Clemson University
TigerPrints

All Theses | Theses

5-2011

OPEN SOURCE RIGGING IN BLENDER: A MODULAR APPROACH

Ryan Cushman
Clemson University, r.n.cushman@gmail.com

Follow this and additional works at: https://tigerprints.clemson.edu/all_theses

Part of the Computer Sciences Commons

Recommended Citation

Cushman, Ryan, "OPEN SOURCE RIGGING IN BLENDER: A MODULAR APPROACH" (2011). *All Theses*. 1055.
https://tigerprints.clemson.edu/all_theses/1055

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact kokeefe@clemson.edu.

OPEN SOURCE RIGGING IN BLENDER:
A MODULAR APPROACH

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Fine Arts
Digital Production Arts

---

by
Ryan Cushman
May 2011

Accepted by:
Dr. Donald House, Committee Chair
Dr. Timothy Davis
Dr. Brian Malloy

ABSTRACT

Character rigs control characters in the traditional CG pipeline. This thesis examines the rig creation process and discusses several problems inherant in the traditional workflow—excessive time spent and a lack of uniformity, and proposes a software plugin which solves these issues. This thesis describes the creation of a tool for Blender 3D which automates the rigging process yet keeps the creativity and control in the hands of the user. A character rig designed by this tool will be fully functional--yet capable of being split into its component parts and reconstructed as the user determines. These body parts are individually scripted with the intent of maximizing reusability, and the code and rigs are distributed to the open source community for vetting. The final tool has been downloaded many times by the Blender community and has met with very postive responses.

DEDICATION

I would like to dedicate this thesis to Sarah, my wife. She did not hesitate when I proposed to throw our lives in the air with a crazy plan to work in feature films. Instead, she made the entire process more of a joy than I can express. Through the good times and the rough, she has been on every side of me holding me up.

ACKNOWLEDGMENTS

First, I would like to thank my family—especially Sarah, who put up with odd discussions and even odder hours during the writing of this thesis.

Thanks to Dr. Donald House, who provided much needed guidance and friendship. I would also like to thank my committee members, Dr. Davis and Dr. Malloy, for their time and help—in both this thesis and across my education.

Several fellow students helped out in a big way, so I would like to say a special thank you to Zach and Tony. Without their expertise, this thesis never may have progressed beyond theory.

Finally, thank you to the Blender community—especially Ton Roosendaal and the crew at "Blender Artists." I could not have gotten any of this accomplished without the many things that they have accomplished in the past and the advice and criticisim that they provided as I worked.

TABLE OF CONTENTS

i. Abstract ii

ii. Dedication iii

iii. Acknowledgments iv

I. Introduction 1

II. Background. 4

2.1 The History of Rigging and Animation 4

2.1a: Rigging and Animation 4

2.1b: Rigging Plugins 7

2.2 History of Open Source Development 11

2.3 The Rigging Process 20

III. Methodology. 24

3.1 Choosing and Learning a 3D Package 24

3.2 The Character Design Process 27

3.3 The Rig Design Process 28

3.3a The Legs and Feet. 29

3.3b The Arms and Hands 33

3.3c The Spine 34

3.3d The Face and Head. 37

3.4 Writing the plugin. 39

3.4a TheModules. 41

3.4b The Main Control Panel. 49

IV Evaluation 54

V. Conclusion 57

VI. Bibliography 59

IMAGE INDEX

Figure 1.1: "Mr. Potato Head" 3

Figure 2.1: "Cockpit Simulation of The Human Factor" 4

Figure 2.2 "A Two Gyro Gravity Gradient Altitude Control System". 5

Figure 2.3: "The Skeletal Animation System" 6

Figure 2.4: BlockParty Workflow 10

Figure 2.5: FK Articulation 21

Figure 2.6: Copy Location Constraints 22

Figure 2.7: Control Objects 23

Figure 3.1: Polygon Modes 26

Figure 3.2: Armature Modes 26

Figure 3.3: A Blender Bone 27

Figure 3.4: Maya Joints 27

Figure 3.5: The Design Process 27

Figure 3.6: Stretchy Limbs. 30

Figure 3.7: Rubber Hosing. 31

Figure 3.8: Knee Correction. 31

Figure 3.9: The Foot Roll Control. 32

Figure 3.10: Leg and Foot Rig Overview 32

Figure 3.11: The Clavicle Control. 33

Figure 3.12: The Finger Roll Control 34

Figure 3.13: Bbone Properties. 35

Figure 3.14: Bbones Curving 36

Figure 3.15: Bending and Stretching Spine. 36

Figure 3.16: Spine Rig Overview 37

Figure 3.17: Facial Rig Overview. 38

Figure 3.18: Mouth Shape Controls. 39

Figure 3.19: Main Control Panel 41

Figure 3.20: A Rig Module 41

Figure 3.21: Control Panel Breakdown 50

I. INTRODUCTION

Character rigging is the part of the character creation pipeline between the modeling of the 3D character and the animation. Character rigs provide controls which dictate the animation of a character. Often thought of as a tedious and overly complicated process, rigging is necessary for even the most basic of animations. Without a rig, a 3D model is nothing more than a digital statue. For complicated scenes (such as crowd scenes), the rigging process can often bottleneck the entire production pipeline as artists must spend time rigging every character.

Rigging is often considered to be a distasteful process for many reasons. For one, it involves tedious repetition of actions; in addition, it requires the implementation of solid logic and problem solving skills. Many rigs are extremely technical and operate as much by programing as manual character animation. The standard solution to these problems is to write a custom tool which simplifies the process and various attempts have been made at this over time.

Many of the best rigging tools are the property of individual studios and cannot be used by outsiders. Of the remaining options, most are either expensive and/or lack functionality. The high price point is a direct result of the sheer effort that goes into maintaining a viable rigging tool. If it is going to have any flexibility or use in a pipeline, it needs a great deal of maintenance and development.

This thesis demonstrates a plugin design that addresses functionality issues and also proposes a development format which allows maintenance to be carried on through the open source movement. A plugin created within the right community and released in

the right format can be more functional and useful than existing rigging tools. The challenge then comes from creating a logical structure for development which lends itself towards open source development and results in a product with all of the features and flexibility a useable rigging pipeline tool requires.

This thesis attempts to begin the removal of much of the mundane tedium from rigging by providing a tool which can both streamline the rigging process for riggers, as well as making rigging accessible to modelers and animators. The goal is to create a plugin for a 3D package which initiates a "character body part library". In the end, every possible body part would be cataloged and a solid rig for that body part would be scripted. Once this point is reached, any character could conceivably be rigged by calling the right combination of these modules. A biped could become a faun by calling quadruped legs. A quadruped could become a centaur by calling a biped spine, arms, and a human face.

A great analogy for the way this tool should work is the Mr. Potato Head children's toy (as pictured in Figure 1.1). When playing with this toy, children begin with a blank slate, and from this point they combine various body parts to create a wide variety of characters. If the child wants a different shape of mustache or arm, they simply switch it out from their box of spare body parts. With this workflow in mind, the plugin has been titled “Mr. Potato Rig.”

Figure 1.1 – “Mr. Potato Head”

Image of Mr. Potato Head and Mr. Potato Head toy are the property

of Hasbro Inc.

The initial modules provided to users will be the individual elements required for a cartoony biped character. The idea is that as the open source community starts adapting the tool, they will add their new work to the release and the desired functionality and flexibility will be achieved through the efforts of many with minimal time by one.

II. BACKGROUND

2.1: The History of Rigging and Animation

2.1a—Rigging and Animation

The term "computer graphics" was first used in the early 1960s. William Fetter of Boeing created a series of drawings entitled "Cockpit Simulation of Human Factor". Shown in Figure 2.1, these drawings are 2D line drawings and bear very little resemblance to modern computer graphics, but they do demonstrate that even the earliest computer graphics simulations had to do with understanding the human form. In 1961, Edward Zajac from Bell Labs created the first computer generated film (pictured in Figure 2.2). His film was called "A two gyro gravity gradient altitude control system" and demonstrated that a satellite could be designed to always face the earth as it circled [5].

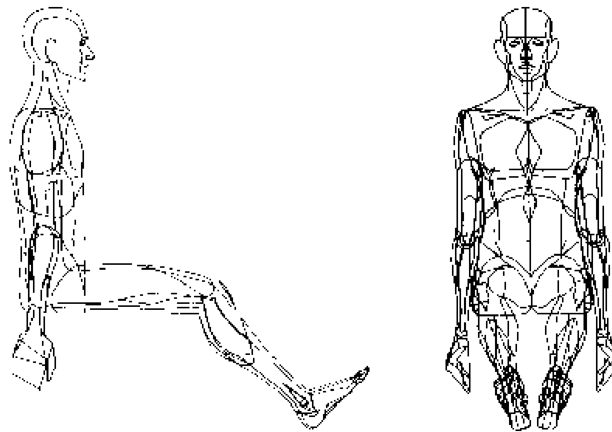


Figure 2.1: "Cockpit Simulation of The Human Factor"

by William Fetter

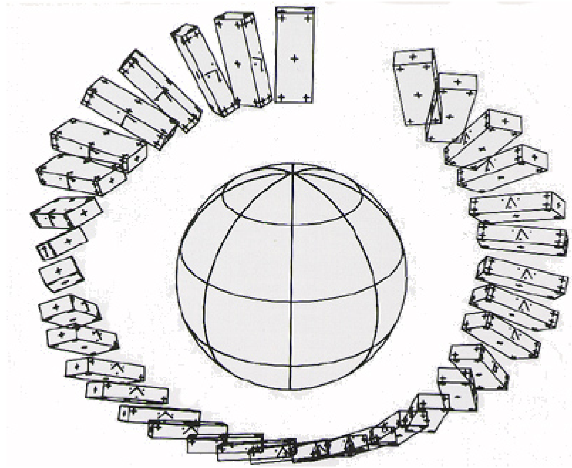


Figure 2.2 “A Two Gyro Gravity Gradient Altitude Control System” by Edward Zajac

The history of character rigging within the broader spectrum of computer graphics is a fairly convoluted story. Many people had many different ideas and implemented them with varying degrees of success. The human form has always been fascinating and that a substantial amount of work has been devoted to the subject comes as no surprise. One of the main early researchers was David Zeltzer.

In 1983, Zeltzer authored a paper entitled “Towards an integrated view of 3-D computer animation.” This paper was written in the early days of computer graphics. At this time, computer graphics driven character animation was still a struggling concept in the minds of many software developers. “How should the artist interact with the characters and the software?”, “to what extent should motion be physically simulated in order to achieve the best results?”, and other questions asked by Zeltzer were heated topics of debate. In the paper, Zeltzer declared that his goal was to “provide a conceptual

framework for 3-D computer animation in general and character animation in particular [23].”

Zeltzer wrote many papers and co-authored several books dealing with the technical side of character animation. His most direct contribution to the field of rigging is a work called the “Skeletal Animation System.” This system was a method of achieving creature animation through the usage of skeletons and motion goals. Zeltzer's work and the basic logic behind it are some of the most fundamental contributions to this topic in the computer graphics industry [6].

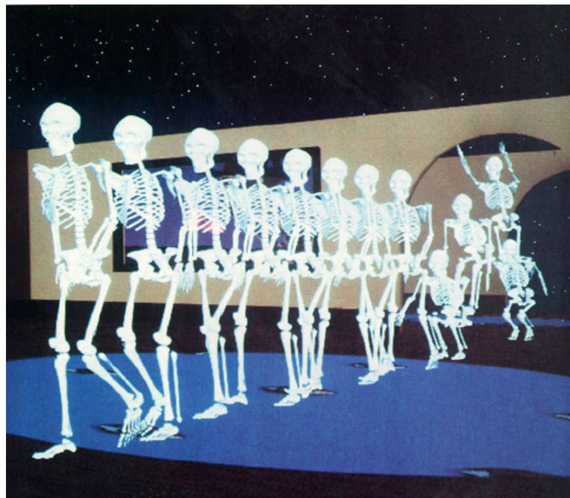


Figure 2.3: “The Skeletal Animation System” by David Zeltzer

Character rigging does not have any one historical originator. Like many scientific disciplines, it has been the result of many researchers making their own contributions and working off of the research that has gone on before them. While Zeltzer's system is one of the best examples of early rigging implementation, a great deal

of work done by other innovators made Zeltzer's work possible and has resulted in the rigging workflow that is generally utilized today. Even today, rigging is still a field under intense development as animators and riggers find new and improved ways to simplify the workflow while improving performance.

2.1b—Rigging Plugins

Rigging plugins appear to be a relatively recent creation, but there has still been a fair amount of work done in the field. If a person wants an "Auto Rig" system, there are typically many to choose from. These tools differ in function and form but usually promise complete character rigs with minimal input from the user. "Advanced Skeleton" [1] and "Character Studio" are two options which are fairly commonly used and which have identical problems. They both make many assumptions about the character and geometry which result in somewhat functional rigs which break easily and are very difficult to modify. These plugins might allow a few basic animation tests for a limited selection of bipedal characters, but they definitely have limited uses and no flexibility at all.

The core idea behind auto rigging is to eliminate user input from the equation and have mathematical formulas dictate how things are set up. There are many things that are wrong with this idea, but there are actually many things that are right. Users need to have control over how things are structured and set up, but on the other hand, once a rigger has created a functional biped rig, why re-invent the wheel every time he wants to rig a similar character?

In 2006, white papers were submitted by both Industrial Light and Magic and Rhythm and Hues detailing their recent advances in modular rigging. ILM's rigging tool is called Block Party and is explained in the paper "BlockParty: Modular Rigging Encoded in a Geometric Volume". Block Party is built based on two separate ideas—rig blocks and volume guides [17].

The first idea (that both papers shared) was the idea of splitting up rigs into their most basic elements--called "blocks"--which could be combined in many different combinations to create any creature. A major benefit of doing rigging this way is that it would result in perfectly identical rig elements across the board--even though the creatures may be set up by a large and diverse team. Further, a change in a block could easily be propagated down the show pipeline. If an animator gave feedback on the arm which required a change, the base block could be modified and--in theory--easily adapted for all other characters that also needed the change.

These blocks were connected together through a system of plugs and sockets. These connections would allow blocks to work together while also being able to be manipulated or changed independently. All plugs and sockets had unique ID numbers which would result in parent\child relationships without need for hard coding.

The second idea has to do with the "Geometric Volume" from the title of the paper--otherwise known as a volume guide [17]. For the first "Chronicles of Narnia" film, ILM had an alarmingly short time to create a large number of characters, rig them with realistic muscle systems, and prepare them for simulation, motion capture, and crowd work. They required a rigging tool that would automate the placement of the

muscles and allow them to quickly prepare as many characters as possible.

The idea is that each rig is associated with a geometric mesh which needs to be sculpted to match the character geometry as closely as possible. This "volume guide" is then used to place and orient joints, generate muscles and collision geometry for cloth simulations, and automate painting of weights. This system allowed non-hero characters, which previously would not have been considered for things like muscle systems, to get the full hero treatment.

> "In pursuit of our original goal, we were able to create one set of muscle geometry and rigs, and have BlockParty replicate them for all creatures that shared the same structure. For instance, after creating a full muscle setup for our leopard, we were able to generate muscle rigs, geometry, deformations and simulation controls for our white tiger, leopard and cheetah. These were then used to drive our volumetric muscle and flesh simulation engine to produce the final result. Creatures like the white tiger were given full muscle setups for only one shot, which would have previously been impossible. Using BlockParty, we were able to derive all thirty-two creatures on 'Narnia' from just three complete creature setups [17] (see Figure 2.4)."

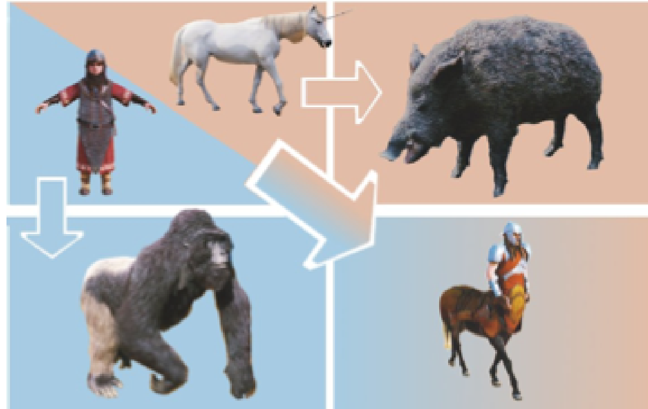


Figure 2.4: BlockParty Workflow

A human was used to create a gorilla, a horse was used to create a boar, and both were used to build a centaur. © 2005 Walt Disney Pictures. All rights reserved.

Rhythm and Hues' paper describing their approach for dealing with character rigging indicates that they developed a similar system--although without as many details [7]. In the end, they scripted 13 modules (BipedArm, HoofedArm, PawedArm, BipedLeg, HoofedLeg, BipedNeck, QuadNeck, BipedSpine, FKBipedSpine, QuadSpine, MultiresTail, Wing) which were used to create every character on the show:

` The team at Rhythm and Hues listed many of the same goals and benefits as the team at ILM. Rig consistency would allow animators to quickly jump between different rigs which had identical rig elements. In addition, the consistency would allow them to transfer motion between rigs (which would be difficult/impossible if the rigs did not match).

In the end, both of these system's greatest strengths are features that are not

appropriate for emulation in "Mr. Potato Rig". Both of these plugins were designed to support feature film pipelines and have complexity that reflects this scope. Features such as transferring of weights, automatic generation of collision geometry, and automatic generation of animation res geometry are useful for a feature studio but make a tool intended for individual artists overly complicated.

This thesis proposes a plugin which is nimble, quick, and--most importantly--simple. It needs to be simple and direct enough so that others could add to it and it can grow from a personal project to a tool that would be supported by the community. The core idea of "Blocks" -- such as presented by both ILM and Rhythm—is appropriate to these goals and is a concept worth emulating and improving upon. Things like muscle systems and automatic transfer of weights will only get in the way of the tools intended purpose.

2.2 The History of Open Source Development:

One of the core ideas behind this project is the benefit offered by developing within the open source community. As one of the base ideas behind the project, it is worth understanding the nature of open source development. One of the core ideas behind open source software has been around since computers began—the idea that software should be free.

For almost as long as there has been software available for purchase, there have been free alternatives. 1984, <u>Time</u> magazine ran an article entitled "Computers: Software is for Sharing." In the article, they tell the story of Dennis Brothers. When Brother's got

his new Mac (a first edition) he realized that it lacked the capability to communicate with other computers over the telephone line. Further, Apple did not provide any documentation on developing new software for their computers. Brother's opened up the computer, studied the electronics, and wrote a program called MacTEP. Without considering selling it, he made it available to anyone who would ask. Within a week, he had a call from CompuServe asking if they could distribute his program to their 135,000 users [9].

In the early days, this was a very common story. At the time of the writing of the Time article, (over 25 years ago) there were already 35,000 free software titles on the market. Like today, many of these options were amateurish and low quality, but even then there were stand out successes. According to the Time article, this sharing of free software grew out of an era when programmers traded programs like baseball cards. "One of the most popular titles was MODEM, a 1977 program that allowed personal-computer owners to send programming instructions to one another by telephone. Its author, IBM Engineer Ward Christensen, takes pride in never having profited from his labor of love. Says he: 'People sometimes send me money out of the blue, but I always send it back [9].'" All of this development happened under the "Free Software" movement -- a movement which would eventually result in the Open Source movement but with some key differences.

In an open letter entitled "Why Free Software is Better than Open Source," Richard Stallman explains that while the Free Software movement is a social movement, campaigning for the rights of users to have free software, the term "Open Source" refers

to a development methodology—a philosophy for writing programs, not just distributing them [18].

While it is true that open source software is traditionally free, there is much more to having an open source project than making it available free of charge. One of the keys to open source development is how the originator handles the rights to the source code. Traditionally, especially in recent years, source code is carefully hoarded by companies as the source of their profit. Code determines companies' quarterly margins and therefore is a carefully guarded treasure. This way of looking at computers and economics has been around ever since IBM separated software acquisition from hardware purchasing.

"In 1984, Richard Stallman, a researcher at the MIT AI Lab, started the GNU project. The GNU project's goal was, simply put, to make it so that no one would ever have to pay for software. Stallman launched the GNU project because essentially he feels that the knowledge that constitutes a running program--what the computer industry calls the source code--should be free. If it were not, Stallman reasons, a very few, very powerful people would dominate computing. The basic tenet of the GNU project and the Free Software Foundation (the umbrella organization for the GNU project) is that source code is fundamental to the furthering of computer science and freely available source code is truly necessary for innovation to continue .[8]"

The term "Open Source" came from a marketing meeting involving many of the leaders of the free software movement in 1997. They were trying to make the jump from computer hobbyists to the leaders of a broad movement. To accomplish this, one of the things that they did was establish a set of firm standards that define what it means for

something to be considered open source [15]. According to “The Open Source Definition," hosted courtesy of The OpenSource Initiative, to be considered open source, a program must abide by the following principles:

1. It must allow individuals to redistribute commercially or free at their discretion and without restriction.
2. The program must be distributed along with its source code. The code must be in its authors ideal form and not purposefully confusing.
3. The author must allow others to modify and redistribute the work under the same license as the initial release.
4. Discrimination against any people, group, or application area is forbidden.
5. Ownership of the license must travel with possession of the source code so that constant modification and redistribution do not require modifications to the license.
6. The license must not be tied to one specific product. For example, if a plugin came with Houdini (www.sidefx.com) but could be modified to work within Maya (www.autodesk.com), it must be allowed to use the plugin outside of the context of Houdini. This does not mean that the plugin must function in all programs, it just refers to the rights to the license.
7. The software must not place restrictions on other software released in the same package (such as requiring all software in the package to be Open Source).
8. The license must not be tied to any specific technology or platform. In other words, it cannot be free only for iPhone users as opposed to Android users.

These guidelines were designed to both formulate concretely what Open Source development means, and also to function as marketing and outreach to the professional world. The standards are intended to make the legal benefits of implementing open source tools clear to the professional market and to allow them to access the many incredible production benefits that come with them [11].

The benefits of developing tools under this platform relate back to the scientific method. Computer *science* is at the root of all software development. Similar to the way that traditional science derives great benefit from the sharing of knowledge, software development gains several things from developing in an open environment [8].

For one, to make any scientific research robust, one must be able to replicate results. In computer science, there is only one way to truly replicate results and that is to share the source code. "One scientist cannot expect to account for all possible test conditions, nor necessarily have the test environment to fully test every aspect of a hypothesis. By sharing hypotheses and results with a community of peers, the scientist enables many eyes to see what one pair of eyes might miss. In the Open Source development model, this same principle is expressed as 'Given enough eyes, all bugs are shallow.' By sharing source code, Open Source developers make software more robust [8]."

Normally, a bug is discovered by a user and then forwarded to the developer in some way for it to be addressed as time allows (through email or a web based tracking system of some kind). The open source pipeline is somewhat different. Rather than a

bug being discovered by a user and forwarded to the developer, often the bug is discovered by an individual with access to the code and the ability to make the required adjustment themselves. This method of quality control would be very valuable in this plugin. On the main structure side, more eyes could eliminate bugs and improve the base logic of the program. This author might come up with a solid initial structure, but the product would benefit from the kind of development that would result from free exploration and the trying of new things.

The rig design phase would benefit from this idea as well. Rigs need to be tested to ensure that they work in as many situations as possible without breaking. Open source rig development would put many eyes onto the rig that was created and would hopefully provoke many changes as users from various backgrounds tried to use the rig in one manner or another and implemented improvements. Character rigs can be built in many ways, and many ways of handling problems can cause other problems or issues. Having many eyes on the stability of the rig is a major benefit to working in this environment.

The other major benefits that would result from developing under an open source license are the twin ideas of creativity and discovery [8]. Scientific development is not a linear/isolated process; instead, individuals benefit from what others have done before them. In the world of open source development, developers often leap frog from each other's successes and failures. Further, someone's success in one area can be implemented into a different area that benefits from the new research.

In 1998, Netscape announced that they were making Mozilla—their major product and the only main competitor to Microsoft's Internet Explorer—open source.

This was the first big commercial application to move to this model of development and it caused quite a stir. Netscape chose this route because they wanted to utilize one of the key benefits of the open source community—innovation. “Netscape looked to the community of independent developers to drive innovation and help them build a superior product [8].”

In his famous post discussing this event, Eric Raymond, one of the main motivators in the movement had this to say: “The Netscape announcement changes everything. We've broken out of the little corner we've been in for twenty years. We're in a whole new game now, a bigger and more exciting one -- and one I think we can win [14].” This memo set in motion the standards and definitions that these developers would live by and initiated a marketing campaign that is still running in high gear today.

“Open source software is a major part of the mainstream information technology economy, and it increasingly dominates aspects of that economy that will probably be the leading edge (in technological and market terms) over the next decade. There exist thousands of open source projects, ranging from small utilities and device drivers to office suites like OpenOffice, database systems like MySQL, and operating systems like Linux and BSD derivatives. Linux and Apache attract the most public attention. Apache simply dominates the web server market—over 65 percent of all active web sites use Apache....If you saw the movies *Titanic* or *Lord of the Rings*, you were watching special effects rendered on Linux machines that are running at companies like Disney, DreamWorks, and Pixar [22].”

And not only the operating systems are open source, there has been a remarkable

amount of development in the visual effects industry in regards to open source tools. The Gimp and Blender are two of the most mainstream applications although there are many others. Gimp is a photo editing, image manipulation tool that shares many features with Photoshop. In 1998, Zach Bene, the moderator of the Gimp's news website, declared that the Gimp was available and was a fully functional alternative to Adobe Photoshop. Further, he explained that the brilliance behind Gimp's success is that because it is an open source tool, if anyone see's a feature that they want from another program, they can add it themselves [19].

In October of 2008, Ryan Paul, a software reviewer for Ars Technica, wrote a product review of "The Gimp" called "Gimp 2.6 Released: One step closer to taking on Photoshop. [12]" In the review, he fully examines the software and decides that while it is not quite as polished as Photoshop, it is a viable competitor and growing quickly.

Blender 3D was born out of a visual effects studio. Ton Roosendaal co-founded NeoGeo—a Dutch animation studio. In 1995, he decided that they needed a 3D package that was designed from scratch for their purposes. This project resulted in Blender. Initially, a company known as NotaNumber was created to develop and market Blender while selling support and tutorial materials. This business model did not work well in a difficult economy so NaN shut down in early 2002—leaving Blender unsupported and undeveloped.

In May of the same year, Ton started the Blender Foundation as a non profit organization. NaN agreed that if he could raise 100,000 euros, they would release Blender as an open source program—a deal that they felt was a long shot. The open

source community provided the money in seven weeks and the Blender source code was released. In the current day, Blender is supported and developed by a team of developers from all across the world [3].

Some of the most modern open source initiatives in the visual effects industry come from Sony Imageworks, Industrial Light and Magic, and Disney. In August of 2009, Sony Imageworks announced that they were making five of their tools available as Open Source. In a letter on their new open-source website, Sony CTO Rob Bredow said "Open Source: It's always been a part of what makes Imageworks tick. However, up until now we haven't had a chance to contribute back to the open source community. We're beginning to change that now [4]."

In July of 2010, Sony announced that they were releasing Alembic as a joint effort with Industrial Light and Magic. Alembic is an open source file format for transferring files between programs. This creates a free and simple file format that studios and companies can implement into their software that will allow people to easily exchange data. Some of the companies that sent out supporting messages about this new open source format included Pixar, The Foundry, NVIDIA, and others [16].

Disney followed quickly after Sony's initial release of tools with a major release of their own. On January 15, 2010, Ptex, a revolutionary texturing application from Disney was released as open source under the BSD license. Ptex is a fully automated texture mapping system which greatly simplifies the texturing workflow. As of August 24, 2010, Ptex has been implemented in Houdini, 3D Coat, and several other tools. *Bolt*, a recent animated feature film, was the first feature film to implement Ptex and many

currently in development are utilizing it in their pipelines [21].

Open source is a powerful and interesting development model with a very practical application. It began as a niche idea that struggled to earn a measure of credibility but has grown into a mainstream development technique with real benefits. This thesis explores the idea that open source development could solve many of the development issues of a rigging pipeline tool.

2.3—The Rigging Process

When rigging a character, an artist makes use of many tools, the most common of which are bones. The function of bones within a character is similar to that of bones in a human--they provide pivot points in space for the rotation of both geometry and other bones. Bones in a character skeleton are connected to each other through parent/child relationships. Bones at the top of the chain (parents) rotate, and bones lower in the chain (children) follow. A good example of this is the leg; in Figure 2.5, the thigh bone is the parent of the calf bone. If the thigh bone rotates upwards--lifting the knee towards the chest--the calf bone follows. This occurs because the calf is the child of the thigh, and thus inherits its rotation. This is an example of "forward kinematics"—i.e. motion that follows the hierarchy [2].

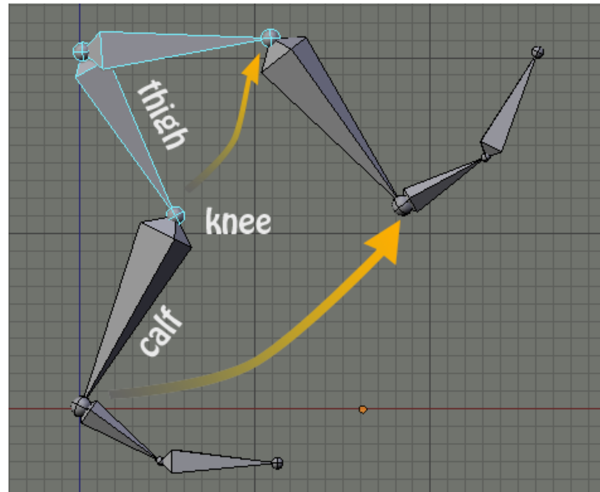


Figure 2.5: FK Articulation

FK animation is perfect if a character needs to stand in one place and swing their arms or other motions that do not require them to stick to any surface. But what if a person needed to walk by a table and plant their hand on it as they pass? It seems like a simple animation but it is actually very difficult to achieve through FK animation. This issue was the motivation for the creation of "Inverse Kinematics"—or "IK".

"Inverse Kinematics enables child bones, such as a hand or a foot, to control their parents [2]." With IK, an animator can grab a character's ankle, translate it, and the knee and thigh will rotate to compensate it. This allows characters to plant their feet as they walk, grab sturdy things with their hands, and accomplish any other animations that would require the child to control the parent.

IK animation is implemented through the usage of a type of "constraint." Constraints are mathematical expressions which are used to control the motion of bones.

A constraint can provide many different functionalities to a rig--from copying the rotation of one bone to another, to causing a bone to stretch to or aim at a target. In Figure 2.6, the blue bone has a "Copy Location" constraint targeted to the red bone. As the red bone is translated in any direction, the blue bone follows while maintaining its offset. There are many different kinds of constraints and they are named differently in every 3D package, but in each they prove to be the main creative outlet for getting the required performance out of a rig [2].

Controls provide a way for animators to interact with the character rig to create the required animation poses. Rather than leaving every part of a rig visible to the animator (for them to accidentally grab and break), a few select objects are left visible for them to animate and the rest of the rig is hidden [2]. Some general shapes are depicted in Figure 2.7. Control objects are usually given shapes which indicate what they do (circles for things which rotate or arrows for things that go one direction).

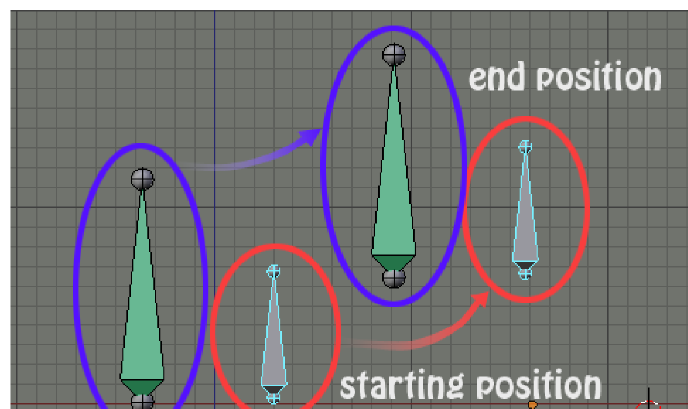


Figure 2.6: Copy Location Constraints

Copy Location constraints connect the transformation of one object to another object. In this diagram, the blue object moves

when the red object is translated.

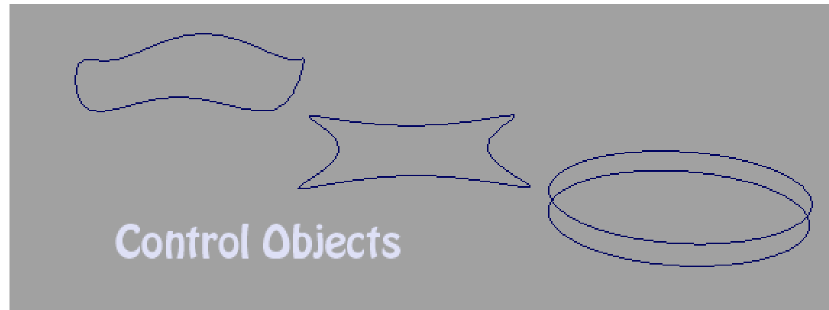


Figure 2.7: Control Objects

These are examples of shapes which are used as control objects in many character rigs. These objects are NURBS curves.

Utilizing these rigging tools and solid logic, character riggers can transform lifeless statues into characters capable of personality and life. Causing the character to come truly alive requires an animator to actually manipulate the character rig, but once the rig exists, the potential is always there.

III. METHODOLOGY

3.1: Choosing and Learning a 3D Package

Before in-depth work can begin on development, many basic descisions need to be made. The first major question was which package should be the focus of development. The chosen 3D package needed to fit some major criteria.

The first major requirement was the functionality of the package. Its rigging system needed to be somewhat traditional and completely functional. In addition, it needed to fully support Python scripting. The second consideration was the originality of the research as it related to the choice of package. Maya is the package of choice for most school-oriented research, but an important part of this thesis is originality. Whatever package was chosen, the results needed to be something which had not existed before and met a need in the user community. This leads to the third major consideration--the size of the user base. Rather than creating something just for personal use, the point of this thesis was for it to be picked up by a substantial userbase.

In the end, Blender was the obvious choice for a development environment. The user base of Blender is extremely large; most of these users are amateur, but they fit the needs of the plugin perfectly. As for functionality, Blender has benefitted a great deal from the open source movement and has many rigging features that surpass those available in other programs. In addition, very little rigging research has been done utilizing Blender. It was the perfect opportunity to develop something original.

Blender was the best choice for all of these reasons and more. It helped a great deal that Blender was actually an open source package upon which an open source tool

could function. In addition, this research coincided with a complete rewrite of Blender's user interface and of the Python API. The early demonstrations of the updated program demonstrated an incredibly powerful, yet lightweight program which could easily handle the most complex rigging tasks. Further, nothing at all had been written for this new version of Blender. Mr. Potato Rig was the very first fully functional script that was released for Blender 2.5.

While Blender does have many benefits over more traditional choices, it is different enough from these packages to require a change in rigging procedure. The Blender workflow differs from the traditional workflow in several ways. The first key difference is the concept of the Blender "object." As shown in Figures 3.1 and 3.2, all objects in Blender have multiple modes that allow different functionalities.

Polygon objects have edit mode/sculpt mode/paint-weights mode/ UV mode etc... Users navigate between the modes using the 'tab' key and if you want to modify an object you just drop into the appropriate mode. Character Rigs are contained in an object called an "Armature". On its own, an armature object is just an empty container with the potential of containing a character rig.

The armature object has 3 potential modes: object, edit, and pose. Object mode is the default mode and it does not allow for anything other than basic transformation of the object. The Edit mode allows the rigger to create and position bones without them affecting any bound mesh or without having to deal with constraints. Pose mode activates constraints and binding and is the mode that animation happens in. Users can switch between these modes at will. If they get their bones positioned and then start

constraining and animating but later realize they need to go back, they can drop into edit mode, shift the bones, and back into pose mode without it hurting the rig.

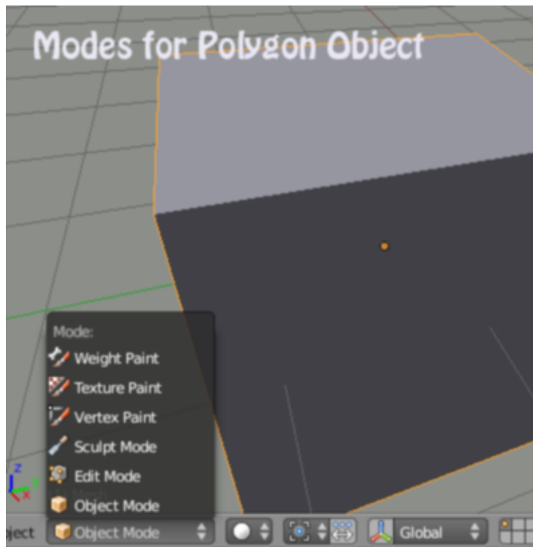


Figure 3.1: Polygon Modes

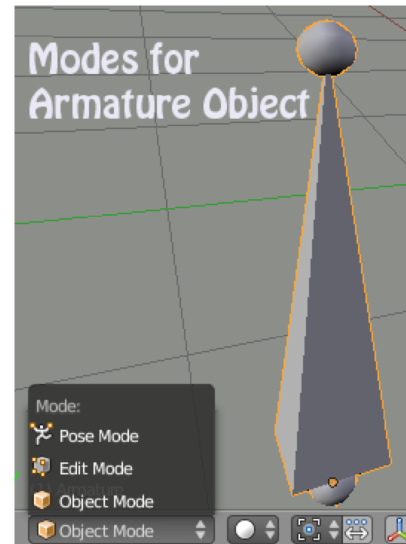


Figure 3.2: Armature Modes

Another key difference between Blender and Maya (although Houdini has the same idea), is that Blender uses bones instead of joints. In Maya, each rotation point is a joint which translates geometry via rotation (Object A in the Figure 3.4). The joints are chained together via parent/child relationship and this relationship is pictured above by object b1 and b2. In this diagram, b1 is the parent of b2. In Blender, all rigging is bone based. Each bone possesses a head and tail which would correspond to joints in Maya (the head is a1 and the tail is a2 in the Figure 3.3). This method has many advantages—including eliminating worrying that the joint orientation does not face down the bone the correct way, In Maya, a joint is a single point that has translation and orientation. In

Blender, a bone has two points as well as a roll property which give it translation/rotation/scale/roll/ and results in a much more stable setup.

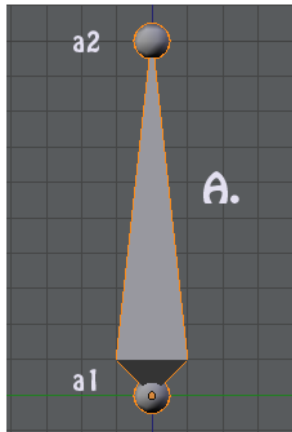


Figure 3.3: A Blender Bone

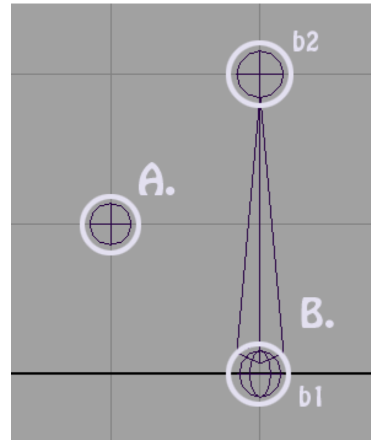


Figure 3.4: Maya Joints

Once the toolset was established and firmly understood, the design process began.

3.2: The Character Design Process

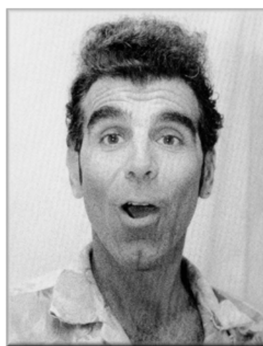


Figure 3.5: The Design Process

The character that was going to be the test subject for Mr. Potato Rig came from an abandoned short film project. Based on differing time tables, it never moved into full

production; however, several things were left over from the research phase. The first thing it gave this thesis was a style of animation that could be used to guide rig performance. Dr Seuss had been a key inspiration for the story and his cartoony style was the best place to start for the rig requirements.

Second, enough basic design work had been done on character design that he just required tweaking to prepare for modeling. The actual character inspiration came from a fusion of a Dr. Seuss and Kramer from “Seinfeld”. Along with this conceptual guideline, the designers received the facial reference photo from Figure 3.5. After several revisions, the concept design was complete and the modeling process began. After another several revisions, the final model was completed—in both high and low resolutions.

3.3: The Rig Design Process

Two main resources contributed to the search for a template for the final rig. Both of these sources served as both inspiration in rigging technique and education in the general usage of Blender. The first of the resources is Bassam Kurdali's *ManCandy FAQ* [10]. This resource is a rigging DVD that contains a final character rig and documents many of the steps involved in the rig's creation. This resource is a bit dated and not many things made it from ManCandy to the final rig, but it was a great inspiration and an excellent educational resource.

Another key resource was the exploration of existing chracter rigs. People approach rigging from many different directions and it was important to have a balanced approach to solving each problem. The Blender Foundation created a short animated film

called “Big Buck Bunny," and then made both the film and the assets used to create the film available for download or purchase. The *Big Buck Bunny Archive [20]* was a great resource of working character rigs. These rigs had the benefit of actually being tested in production and demonstrated techniques of rig design which worked in theory as well as “in the trenches.” The archive had several different character rigs and each of them gave valuable insight as to how many different problems could be approached.

The creation process began with learning Blender and building a rig. Using these resources and a lot of trial and error, eventually a rig was designed which had all of the necessary functionality and could be split up into its individual modules.

3.3a: The Legs and Feet

The design of the legs started with an evaluation of the personality and style of the character. As a “Seuss-like” cartoon, he needed a very stylized and exaggerated style of walking and animation. The legs received a basic IK/FK switch to let the animators switch back and forth between IK and FK based on the needs of the scene. This was enhanced by adding stretch to the IK. This allows the legs (and arms) to stretch as demonstrated in Figure 3.6.

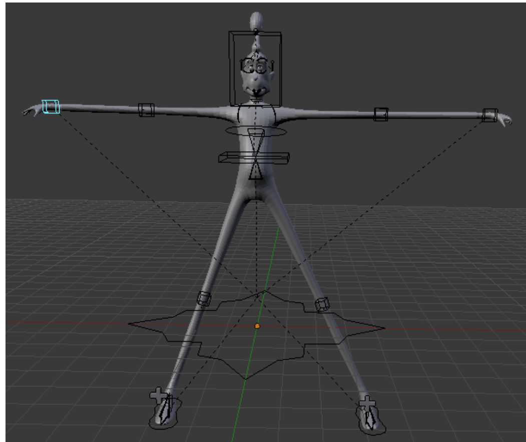


Figure 3.6: Stretchy Limbs

The stretch effect was accomplished through use of a special Blender constraint called "Stretch To." This constraint causes a bone to point towards its target and grow along its length axis to meet it wherever it goes. A major difficulty involved in implementing this feature was that the calves would inherit their parents growth when they stretched as well as stretch on their own. This type of problem is known as "double transformations." This happens because in a parent/child relationship, all stats are inherited by the children. The calf bones would grow when their parents did and then grow again for themselves. To get around this, the relationship was broken and replaced with a system of constraints which worked similarly to a parent relationship but without the double transformations.

Another key feature of a cartoony animation is the ability of limbs to bend (as shown in Figure 3.7). This feature is known as "rubber hosing," and was implemented through use of a curve deformer. The curve deformer modifies the shape of geometry

based on alterations made to a controlling nurbs curve. If the curve slopes in a direction, the geometry gently slopes as well. For this rig, each section that needed to rubber hose had its own curve parented to its accompanying bone. For example, the calf curve was the child of the calf bone. A controller was then provided that modified the shape of the curve.

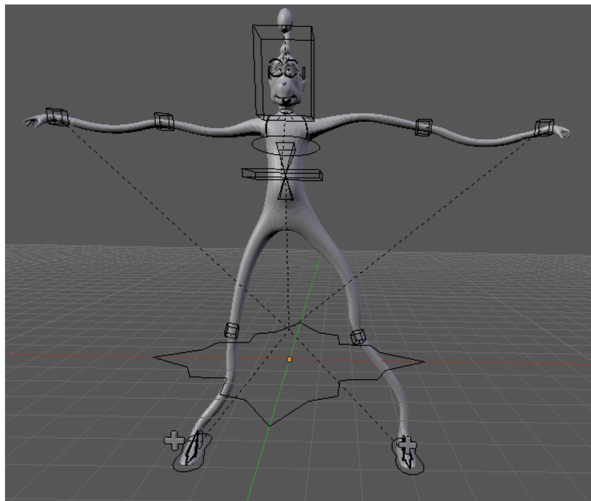


Figure 3.7: Rubber Hosing

The limbs have extra controls which allows them to bend and curve

The final limb feature that was implemented into the legs was knee correction. A pole vector determines the general bend direction of the leg, but an extra controller is placed at the knee which can be translated to tweak animation if required. This feature is demonstrated in Figure 3.8.

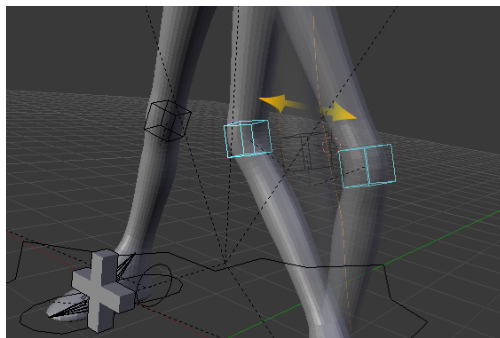


Figure 3.8: Knee Correction

For the feet, the rig utilized a standard reverse foot and fairly standard controllers. As demonstrated in Figure 3.9, the control off of the back of the heel controls the foot-roll. Rotate it forward and the foot rolls forwards; rotate it backwards and the foot rolls onto its heel. The remaining two foot controllers allow the animator to tweak these poses by pivoting the foot up or down and bending the toe.

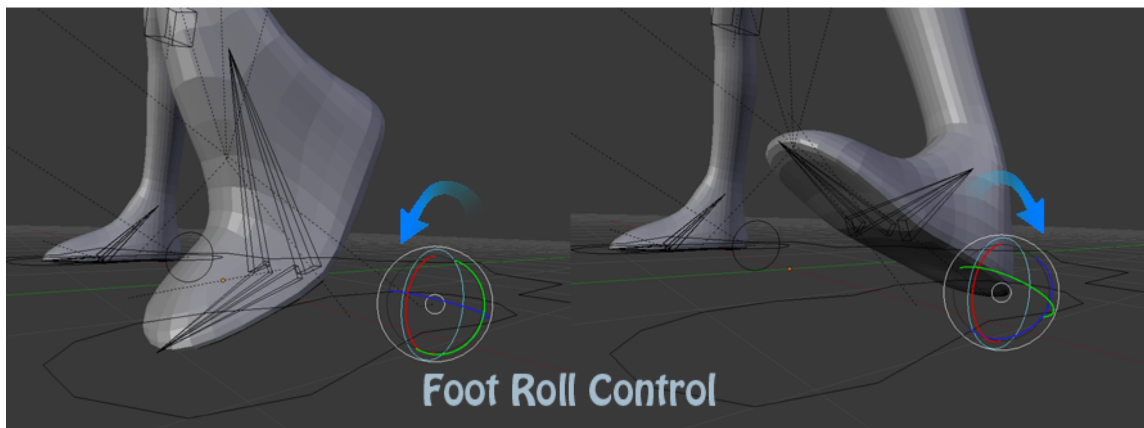
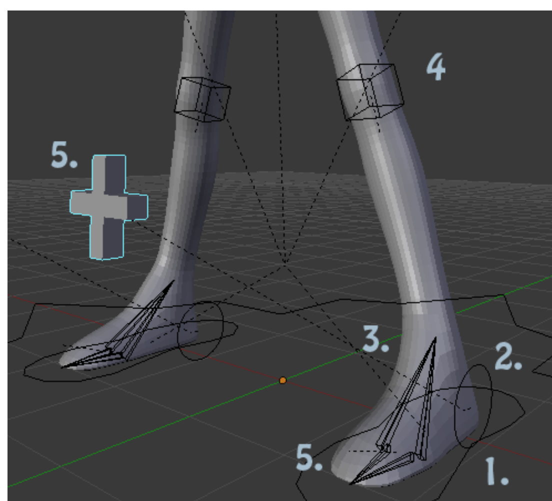


Figure 3.9: The Foot Roll Control

The final leg and foot rig is pictured in Figure 3.10.



1. Main Foot Control
2. Foot Roll Control
3. Ball Pivot
4. Knee Corrector
5. Toe Bend
6. Knee Pole Vector

Figure 3.10: Leg and Foot Rig Overview

3.3b: The Arms and Hands

The arms shared much of their structure with the legs. Both have IK/FK switches, stretch, and rubber hosing. There are two main areas however in which the modules are handled differently—the clavicle and the hands. In the human skeleton, any time the hand rises above shoulder level, it does it through rotation of the clavicle. Without clavicle animation, arm movement looks fake and broken because the arms will have a pinch effect—as shown in Figure 3.11.

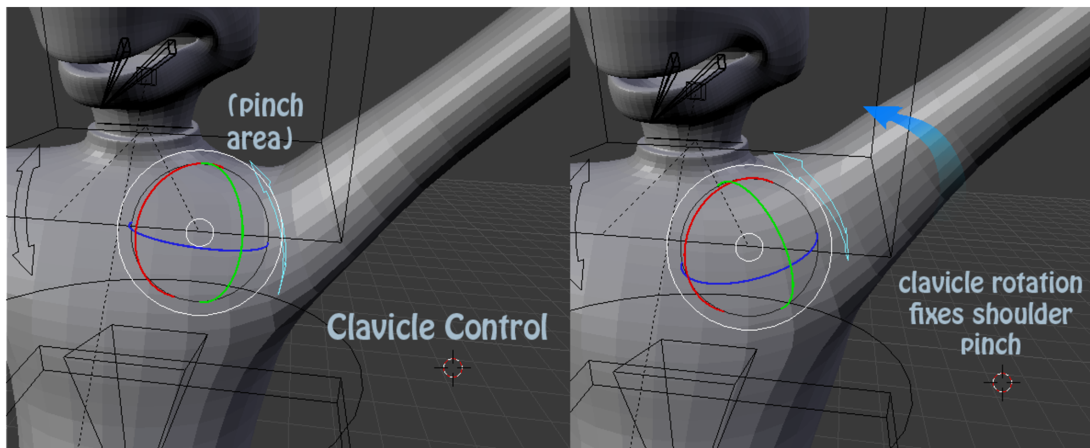


Figure 3.11: The Clavicle Control

Many people try to automate clavicle animation through expressions and other node based systems. These systems are often tricky and easy to break so this rig utilizes a simple FK corrective system. The animator animates the arms as needed and then compensates for the clavicles afterwards. This has the disadvantage of being completely manual but is very easy to do and will not break as easily.

The other main way in which the arms differ from the legs is the hands/fingers.

The *ManCandy FAQ* made a great start at demonstrating a really interesting finger rig. Bassam's concept was that the fingers would be controlled by a 3 joint IK chain. The IK target on the end is the child of a stick controller that sits over the finger. When the stick controller is scaled down, the finger rotates inwards [10]. Nathan Veghdal, from the Blender foundation team, took this functionality one step further by enabling the scaling up of the stick controller to curl the finger upwards [20]. In addition to this improvement, he accomplished the effect through expressions instead of constraints which is a much more reliable solution and worked great for both the fingers and ears. The functionality of the finger rig is demonstrated in figure 3.12. As the control scales, the finger curves down or up.

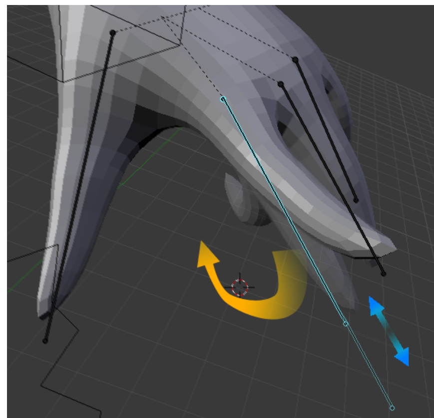


Figure 3.12: The Finger Roll Control

3.3c: The Spine

The spine tends to be the root of all control in a character rig and was tricky for this character for several reasons. The first reason is that none of the standard strategies for rigging the spine work in Blender. The two general methods in the industry are 'spline IK' and the 'ribbon spine'. Spline IK is currently an option in Blender, but only as

of July 2010. The ribbon spine (which is fast becoming an industry standard) is a very complicated setup which can be faked in Blender but it is a tricky setup and the rig needs to be simple and difficult to break.

The second consideration is the performance that the character rig needs to achieve. To stay consistent with the rigging philosophy for the arms and legs, the spine needs to have squash and stretch to it. Animation starts with the hips, and if the arms and legs are going to have this style, it should be motivated from the root.

Once again, the rig design has benefited strongly from a feature unique toBlender -- thee “bbone”. Bbone is an abbreviation for “Bezier Bone” or “Bendy Bone.” As the name implies, bbones can bend. Instead of linear rotation transformations being applied to vertices like would happen with traditional bones, bbone's allow the user to gradually curve the transformations like a bezier curve (see Figure 3.14). As is shown in Figure 3.13, each of the bbones comes with a 'segments' property which controls the smoothness of the curve and 'ease in/ease out' controls for in-depth tweaking of the effect. Not only does this allow the user to have bones that bend, it also allows graduation in twisting—such as in the forearm or calf. This results in spines that are drastically simpler than those created in Maya or other programs. Where a good spine in Maya has between 6 and 12 bones, a Blender spine has 3 (or sometimes two).

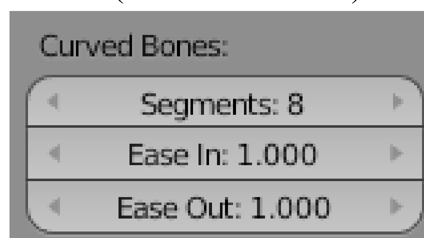


Figure 3.13: Bbone Properties

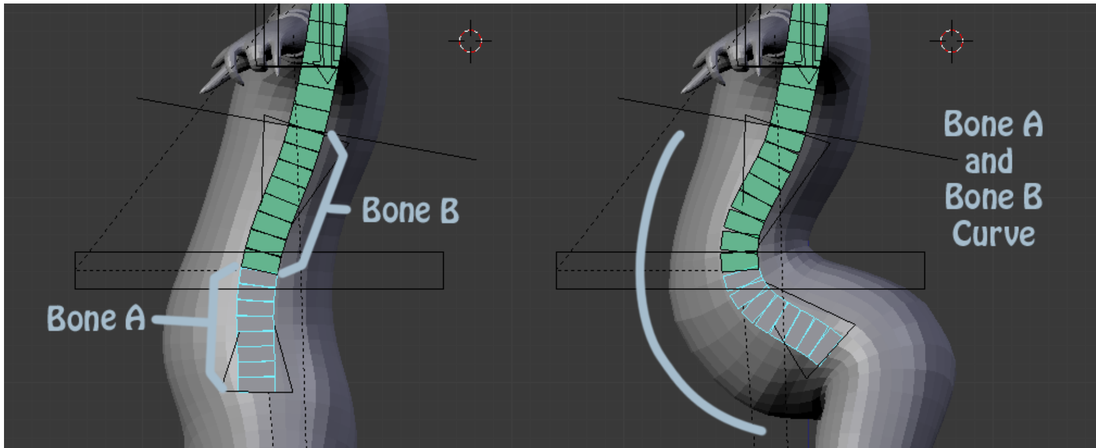


Figure 3.14: Bbones Curving.

For the stretch in the spine (as demonstrated in Figure 3.15), a "Stretch To" constraint was added to the second spine bone and aimed at the controller. The bbone attributes helped control things like twist and roll. To avoid double transformations, the bones above the stretch bone were attached to the stretch bone via constraint instead of through a parent/child relationship. Originally, the rig was designed with rib bones to help the character maintain volume, but it ended up that the style of this character was best suited to a single chain up the back. The final spine rig is displayed in Figure 3.16.

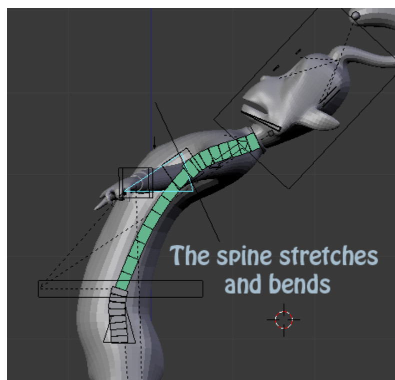
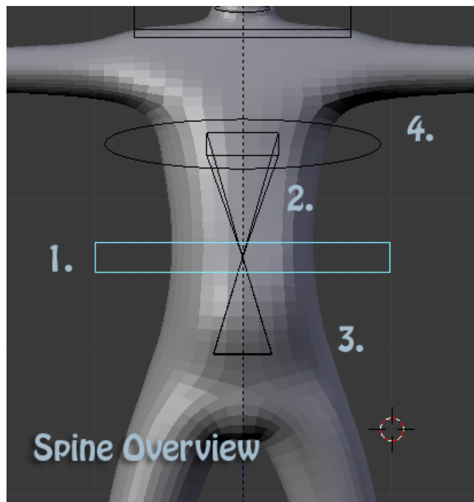


Figure 3.15: Bending and Stretching Spine



1. Main Torso Control

   --Controls everything except IK hands and feet.

2. Midback stretch/bend control
3. Hips Pivot
4. Upper Back Pivot

Figure 3.16: Spine Rig Overview

3.3d: The Face and Head

The face was implemented last, and right up till the end a facial module was not a part of the rig design. Typically facial setups are driven by "blend-shapes". The modeler sculpts the facial model into a smile, frown, or any other facial expressions that the animator might need. The rigger then ties these alternate expressions into the rig and the animator can animate back and forth between the expressions as needed. Because every face is different, automating blend-shapes would be a monster endeavor and would have questionable results at best.

As was pointed out by several individuals on the Blender forums, mechanical/deformation based facial rigs are an alternative method of setting up a face. Deformation based facial rigs accomplish most of their control through the clever usage of bones and constraints. This is not as efficient as a blend-shape driven facial rig but it

has the advantage of being completely compatible with the rest of the rig. The final facial setup is demonstrated in Figure 3.17.

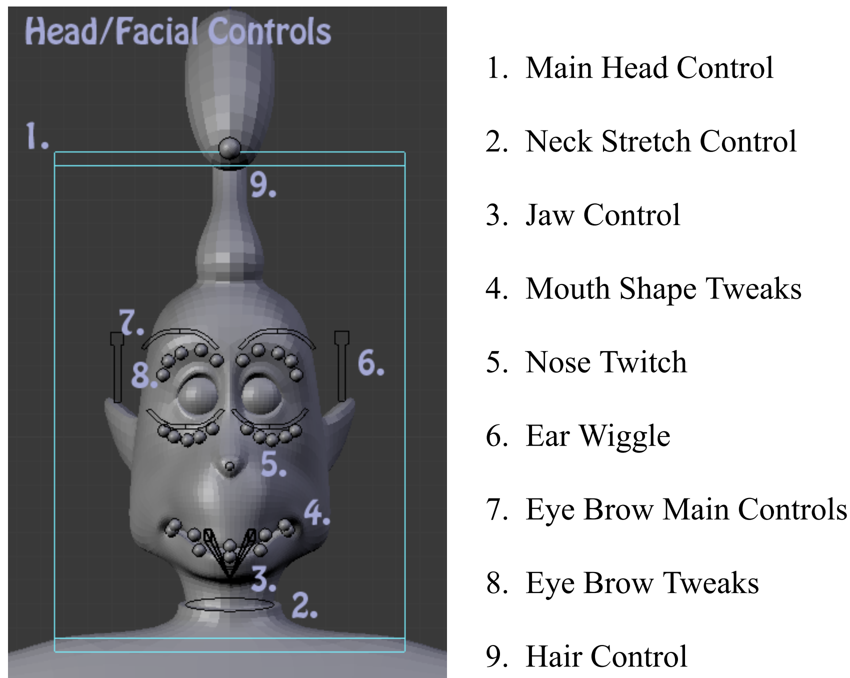


Figure 3.17: Facial Rig Overview

"Bendy bones" were used extensively for the mouth, eyes, ears, and nose. In Figure 3.18, the orange spheres are controls which are translated to deform the mouth. In between the spheres are bbones which have "Stretch To" constraints on them aiming at the appropriate control spheres. When a control sphere is translated, the bones bend smoothly allowing nice deformation as is shown in the diagram below. This setup is used for the mouth and then for around the eyes. The nose is controlled by one bendy bone with a stretch to constraint to a controller at the tip. This allows the animator to wiggle

the nose. The ears are designed to function like the fingers. When the ear controller is scaled, the ears wiggle back and forth. Finally, the neck has stretch functionality similar to the arms/legs/and spine.

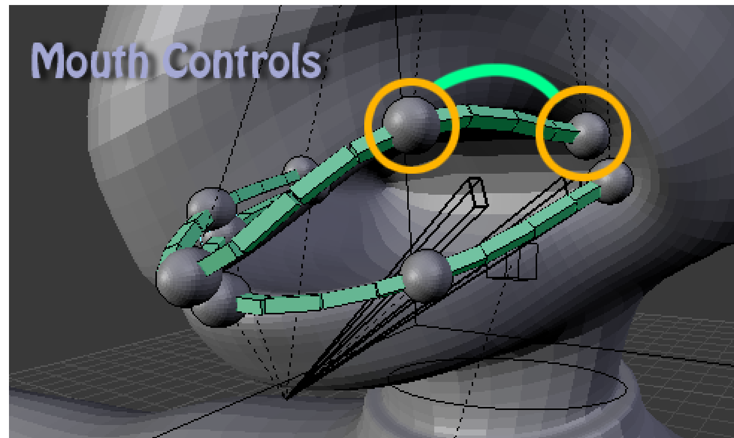


Figure 3.18: Mouth Shape Controls

For a full production facial rig, the facial setup would need many more options. Most of these options would need to be unique to specific facial features and would change depending on the character. Blend-shapes would eventually need to be involved at some level to hit some facial shapes that bones could not achieve. That said, this basic rig provides a lot of plenty of initial control and is incredibly easy for the user to animate.

3.4: Writing the Plugin

After the rig design was completed, the next step was utilizing Python to automate the rig creation. This was the most difficult challenge of the creative process. Blender's being in the middle of a complete redesign brought many positive things with it, but it

created one massive problem—every time the program was updated, all of the code was broken. The API was in constant flux and just as much time was spent fixing code that had worked previously as was spent writing new code. In addition, for most of the creation process, the API was mostly undocumented.

It was at this point that the open source community was of immense help. Blender has several IRC channels dedicated to discussion about developing for Blender. Most of the people populating these channels are the ones who support and maintain Blender. Individuals on these channels were often able and willing to answer any questions in regards to plugin development. They delivered a great deal of Blender specific advice as well as intermediate/expert python tips. There were many times when someone on the forums or IRC came through with a suggestion or a bug fix which solved a substantial problem.

Mr. Potato Rig's general design has two basic components—the main control panel (Figure 3.19) and the individual rig modules (Figure 3.20). The user types the rig modules that they want to be used in their rig into the control panel, and then they use the control panel to build the rig. Both of these components are closely tied together in design, but for the best understanding of the backbone logic, the individual modules are the best place to start.

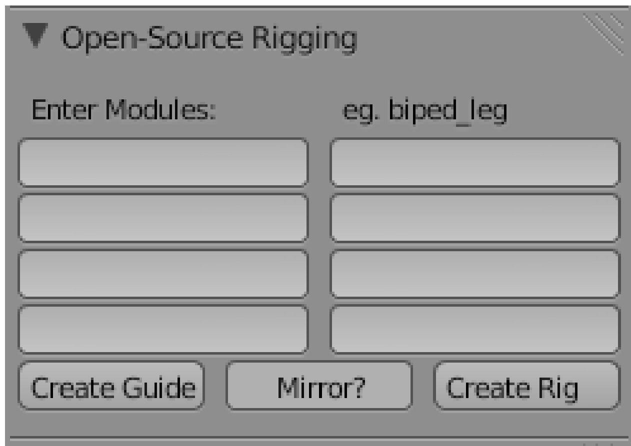


Figure 3.19: Main Control Panel

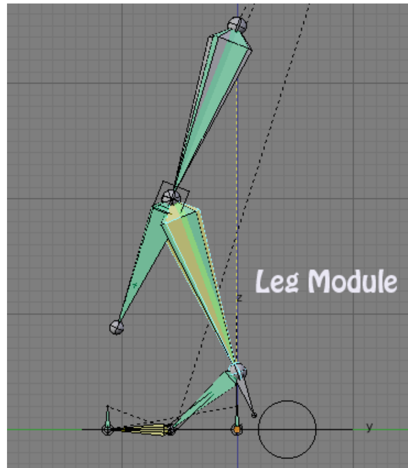


Figure 3.20: A Rig Module

This is an example of the leg module. A full cartoony biped rig includes 4 modules: Leg, Arm, Spine, and Head.

3.4a: The Modules

All modules are based around the core idea that the user only has to have two things in their script to make it compatible with the Mr. Potato Rig framework—a createGuides() function and a createBones() function. Within these functions, the user can approach creating their modules any way they wish. Not all rigs would benefit from identical structures and not all rigger's workflows would be similar. Leaving the riggers

with only these two requirements was intended to make the environment more open towards individual development. While this flexibility is valuable, this thesis also attempts to provide a guide for how things could be structured and some utility functions to make wrting modules easier.

In each of the provided modules, the createGuides() function follows the following process:

1) Examines a list of guide names and positions to see if the objects already exists.
2) If it does not exist, an appropriately named “empty” (a Maya locator) is created at each location.
3) If it does exist, the guide is moved back to the default position.

This function is fairly simple and involves basic object creation, naming, and positioning. Only the locations of the guides are important; the rotation and scale are not used. createGuides() does not need to be run from within the control panel to work but can just be called at the end of the script for testing purposes.

The createBones() function is a great deal more complicated and varies from module to module. The first thing that the function does is jump into edit mode on the armature object and create and position all of the bones for the module (using the positions from the empties created in the previous function). In Blender, almost every element in a rig is a bone—everything from control objects to IK targets and utility bones. Not every bone is necessarily visible in the final rig; for control bones, a rigger attaches a new shape to them so that they look like traditional controls. Other programs

make extreme usage of nulls and empty groups but Blender does not need this clutter. Because objects which are not bones cannot be easily brought into an armature object, everything is done with bones instead.

As it is creating bones, the script is using one of several “check” functions to determine if the bones already exist. The function returns a boolean value, and if it finds the bone it does nothing; if it returns false however, it creates the bone at (0,0,0) and names it appropriately.

Next, the following function is run on every bone in the armature inside the createBones function:

setupBone(newBone, headLoc, tailLoc, newParent, connected, deform, present):

a) newBone : A string that contains the bone name

b) headLoc: a vector location for the head of the bone

c) tailLoc: a vector location for the tail of the bone

d) newParent: a string containing the new bone's parents name

e) connected: A boolean value which determines whether Blender combines the head of the new bone and tail of its parent (so that they translate together in edit mode).

f) deform: a boolean value that determines if the bone deforms the mesh that the armature is bound too. With this as false, the bone is not a part of the weight table.

g) present: an integer value representing which layer the bone should be visible on. Typically control bones, bind bones, and utility

bones are on different layers so that the rig does not get cluttered.

The setupBone() method is called on every single bone in the rig. It sets up all of the basic details about where a bone falls within the armature. Often the locations come from the coordinates of the guide's setup in the earlier createGuides() function; in other cases, the locations are determined based on vector math.

Next, the module puts the armature object into “Pose” mode and creates all of the constraints for the module. Because of Blender's open source nature, it has a lot more constraint types than the most other 3D packages. Many functions that riggers need to set up through expressions in Maya come as standard constraints in Blender—making them quite easy to set up. Some of the key constraints used in the rig are “Stretch To” constraints. These were what were used to create stretchy limbs and a stretchy spine in the discussion about the design process. Essentially, a source bone has the constraint placed upon it and the constraint is given a target. The bone will then keep its head in the correct location but will stretch and rotate so that the tail follows wherever the target goes.

Some of the most common constraints are “Copy Location” (Point Constraint), “Copy Rotation” (Orient Constraint), and “Copy Scale” (Scale Constraint). These function identically to the way they are expected to function in most 3D programs. To apply the constraints in these modules, it was a bit more complicated than applying them via the UI. Traditionally, the user selects the driver object and then the driven object and then applys the constraint. Unfortunately, a combination of bugs and the fact that the constraints needed to be modified after they were created meant that a slightly more

complicated approach was required.

```
#SETS UP COPY LOCATION CONSTRAINT BETWEEN TWO BONES
def copyLoc(name):
	rig = bpy.data.armatures["Character_Rig"]
	driven = rig.bones[name]
	for bone in rig.bones:
		bone.selected = False
	driven.selected = True
	rig.bones.active = driven
	bpy.ops.pose.constraint_add(type='COPY_LOCATION')
	driven.selected = False
```

This is an example of a basic function which applies a constraint to a bone whose name is passed into the function as a string “name.” The function cycles through all of the bones in the rig and makes sure they are not selected. It then selects the bone which is to receive the constraint, makes it active, and applies the constraint. Finally, it deselects the bone.

The final step involves grabbing the constraint from the bone and modifying it so that it has the right settings.

```
	copyRot("knee-pinch_def" + suffix[i])
	con_Bone = armOb.pose.bones["knee-pinch_def" + suffix[i]]
```

```
for constraint in con_Bone.constraints:
        constraint.target = armOb
        constraint.subtarget = "calf_def"  + suffix[i]
        constraint.influence = 0.500
```

This is an example of code that sets up a copy rotation constraint for a knee-pinch bone to hover 50% of the way between the calf and the thigh (to relieve geometry pinching). It gets the name of the bone, cycles through the constraints on it (in this case there is only one) and sets the appropriate settings. The target and subtarget settings determine which bone has the rotation that the constrained bone will copy and the influence setting determines how much of that rotation it will adopt. By adopting 50%, The bone will provide a smooth transition for a knee.

Almost every constraint shares this type of workflow. Some constraints, such as IK constraints, have more settings that must be custom set for each constraint (chain length/pole vector/etc...). These extra settings go at the end and are written as follows:

```
constraint.chain = 2   #  Sets the length of IK chain
constraint.poleVector = 'knee_pv' + suffix[i]  # Sets which bone
        functions as PV
```

With adaptation by the open source community as a major goal from the beginning, it was important to write code that could be reused by others. The search for reusable code led to abstracting many different functions so that they can be used and applied in many different ways. The reusable functions that this thesis provides come in

several major categories.

Some of the main function scripts that the thesis supplies are basic vector math scripts. Rigging requires a lot of vector and matrix multiplication and these variable types and their functions are not naturally supported by Python. Because these would be needed often, dot product, cross product, normalize vector, and other assorted functions are supplied for basic use.

In addition, several functions are provided which use the above vector math to mathematically determine specific positions, rotations, and scales for controllers and bones.

setupPoleVec(loc1, loc2, loc3, name, suffix)

a) loc1 – loc3: Vectors which contain the positions of the joints to use to determine the new object's position.

b) name: a string new object name

c) suffix: a string containing "_R" or "_L" to determine whether the Pole Vector is on the Right or Left side of the body.

The above script takes in three vectors ([Hip, Knee, Ankle] or [Shoulder, Elbow, Wrist]) and uses vector math to determine the appropriate location for a pole vector (so that it can be applied to an IK chain without twisting it). The function than creates a bone with its head at that vector and its tail a small distance down the same vector. The function will use the varying sizes between the input vectors to determine a scale factor

which will be used to compute the length of the new bone. For example, if the length between the wrist and elbow and the elbow and shoulder averages out to 1.5, the pole vector bone will be 2 times that distance out (3 Units), and ¼ of that distance long (.375 of a Unit).

Sometimes a vector needs to be repositioned and rotated based on an offset origin. For example, if the script needs to triangulate 3 bones to find their normal and then use that normal for joint orientation, the computed location will have to be rotated 90 degrees around the normal in order to be used. This function was provided and utilized several times.

The final step in the modules was to set the bone shapes for the control objects. Unlike in most programs, Blender control objects are based on polygons. Normally this would result in a control object which would both obscure the character while animating and then would render along with the character, but in Blender, any object designated as a "bone shape" is automatically displayed in wireframe and will not render. To create a control object shape, the script would involve the following:

1) mesh = bpy.data.add_mesh("controllerName")

mesh.add_geometry(number of points, number of edges, number of faces)

- The first line adds a mesh object and the second initializes the geometry within the object. Because the mesh is going to be viewed wireframe regardless, the number of faces can be zero.

2) mesh.verts[0].co = [x, y, z]

- This would be called for every vertex in the mesh as declared in

the previous line to set up all of the points locations.

3) mesh.edges[0].verts[0] = 0

mesh.edges[0].verts[1] = 1

- Each edge has a start and end verticie. The final integer corresponds to a position in the mesh.verts[x] array. 0 would equal mesh.verts[0] and 1 would equal mesh.verts[1].

After a module includes all of the previous functionality, it has everything it needs to create a rig for one specific body part. Creating a framework for all of these functions to work together required a great deal of experimentation before the right system could be designed. The main control panel is where all of the modules come together to create a full character rig. It should also be noted that the main control panel is the only area that the community would likely avoid changing too greatly because breaking it would make most of the existing tools inert.

3.4b: The Main Control Panel

Several things needed to be accomplished with the design of the control panel:

1) It needed a way to logically gather module names and create their corresponding rigs.
2) It needed to possess a clear and easy to use GUI that would make the plugin as simple as possible. It should not be cluttered with a million different tools and options that could make it slightly more powerful but

less simple.

3) It needed to be able to handle multiple scripts and connect them or individual scripts and leave them seperate.
4) It needed to possess all of the “Top Level” functionality
5) It needed to be as fool proof as possible

Because the general workflow idea is to gather module names, create guides, and then create the final rig, the design process begins with creating a structure for dealing with module names. The first challenge in this idea is how to gather and store user input. To do this, Mr. Potato Rig makes extensive use of Blender's basic GUI design library. In the end, a simple collection of text fields/check boxes/ and buttons was utilized to move through the steps.

To gather user input, the tool panel was split into sections and columns using available functions (to help place the objects more precisely) and then calls were made to the appropriate GUI functions to create the interface (which can be seen in Figure 3.21).

Figure 3.21—Control Panel Breakdown

a. This is a demonstration of a “text label” and does not directly gather feedback from the user. It is created by calling--

row1.label(text="Enter Modules:").

b. These fields are text input boxes and they gather module names from the user. These fields are an example of 'property fields' and are created by calling: row2.prop(context.scene, "*variable to store string,*" text="*default text for field*").

c. The buttons are an example of a GUI operator field and are set up by calling rowOps.operator("*guides*," text = "Create Guides"). The *guides* string defines which operator function will be called when the operator is activated and the text field declares what is printed on the button. In this case, the operator *"guides"* will be called when the "Create Guides" button is pushed.

d. This final element is a property like the input boxes with one dramatic difference. Rather than gathering any data, it just gathers a boolean value. If activated, it will return true when queried and if not activated it will return false. It is created by calling: rowOps.prop(context.scene, *booleanVariable* to store value, text=*stringVariable* for default text, toggle = True)

Implementing the functionality was the bulk of the work and required many iterations until the basic design logic worked with all of my varied goals and was fairly break proof. All of the block names that the user enters into the text fields are captured into a python list when the "Create Guides" button is pressed. The empty fields are

ignored, and the script then dynamically loads each remaining module and runs it's createGuides() function.

Dynamically importing modules is a fantastic ability of Python and is one of the things that made writing this plugin possible in the first place. The important function is __import__(string) and it functions exactly as if you had placed an import statement at the top of the code. For example, the line “mod = __import__('sys')” will import the sys module and store it in the object 'mod'. Functions from sys could then be called by invoking mod.*function()* [13].

Given a list of strings containing module names from the text fields, the following code would import each module and run its createGuides() function.

```
for mod in modules:
    rig = __import__(mod)
    rig.createGuides()
```

The function cycles through the list of strings, imports each module, and calls the appropriate function. Without dynamically loading libraries, the logic of the program would have been much more complex and much more difficult to make simple for the open source community.

Once the user has the guides in the correct position, they select whether the final rig should be mirrored, and then hit the “Create Rig” button. The first thing this operator does is check to see if two required elements exist in the scene—an armature object and a rig object. If both of these objects are found and are properly set up the script does nothing, if they do not exist it creates them, and if they do exist but are not linked or

named correctly, it sets them up correctly.

Once the scene is set up, the function uses the same list of modules that were imported when the guides were created and goes through each of the modules like it did to create the guides. The only differences are that it calls the createBones() function instead of the createGuides() function and it passes in the value from the mirror toggle. If the mirror toggle is true, any blocks which have a mirror option will be mirrored across the positive X.

Using Mr. Potato Rig, riggers can construct a fully functional cartoony biped rig. The functionality is present as was originally designed. The interface is fairly minimalist but the tool's capabilities are exactly as intended.

IV. EVALUATION

Since its initial release, Mr. Potato Rig has received valuable feedback and constructive criticism from several individuals within the Blender community—typically through forums, email, or IRC. The final release provides a tool that fits a specific need in the Blender library of tools. It is a solid plugin with good implementation and features.

There have been, however, several draw backs. The main problem has been the constantly changing Python API. For two months (March and April of 2010) the tool was kept 'live' on the forums so that it would work with the most recent builds of Blender. This required constant updating and fixing as each new update broke the scripts. While the new interface has amazing new features and the design is one of the best of the current generation of 3D programs, at that point it was not quite ready for full scale plugin development. Every change that the developers made improved the interface of the tool, and waiting six months would have made the process much smoother.

Because of this issue, the open source development did not quite happen to the extent that was originally intended. Solid feedback was received from the community on the character rig and the scripts, but not many people spent a lot of time actually tweaking things on their own. It is likely that if the environment had been more stable, than it would have been a different story. That said, the plan is to update the code now that the API is finalized and and benefit from the communities' input.

There are several features that could improve the functionality of the plugin. First, instead of having the user position guides for the rig placement, having them position a low resolution skeleton would be much more practical. The guides can be very

difficult to see and in tight areas—such as fingers—it is difficult to get the positioning exactly right. This would be a basic switch but would require some restructuring on the back end (code would have to be changed to query bone coordinates instead of locator coordinates).

On a related note, the ability to go back and forth between the “guide” and “rig” stages of rig design is an important feature. At the moment, once the user hits the “Create Rig” button they can not go back and easily change positioning and proportions. The ability to go back to “guide mode” would be a simple addition that would really improve the user experience.

A fairly major design flaw in the current release is that the user is limited to one armature object per rigging scene. Most 3D packages seperate data by having each asset (character, prop, etc...), in its own scene, and then the user will “reference” in whatever assets they need for a given shot. Most films done in Blender, however, seem to contain all of their assets in one core shot. To accommodate this, some of the code would need to be abstracted to identify different armatures.

The final design improvement deals with how the modules interact with each other. At the moment, the linking of modules occurs via hard-coded parent relationships. The clavicle looks for a specific spine bone to make its parent and, if it finds it, the relationship is created. This method requires a user to be able to read Python if they want to restructure their rig. The “hot swap” functionality that is one of the base ideas of the script should have a more abstract functionality. That said, “Rigify”, a rigging tool by the Durian team, was released shortly after “Mr. Potato Rig” and it does possess a more

abstract way of linking modules. Their implementation and workflow is powerful but complicated, and it is generally described as confusing for most users. With the hard-coded method, it is a bit less flexible, but it is very simple for the user.

V. CONCLUSION

The initial release of the "Mr Potato Rig" provides many options to the user. For those who would like to utilize the initial modules, a cartoony biped is available. For character riggers who would like to script their own modules and characters, the design lends itself towards expansion by others. In addition, many utility functions have been provided which can greatly ease the workload of new riggers.

In the near future, Mr Potato Rig will be fully updated and compatible with the final release of Blender 2.5. In regards to additional features, the first step will be to address the deficiencies listed above. Many of these features will be simple to implement and will not require a great deal of rethinking of the base logic. The point of the tool is that it be used on projects and adapted by them. If this can happen, development will take on a momentum of its own as teams adapt the tool and it evolves to meet their needs. To get to this point, it needs several of the above features to make it show ready.

In addition to new features, the plugin needs more modules. The intended goal is a full character body part library. Right now, it has a solid beginning for cartoony biped body parts but this is a very specific type of character. To extend functionality, important development needs to be done in regards to quaraped and avian blocks. The addition of bird wings and quadruped legs would allow a great deal of flexibility in available creatures to be rigged.

Based on modern FX software development, it appears that the open-source model is an ideal method for supporting tools for a show of any size. The Blender community has a passion for their platform, and the support they provided was amazing.

Mr Potato Rig is a viable model for an open source modular rigging platform, and in the near future it will be a fully functional tool, capable of helping users of many different skill levels to navigate rigging inside of Blender.

VI. BIBLIOGRAPHY

[1] “Advanced Skeleton.” Animation Studios. 2010 <http://www.animationstudios.com.au>

[2] E. Allen and K. L. Murdock. *Body Language: Advanced 3D Character Rigging*. Indianapolis, Indiana: Wiley Publishing. 2008.

[3] Blender Foundation. “History.” <http://www.blender.org/blenderorg/blender-foundation/history/> 2009.

[4] M. Broersma, “Sony Opensources Digital Effects,” ZDNet. 2009. <http://www.zdnet.co.uk/news/application-development/2009/08/03/sony-open-sources-digital-effects-software-39703959>. (accessed 9 September 2010)

[5] W. Carlson. “A History of Bell Labs.” *A Critical History of Computer Graphics and Animation.* 2007. <http://design.osu.edu/carlson/history/tree/bell.html>.

[6] W. Carlson. “A History of CGRG/ACCAD at The Ohio State University.” *A Critical History of Computer Graphics and Animation.* 2007. <http://design.osu.edu/carlson/history/ACCAD_overview/overview3.html>

[7] J. Dave, B. Hiebert, T. Kim, I. Neulander, H. Rijpkema, and W. Telford. “The Chronicles of Narnia: The Lion, The Crowds and Rhythm and Hues.” *SIGGRAPH '06 Course Notes. 2006.*

[8] C. DiBona, S. Ockman, and M. Stone, *Open Sources: Voices from the Open Source Revolution*, O'Riley, 1999.

[9] P. Elmer-Dewitt, L. Mondi, and J. Prim, “Computers, Software is for Sharing.” Time. 1984, <http://www.time.com/time/magazine/article/0,9171,926733-1,00.html> (accessed 27 August 2010)

[10] B. Kurdali, *ManCandy FAQ.* The Blender Foundation. Amsterdam: Netherlands.

[11] Open Source Initative. “Open Source Guidelines.” <http://www.opensource.org/docs/osd>. (accessed 25 August 2010)

[12] R. Paul, "Gimp 2.6 Released, One Step Closer to Taking on Photoshop." Ars Technica. <http://arstechnica.com/open-source/news/2008/10/gimp-2-6-released-one-step-closer-to-taking-on-photoshop.ars> (accessed 2 September 2010)

[13] M. Pilgrim. "Dynamically Importing Modules." 2004. <http://diveintopython.org/functional_programming/dynamic_import.html> (accessed 13 September 2010).

[14] E. S. Raymond, "Goodbye Free Software,Hello Open Source," 1998, <http://www.catb.org/~esr/open-source.html>. (accessed 25 August 2010)

[15] E. S. Raymond, *The Cathedral and The Bazaar.* Thyrsus Enterprises. 2009. <http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/index.html#catbmain>.(accessed 27 August 2010)

[16] "Sony and ILM release Alembic, an open source graphics interchange format." The H Open. July 28, 2010. < http://www.h-online.com/open/news/item/Sony-and-ILM-release-Alembic-an-open-source-graphics-interchange-format-1046729.html> (accessed 7 September 2010)

[17] J. Smith and J. White, "BlockParty: Modular Rigging Encoded in a Geometric Volume." *SIGGRAPH '06 Sketches.* 2006.

[18] R. Stallman, "Why Free Software is better than Open Source", <http://www.gnu.org/philosophy/free-software-for-freedom.html>. (accessed 25 August 2010)

[19] M. Stutz, "A Thrifty Photoshop Built for the Web," Wired. 1998, <http://www.wired.com/science/discoveries/news/1998/03/10975>. (assessed 2 September 2010)

[20] N. Veghdal, *Big Buck Bunny Archive.* The Blender Foundation. Amsterdam: Netherlands.

[22] S. Weber, *The Success of Open Source,* Cambridge: MA, Harvard University Press, April 2004.

[23] D. Zeltzer. "Towards an integrated view of 3-D computer animation." *The Visual Computer.* 1985

[21] Walt Disney Animation Studios. "Ptex: An Overview." January 2010.