition file. It is used to match metadata component nodes to their corresponding component element in the definition file when the *Update* function is called.
Controller is connected to the **MetaParent** attribute on the controller associated with this global component.
ControllerGroup is connected to the **MetaParent** attribute on the group holding the controller.
RigParentConstraint is connected to the **MetaParent** attribute on the parent constraint from the global controller to the rig group.
RigScaleConstraint is connected to the **MetaParent** attribute on the scale constraint from the global controller to the rig group.
NoTransformScaleConstraint is connected to the **MetaParent** attribute on the scale constraint from the global controller to the NoTransform group.

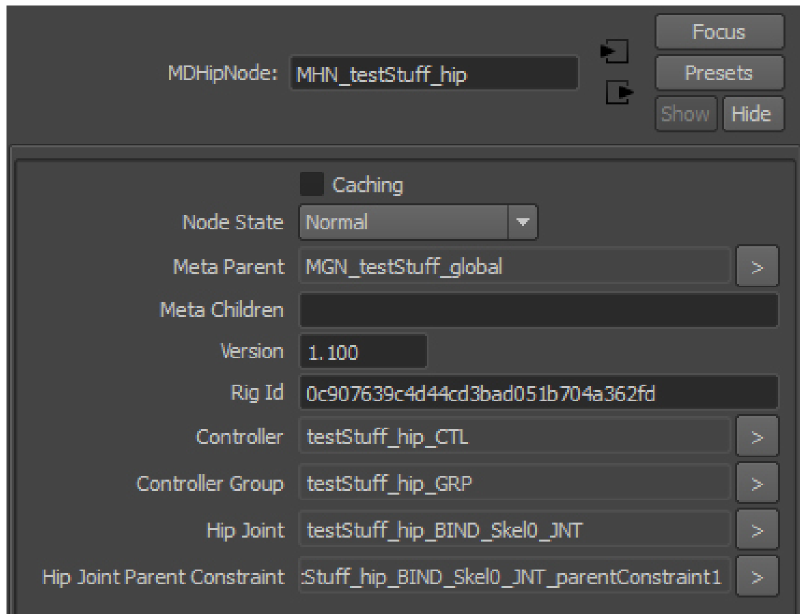


Figure 19: Attributes of a MDHipNode listed in Maya.

MDHipNode Attributes
HipJoint connects to the **MetaParent** attribute on the joint associated with this hip component.
HipJointParentConstraint is connected to the **MetaParent** attribute on the parent constraint from the hip controller to the hip joint.

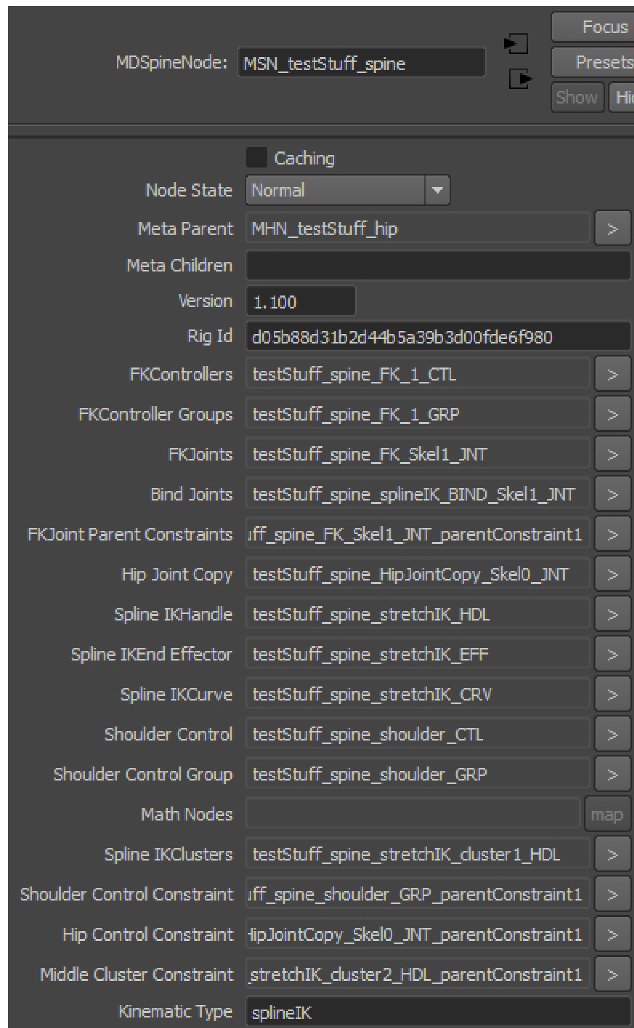


Figure 20: Attributes of a MDSpineNode listed in Maya.

MDSpineNode Attributes
FKControllers connects to the **MetaParent** attributes on all of the FK controllers for this spine component.
FKControllerGroups connects to the **MetaParent** attributes of all the groups containing FK controllers for this spine component.
FKJoints connects to the **MetaParent** attributes of all joints in the FK chain for the spine. If the Kinematic Type of the spine is set to FK, there is no separate FK joint chain so this attribute is not connected to any joint objects.
BindJoints connects to the **MetaParent** attributes of all the joints in a spine component used for binding skin weights to a mesh.
FKJointParentConstraints connects to the **MetaParent** attributes of the parent constraints from FK controllers to FK joints.
HipJointCopy is attached to the **MetaParent** attribute of a copy of the hip joint for the FK joint chain, if the spine is attached to a hip component.
SplineIKHandle is attached to the **MetaParent** attribute of the spline IK handle, if the Kinematic Type of the spine is set to "spline IK" or "stretchy spline IK".

SplineIKEndEffector is attached to the **MetaParent** attribute of the spline IK end effector, if the Kinematic Type of the spine is set to “spline IK” or “stretchy spline IK”.
SplineIKCurve is attached to the **MetaParent** attribute of the spline curve used for a spline IK setup, if the Kinematic Type of the spine is set to “spline IK” or “stretchy spline IK”.
ShoulderControl is attached to the **MetaParent** attribute of the shoulder controller, which is only present if the Kinematic Type of the spine is set to “spline IK” or “stretchy spline IK”.
ShoulderControlGroup is attached to the **MetaParent** attribute of the group containing the shoulder controller, if it exists.
MathNodes is attached to the **MetaParent** attribute of math nodes used to calculate scaling values on joints if the Kinematic Type is set to “stretchy spline IK”.
SplineIKClusters is attached to the **MetaParent** attribute of clusters used to control the spline curve if the Kinematic Type is set to “spline IK” or “stretchy spline IK”.
ShoulderControlConstraint is attached to the **MetaParent** attribute of the parent constraint from the last FK joint in the group containing the shoulder controller.
HipControlConstraint is attached to the **MetaParent** attribute of the parent constraint from hip component’s group to the copy of the hip joint, if it exists.
MiddleClusterConstraint is attached to the **MetaParent** attribute of the parent constraint from the shoulder controller and parent controller of the parent component to the middle cluster controlling the spline IK curve, if it exists.
KinematicType holds the current Kinematic Type of the spine, as defined in the rig definition file the last time it was loaded or updated. This is drawn from the KinematicType attribute of the spine element in the file.

APPENDIX C

C++ CLASSES

XmlGuide loads an XML file into a hierarchy of guide information data structures. It uses the RapidXML library for file parsing. It contains a pointer to the guide object for the top level component.

ComponentGuide is the abstract base class for all component guides. It provides shared functionality and a consistent interface for component guide objects. It can contain a list of one or more child **ComponentGuide** objects.

GlobalComponentGuide derives from **ComponentGuide** and provides functionality for loading guide information for a global component.

HipComponentGuide is similar to **GlobalComponentGuide** but for hip components.

SpineComponentGuide is similar to **GlobalComponentGuide** but for spine components.

Rig is the top level class for all functionality related to loading, updating, and removing rigs. It contains a pointer to an **XmlGuide** object which it uses for its load and update functions. It contains member variables for the **RigIdManager** and top level **Component** object.

RigIdManager manages a dictionary data structure whose keys are RigIds, and values are pairs of pointers to **Component** objects and metadata component node handles. This is used to map components in a rig definition file to their matching metadata components nodes when the rigs are updated.

Component is the abstract base class for all component classes. It provides a consistent interface to load, update, and remove individual components. It contains a pointer to a **ComponentGuide** object. It can also contain a list of one or more **Component** objects.

GlobalComponent derives from **Component** and provides functionality for loading, updating, and removing global components.

HipComponent similar to **GlobalComponent** but for hip components.

SpineComponent similar to textbfGlobalComponent but for spine components.

LoadRigUtils provides utility functions to simplify common tasks necessary for loading, updating, and removing rigs. Some of these functions are exposed as commands through the plugin so they can be used in scripts.

MyErrorChecking provides macros useful for making error checking in Maya plug-ins less labor intensive. The code for these macros was lifted from *Complete Maya*

Programming [Gould, 2003].

APPENDIX D

COMMAND PARAMETERS

The long name of parameters in the listing is shown first, with the short name in parentheses afterwards.

updateMetadataManager Parameters
name (n) is an optional parameter. It is provided only the meta-root node of the provided name is updated.
force (f) is an optional parameter. It is provided version checking is ignored when the update is performed, forcing updates.
replace (r) is an optional parameter. It is provided an alternate path to the rig definition file is used instead of the path provided in the **MetaRootNode XmlPath** attribute.
globalPos (g) is an optional parameter. It is provided when the update is performed the world space position of keys on animation curves attached to controllers will be maintained.

loadRig Parameters
path (p) is a required parameter. It must contain an absolute path to an XML rig definition file.

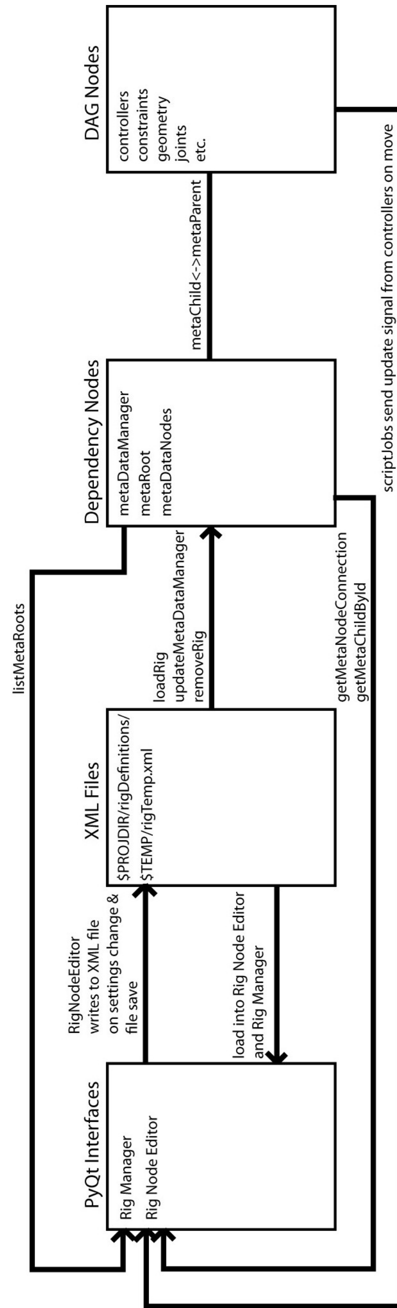
removeRig Parameters
name (n) is a required parameter. It must contain the name of a **MetaRootNode** for a rig instance currently loaded into the scene.

getMetaNodeConnection Parameters
node (n) is a required parameter. It specifies the name of a metadata node. The attributes of this node will be used to find connections to other nodes.
connection (c) is a required parameter. It specifies the name of an attribute on the given metadata node.
index (c) is an optional parameter. It specifies the index number of a connection on an attribute if that attribute is connected to multiple nodes. If there are multiple connections and the index is not specified, then the first object connected is returned.

getMetaChildById Parameters
node (n) is a required parameter. It specifies the name of a metadata node. The **metaChildren** of this node will be used to find connections to other metadata nodes.
rigIdentifier (rid) is a required parameter. It specifies the rigId string to search for in the list of child metadata nodes.

APPENDIX E

ARCHITECTURE DESIGN DIAGRAM



APPENDIX F

GLOSSARY

DAG stands for Directed Acyclic Graph. The DAG is a subset of Maya's Dependency Graph where no child node can be its own parent. The DAG is used for objects that have a 3D position associated with them in the scene, such as geometry or groups.. 16, 18, 38

Dependency Graph is the fundamental structure by which every scene construction operation in Maya is accomplished. This is done by creating graphs of connected nodes that recieve input, process it, and then create output to other nodes.. 5, 6, 16

JSON stands for JavaScript Object Notation, and is a text-based open standard for exchanging data. It originated as a format for sharing data between JavaScript executed in web browsers and web servers. v, 5–8

metadata is a term that means "data about data". Metadata is useful in web technologies to describe what type of content is on a website. In a rigging context, it is used to describe components of a rig in an abstract way, such as calling a piece of a rig an "arm" or "leg". 1, 4, 5, 7–10, 16–18, 35, 38, 40, 45–47

Metadata Node Network is a network of nodes constructed in a scene graph in order to enable semantic traversal of a rig. The network is attached to every object in the scene that composes a rig. The rigging system implemented for this thesis creates a Metadata Node Network in Maya's Dependency Graph. v, 1, 5, 9, 10, 12, 14, 16–18, 27

Node Editor is the interface for authoring new rig definition files. The rig is represented in this interface as a graph of interconnected nodes which can be created, connected together, or deleted. The Node Editor can only be accessed while in Rig Edit Mode.. v, 6, 11, 13, 18–20, 22, 23, 26, 34

PyQt is a binding of the Qt framework which is commonly used for developing graphical user interfaces. 1, 6, 14, 18

Rig Edit Mode is a special mode of operation that the user can enter through the Rig Manager interface. Rig Edit Mode loads an instance of a selected rig in the scene which alters the default configuration of the rig when changed. The Node Editor interface is also accessible while in Rig Edit Mode.. 11, 13, 16, 19, 20, 22

Rig Manager is the interface used for loading, updating, and removing rigs from a scene. Rig Edit Mode can also be accessed through this interface.. v, 1, 10, 11, 14, 18–22

Semantic Network is a graph that represents knowledge with interconnected nodes and arcs. Semantic networks have been used in philosophy, psychology, linguistics, and computer software. Metadata Node Networks use many of the same concepts devised for semantic networks.. 3, 4

XML stands for Extensible Markup Language. XML is a document format traditionally used in web technologies for distributing arbitrarily structured documents and data. v, 1, 2, 5–8, 14, 17, 24–26, 31, 38, 45, 46