


5-2014

An Automated Build System for Articulated Characters

James Sledge

Clemson University, bensledge3d@gmail.com

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

 Part of the [Communication Commons](#), [Computer Sciences Commons](#), and the [Film and Media Studies Commons](#)

Recommended Citation

Sledge, James, "An Automated Build System for Articulated Characters" (2014). *All Theses*. 1958.
https://tigerprints.clemson.edu/all_theses/1958

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.

AN AUTOMATED BUILD SYSTEM FOR ARTICULATED CHARACTERS

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
James Benjamin Sledge
May 2014

Accepted by:
Dr. Timothy Davis, Committee Chair
Dr. Jerry Tessendorf
Dr. Sophie Jörg

Abstract

Rigging is the process of designing and implementing the manipulation architecture for an animated three-dimensional character. Rigs that give the animator the most control tend to be the most difficult to set up and maintain. Due to the linear nature of some elements of rigging, the more complicated a rig, the more time-intensive—and therefore more expensive—to achieve a high quality rig. A solution to complex rig iterability is to automate as much of the process as possible. The topic of this thesis is a framework for modular rigging automation, with a focus on quick and efficient rig iteration. A rigger is able to design a rig from predefined module elements (rig blocks) or quickly script new blocks. A rig is deconstructed into these elemental blocks and merged into a single rig script to regenerate the rig and attach a character's geometry.

Acknowledgments

I would like to thank the amazing people in the Digital Production Arts program, especially my adviser, Dr. Davis, and my committee, Dr. Tessendorf and Dr. Jörg. Without their instruction and guidance (and deadlines) this project would never have been completed. Animation is a collaborative process and I am grateful that Alex Beaty and other DPA students graciously allowed me to experiment with characters from their production. The constant feedback loop that exists in the DPA lab is the perfect environment in which to perfect new ways of doing things.

Next, I would like to thank Robert Helms (of DreamWorks Animation) and Chase Cooper (of Industrial Light and Magic) for allowing me to pick their brains over the summer and at SIGGRAPH in 2013. Their willingness to answer what questions they could about rigging and digital production in general provided the inspiration for this thesis.

Finally, I would like to thank my family and friends (especially Mandy, Gina, Skye and Jarrett) for helping me get through the marathon that is graduate school. Without their constant support and encouragement I have no doubt I would have given up on this grad school thing all together.

Table of Contents

Title Page	i
Abstract	ii
Acknowledgments	iii
List of Figures	v
1 Introduction	1
2 Background and Related Work	3
2.1 A Brief History of Animation	3
2.2 Computer Graphics	4
2.3 Autodesk Maya	8
2.4 Automatic Rigging	10
3 Implementation	13
3.1 Blocks	14
3.2 Layering Blocks	23
3.3 Rigging a Character	23
4 Results	30
5 Conclusions	34
Appendices	36
A Code Examples	37
References	48

List of Figures

2.1	Zajac’s <i>A two gyro gravity gradient altitude control system</i>	4
2.2	Plotter drawings from Fetter’s representation of human figures in plane cockpits.	5
2.3	A 3D model of Catmull’s hand in “Halftone Animation.”	6
2.4	Burtnyk and Wein’s skeleton technique used on human legs.	7
2.5	Zeltzer’s motor control techniques.	7
2.6	Industrial Light and Magic’s BlockParty 2 interface.	11
3.1	NURBS curve control shapes created by the Ctrl class.	15
3.2	The ribbonIk block in a Maya scene.	18
3.3	InlineOffest controls (shown in blue) added to a ribbonIk block.	19
3.4	Joint chains with IK splines applied.	21
3.5	Rigging spec sheet for the Pirate Captain character.	24
3.6	Storyboards showing the Pirate Captain’s movement.	24
3.7	The outer tentacle rig.	25
3.8	The inner tentacles after creating the RibbonIk blocks.	28
3.9	The joints and control shapes for the jellyfish’s lower bell.	29
4.1	The full Pirate Captain rig with all controls activated.	31
4.2	The fully posed Pirate Captain rig.	31
4.3	Lower bell range of motion tests.	32
4.4	The outer tentacles hair simulation.	32
4.5	Sea anemone rigged using the ManDynHair block (rigging and animation by Alex Beaty).	33

Chapter 1

Introduction

At its most basic level, computer animation can be simplified to using a computer to draw images on a screen. When these images are displayed in quick succession, they give the illusion of movement. In 3D computer animation, these images—referred to as frames—are generated by tracking a series of points, referred to as vertices, in the xyz coordinate space, and projecting those vertices onto a two-dimensional viewing plane. These points are typically combined into sets of polygonal faces that comprise the 3D representation of an object, referred to as a model, mesh or geometry. The core concept of computer animation is that these vertices and polygons can be manipulated on a frame-by-frame basis to create the performance of a character. To facilitate this manipulation, a hierarchical skeleton of “joints” is threaded into the geometry. The vertices that comprise the mesh are “skinned,” or “bound,” to one or more joints that can be positioned by translating, rotating and scaling them to achieve a desired shape.

Rigging is the process of designing and implementing the manipulation architecture for an animated three-dimensional character. During this process, a hierarchal system of movement and articulation control for the animator is built into a 3D model. Like all elements of the animation production pipeline, rigging requires constant refinement through multiple iterations. Rigs that afford the animator the most control are often the most complex and difficult to set up, and therefore more difficult to iterate and improve upon.

Due to the linear nature of some elements of rigging, the more complicated a rig, the more time-intensive—and therefore more expensive—to precisely recreate each element of the process for every iteration.

A solution to complex rig iterability is to automate as much of the process as possible. The topic of this thesis is a framework for modular rigging automation, with a focus on quick and efficient rig iteration. A rigger is able to design a rig from predefined module elements (rig blocks) or quickly script new blocks. A rig is deconstructed into these elemental blocks and merged into a single rig script to regenerate the rig and attach a character's geometry.

This thesis starts with an overview of the history of computer animation and other notable automated rigging solutions in Chapter 2. A detailed examination of the proposed framework follows in Chapter 3. An example character is rigged using the suggested method with the final control system depicted in Chapter 4. Finally, Chapter 5 discusses the conclusions and future work.

Chapter 2

Background and Related Work

The importance of rigging, and more specifically automated rigging, can be understood by examining its historical context. Ultimately, the goal of rigging is to better facilitate the animation of a digital character; thus, exploring developments in animation and computer graphics is valuable. The industry standard animation and production tool, Maya, is explored as a platform for a rigging system. Finally, this chapter analyzes contemporary automated rigging schemes.

2.1 A Brief History of Animation

The idea of representing motion with a series of images dates back to human-kinds' early days. Since early cave drawings, humans have had a fascination with recording movement in fixed drawings. These primitive attempts at motion synthesis eventually evolved into the optical toys of the 1800s. The Thaumatrope, Phenakistiscope and Stroboscope were popular novelties of the day that allowed an individual to view a series of images in rapid progression, thus creating the illusion of motion [Atk08]. In 1892, French artist Charles Émile Reynaud first debuted his technique for the projection of hand-painted gelatin squares. His invention transformed simple parlor tricks into public entertainment by allowing the animation to be viewed by a larger audience. His "Theatre Optic" would eventually give

way to more sophisticated cinema techniques that more closely resemble modern technology [Atk08].

In 1891, Thomas Edison invented the kinetoscope—a motion picture viewer—and launched a new film industry. In 1908 French artist Émile Cohl produced a short piece that is considered to be the first animated film, *Fantasmagorie*. In the early 1900s, animation technology continued to advance. John Bray and Earl Hurt patented the use of translucent cels and a peg-based registration system to aid in animation production. In 1915, Max Fleischer, from Bray’s studio and the creator of Betty Boop, patented the technique of rotoscoping—animating by tracing an image. The 1920s and 1930s were dominated by advancements from Walt Disney. Disney’s innovative multi-plane camera revolutionized the art of animated film-making with the ability to use parallax and rack focus in hand-drawn films. Nothing, however, has had as great an influence on the way modern animation is produced as the rise of computer graphics [Par12].

2.2 Computer Graphics

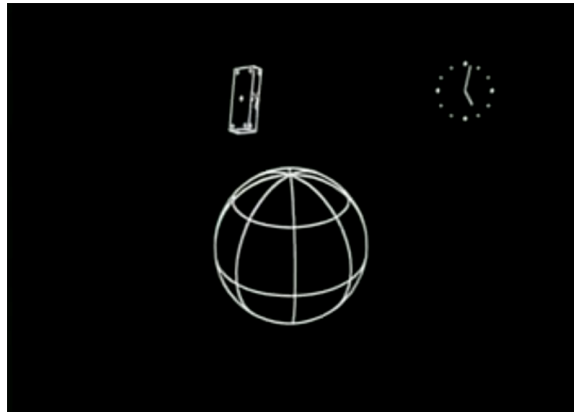


Figure 2.1: Zajac’s *A two gyro gravity gradient altitude control system* [Dre12].

In 1961, Edward Zajac and Bell Laboratories produced the first computer-generated film titled, *A two gyro gravity gradient altitude control system* [Car07]. Figure 2.1 shows a frame from the generated animation [Dre12]. The film was produced as part of the simulation

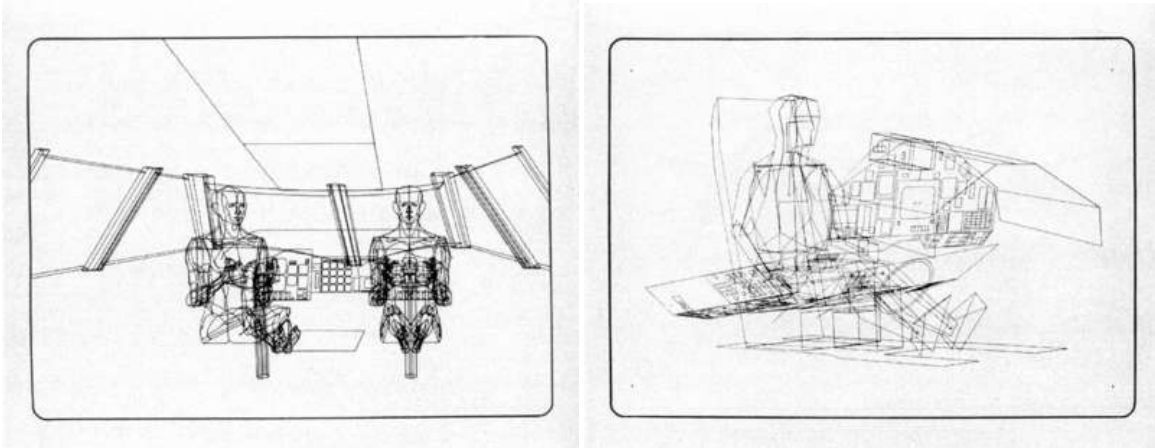


Figure 2.2: Plotter drawings from Fetter’s representation of human figures in plane cockpits [Fet14].

of a satellite orbiting Earth, and depicts a method of using gyroscopes to orient the satellite such that a given side is always pointed toward the planet [Lew64]. A few years later, William Fetter (who coined the term “computer graphics” [Cus11]) created the first computer model of the human form [War97]. These human figures were little more than wire-frame line drawings [Fet14] (Figure 2.2), but were still shown in art exhibitions at the time [Dre12].

The first computer generated-animation used in a feature film was Ed Catmull’s “Halftone Animation” (Figure 2.3). Created in 1972 at the University of Utah, Catmull and Fred Parke’s experimental animation of Catmull’s hand closing and rotating to point at the camera was the first to focus on three-dimensional objects moving in space, and also included a rendering of an animated face [Dre12]. The 1976 film “Futureworld” used clips of Catmull’s hand on computer monitors during a scene. The Library of Congress added Catmull’s animation to the National Film Registry in 2011 [Mea11].

The early days of computer animation closely resembled traditional hand animation. An artist would draw a series of key-frames and use various algorithms to generate the “in-betweens.” This system saved the animator the time of drawing the intermediate frames; however, the animator was forced to redraw the image for every key-frame, and could only control the interpolated motion by the number of frames between two key-frames [Bur75]. To give artists more control, techniques for path animation were developed, which allowed



Figure 2.3: A 3D model of Catmull’s hand in “Halftone Animation” [Mea11].

animators to better influence the motion of the in-betweens. This rudimentary control works well for 2D animation, but 3D characters often include hundreds of degrees of freedom, thus making fine-tuned adjustments impractical.

In 1976, Nestor Burtnyk and Marceli Wein of the National Research Council of Canada further improved upon this technique by proposing that a skeleton be placed around parts of a drawing. This skeleton could then be animated like a stick figure, and would deform the final drawing based on the skeletal motion (Figure 2.4) [Bur76]. The idea of using a low-resolution representation of a more complex shape was one that would be extended into three dimensions by David Zeltzer, Norman Badler, John Chadwick and others [Zel82][Bad87][Cha89]. Zeltzer’s system for motion control (Figure 2.5) proposed a series of hierarchical state machines that allowed animators to set the state at each level of the control structure for a key-frame, and directed the computer to interpolate between the states [Zel82]. Badler suggested an inverse kinematics solution where goal position was key-framed and the skeleton would move towards a position that satisfied those goals [Bad87]. Chadwick’s animation prototype, Critter, used a layered approach in which model deformations were layered to create the final look. The animator would pose the model’s skeleton, which would drive dynamic deformations for muscles, and add a skin layer to perfect the final pose [Cha89].

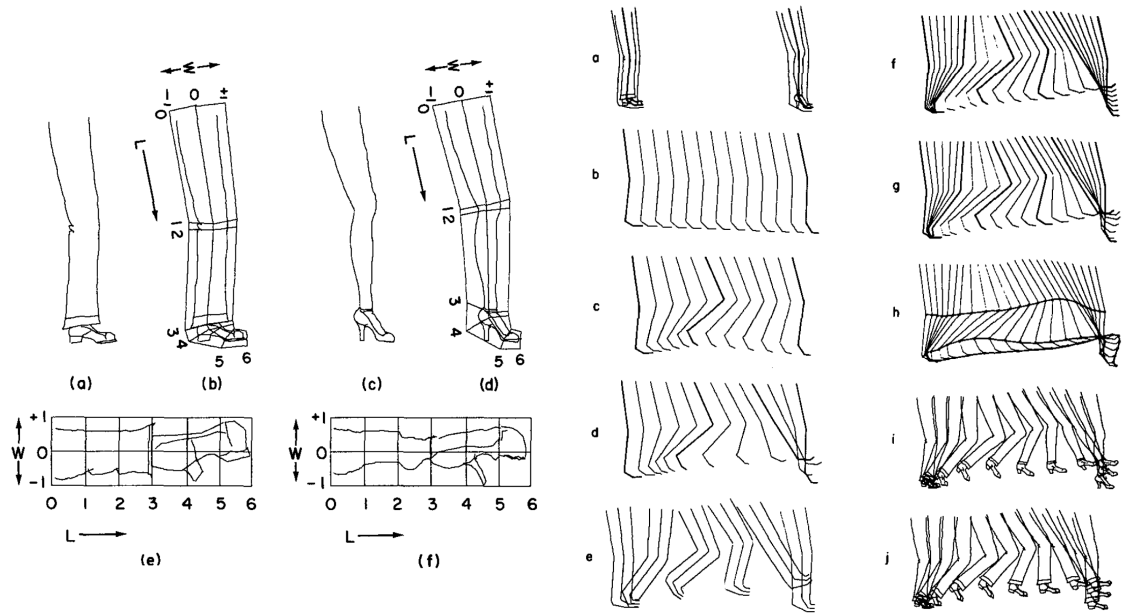
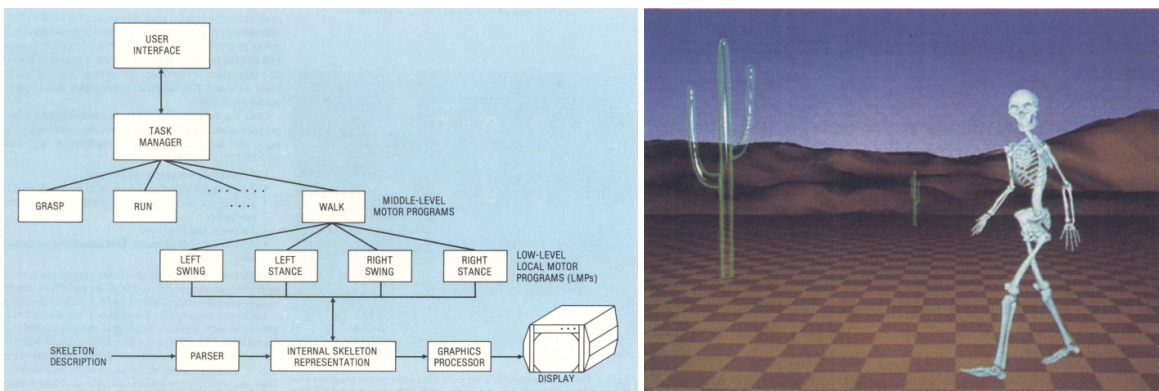


Figure 2.4: Burtnyk and Wein's skeleton technique used on human legs [Bur76].



(a) Zeltzer's structure of motion control programs. (b) A frame from an animated walk with a detailed model.

Figure 2.5: Zeltzer's motor control techniques [Zel82].

Since the 1980s, several motion control systems have been developed for animated characters. These systems allow artists to manipulate digital characters via motion capture [Bod97], dynamic simulation [KA03], physical puppet controls [Num11] and direct rotational joint control [McL11]. For artistic applications, most systems have evolved into a three-tiered approach. First, an underlying motion system comprises a character’s skeletal joints. Second, a control system is added to allow an animator to control the movement of the character. Third, a deformation system is introduced that defines the way the final geometry is manipulated [O’N08]. Modern rigs are an amalgamation of these three components [McL11].

2.3 Autodesk Maya

A major commercial animation software package on the market today is Autodesk Maya [Aut14a]. Maya has become the industry standard for modeling and animation. Its robust tool set and extensible embedded scripting language and API make it an attractive option for studios around the world.

2.3.1 History

Alias Research came to the forefront of artistic computer graphics in the 1980s with its PowerAnimator software. In an effort to prohibit Microsoft from entering the entertainment software sector, hardware maker Silicon Graphics, Inc., (SGI) purchased Alias and Wavefront Technologies (makers of The Advanced Visualizer) and renamed the new subsidiary Alias|Wavefront [WSJ95]. Microsoft had just purchased Softimage, the leading animation software company at the time [Boz95], though Softimage would later be acquired by Autodesk, Inc., in 2008¹ [PRN08]. The new company combined code from Alias’s Power Animator and Wavefront’s The Advanced Visualizer and named the new software Maya. The first official version of Maya was released in 1998 for SGI’s IRIX and Windows NT machines

¹In March 2014, Autodesk announced that Softimage would be discontinued after the April 2014 release of Autodesk Softimage 2015 [Aut14b].

as a competitor to Autodesk’s 3D Studio MAX (later renamed to 3ds Max) and Softimage [Gia98]. In 2003 Alias|Wavefront received an Academy Award for Maya’s impact on the visual effects and animation world [Pla03]. In 2004 SGI sold Alias to technology equity firm Accel-KKR [Tel04], and after a year of operating on its own, Alias was sold to Autodesk [Rob05]. Today, Autodesk bills Maya as “comprehensive 3D animation software” with a robust feature set including tools for modeling, rigging, animation, surfacing, dynamics and rendering [Aut14a].

2.3.2 Animation and Rigging

Everything in Maya is built around a dependency graph node network called a scene. Nodes are chained together with connections to create and manipulate geometry in 3D space. Inputs to nodes are called attributes, most of which can be keyframed (and are sometimes referred to as channels). Nodes that have a visual representation in 3D space are called objects. Some objects include child shapes that share attributes with their parent objects. Objects that occupy a location in space are termed transform nodes—sometimes abbreviated “xform.” A special class of transform, called a locator, is typically used for display purposes only [Aut11].

Rigging in Maya usually starts with an underlying skeleton structure comprised of joints. Motion systems and control objects are then applied to the joints. Control objects are non-rendering objects—typically NURBS curves or surfaces—that visually represent the rig to the animator. Attaching the joints to geometry is a process called skinning, which assigns influence weights for each joint to every vertex. Maya provides three methods for skinning: rigid bind (where vertices are weighted to only one joint), smooth bind (where vertices may be weighted to any number of joints), and dual quaternion skinning (which is similar to smooth bind, but uses a more advanced algorithm for volume preservation). Joints that are used in the skinning process are sometimes called bind joints, and are typically grouped together in a hierarchy, referred to as a “joint chain.”

2.3.3 Scripting

One of the greatest advantages of the Maya software package is extensible scripting and API support. Since its first release, Maya has included an embedded scripting language, MEL (Maya Embedded Language). MEL’s C-like syntax allows users to add features and automate tasks, but the language is relatively rudimentary and does not allow programmers to use external libraries. To address the growing need for a more robust scripting environment, Python support was added. Maya’s implementation of Python provides access to the vast collection of libraries and utilities available to programmers, and thus improves upon MEL. At the lower level, however, the Maya Python API only wraps calls to MEL functions and does not add many features where interaction with scene objects is concerned [Mec12].

In an attempt to expand the scripting capabilities in the Maya environment, PyMel was developed by Luma Pictures and later released under an open source license by Pipeline Supervisor Chad Dombrova [Lum14a]. The project focuses on making object-oriented Python scripting in Maya easier and more readable. One major advantage, from a coding standpoint, is that PyMel’s node architecture is not tied to a string-based object name in the Maya scene. Traditionally selecting objects through a script could only be accomplished via a search for the object’s name (which can be changed outside the script by a user at any time). PyMel removes this concern and allows scripts to reference a Maya object directly, without regard to that object’s name [Lum14b], as part of the official commercial release of Maya since 2010.

2.4 Automatic Rigging

The current trend in modern rigging is automated procedural setups. Automated work flows reduce user error, improve efficiency and provide a consistent interface to animators across multiple characters and shows. The initial inspiration for this project was Industrial Light and Magic’s (ILM) procedural rigging tool, BlockParty 2. The goal of the BlockParty 2 team was to build on their previously successful rigging tool, BlockParty, by

adding a 2D node editor. In both BlockParty systems, rigs are decomposed into modular parts, which are stored in a library through which rigging teams are able to quickly compile them into completed, animator-ready rigs [Smi06][Ros13]. The node editor in BlockParty 2 helps rigging artists visualize the way the rig is constructed by providing a more detailed user interface [Coo13]. Figure 2.6 shows the BlockParty 2 interface [Ros13].

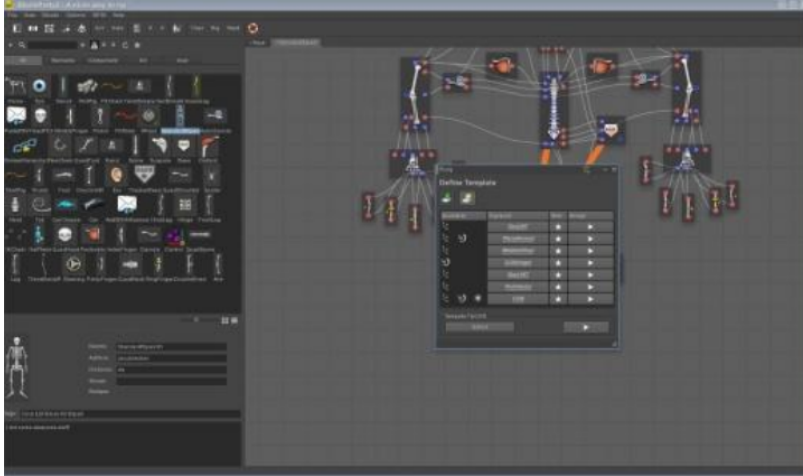


Figure 2.6: Industrial Light and Magic’s BlockParty 2 interface [Ros13].

Walt Disney Animation Studios used a different approach with their dRig system, a scripting pseudo-language that abstracts the code behind their build system. Like ILM, Disney’s solution requires modular rig pieces that are assembled to form a character; however, instead of locking riggers into a visual node structure, dRig allows rigging artists to design their rigs via a series of text commands. These commands are then compiled into code that can be run in Maya to generate the final rig [Smi12].

Other automated rigging solutions have been proposed. Péter Borosán, et al., developed RigMesh, a technique that integrates rigging into the modeling phase of production. An artist uses a sketch interface to design a model that is generated with a built-in joint hierarchy [Bor12]. Placing joints during the modeling step is an idea with merit; however, the method of automatically placing joints (or assigning the task to modelers) typically yields poor placement and introduces more problems later in the pipeline [Hel13]. Lawson Wade

created a method of automating anatomically correct joint placement [Wad00]. While his technique seems to be more accurate than Borosán, et al., it does little to address the creation of an animator-friendly control system. Ryan Cushman created an automated rigging plug-in for the open source animation software, Blender [Cus11]. While similar to the blocks system presented here, his solution is limited to a bi-pedal humanoid.

All rigging systems, whether automated or manual, exist to better extend control to an animator. This thesis proposes a system of rigging articulated characters that builds on the works mentioned here. With a solid understanding of the history and principles of animation, as well as an awareness of computer graphics and previous attempts at automating the rigging process, the framework proposed in this thesis can be examined.

Chapter 3

Implementation

Due to the repetitive and iterative production process involved with creating 3D animated films, a rigging solution was needed that would allow rigging artists to quickly test new ideas without arduously recreating previous work. The designed system should be usable in current and future pipeline setups, as well as easy for rigging artists to understand and apply.

The implementation of the system discussed here is similar to Walt Disney Animation Studio's dRig [Smi12]. First, a rigging artist creates a rig script by importing the rigging module and creating block objects. The build script is then run and the rig is created in the Maya scene. Since Maya provides extensive support for Python programming and the PyMel project allows intuitive access to the Maya API, Python was chosen as the language for this project.

In this chapter, a description of the proposed system is provided, followed by an explanation of each rigging block. Finally, an example character rig is generated using the instructions provided by the blocks in the system.

3.1 Blocks

Like ILM’s BlockParty systems [Smi06][Ros13], this tool is built around the idea of modular rig pieces called blocks. Each block is derived from a base block object in Python and contains the instructions that build a rig segment in the Maya scene. Each block object is used to track the Maya nodes the block creates in the scene. Since block attributes can be considered pointers to Maya nodes, blocks can theoretically be edited without affecting the actual nodes Maya evaluates.

All blocks follow the same general structure. On instantiation they accept parameters that are used to build the block’s nodes. Each block also contains attributes that point to any objects it created in the scene, which allows artists to re-use certain nodes created for other blocks. Finally, every block implements a build method that runs the code to create the nodes that comprise the block in the Maya scene.

To create a block and build the requisite nodes in the Maya scene, the block object is instantiated as in the following:

```
# Import the blocks module
import blocks

# Create a new block
b = blocks.Block(name='blockName')
```

The attributes associated with that block can be accessed using Python’s dot notation. For example, the name of a block can be acquired thusly:

```
# Print the name of the block
print b.name
```

3.1.1 Controls

Providing a visual method for animators to interact with a rig is an important part of every system. In Maya, control objects are typically created from non-rendering NURBS

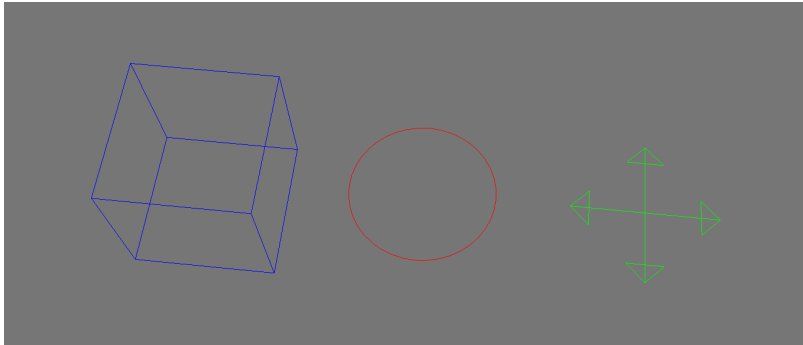


Figure 3.1: NURBS curve control shapes created by the Ctrl class.

curves or NURBS primitive geometry, for ease of creation. While not actually a block, the Ctrl class was created to provide a consistent interface for creating control shapes. The Ctrl class defaults to creating a NURBS circle control shape, but the class supports box and arrow type control shapes as well (Figure 3.1). The class contains the following parameters:

- xform: a transformation node which indicates the location of the control shape
- name: the name of the control object
- shape: the shape of the control object
- radius: the size of the control object
- normal: a three-tuple that indicates the axis around which the control shape is oriented
- group: a boolean that indicates whether the control shape should be inside a null-transform group
- color: the color of the control shape
- lock: a list of attributes to be locked.

To instantiate a control shape, the Ctrl constructor is called with the desired parameters. The control itself can then be accessed from the “ctrl” attribute on the Ctrl object.

This class provides an interface for control object creation for the block in the framework. It allows standard control object types and configurations to be defined once for the entire system, instead of redefining the appearance of a control object in each block.

3.1.2 Base Block

The base block is the object from which all blocks are derived. It serves as the template for the rest of the blocks. The base block has six attributes (construction parameters are marked with *):

- `name*`: a string used to name all the nodes created by the block
- `controlRadius*`: a decimal or float that sets the size of the control objects created by this block
- `controlColor*`: an integer value from 0 to 32 that sets the color of the control objects created by this block
- `controlShape*`: one of a set of predefined constants that determines the shape of the control objects created by this block
- `jointPostfix*`: the postfix appended to the name of the joints created by the block
- `bindJoints`: a list of any joints created by the block to be skinned to the character's mesh
- `controls`: a list of any control objects created by the block.

The base block class also contains a `build` method, but the method simply raises a Python `NotImplementedError` message.

3.1.3 Other Blocks

A rig system, like the one proposed here, is only as good as the rigs that can be built with it. The following blocks were created for this project and comprise a library of

rig pieces that can be compiled into full character rigs. The full source code is available in Appendix A.

3.1.3.1 Fk

The foundation of a good character rig is a forward kinematics (FK) system. FK controls give the animator direct control over the rotation values of the joints and provide the most straightforward control over the animation. The FK block takes a joint as a parameter, and creates an FK controller for each joint in the hierarchy under it.

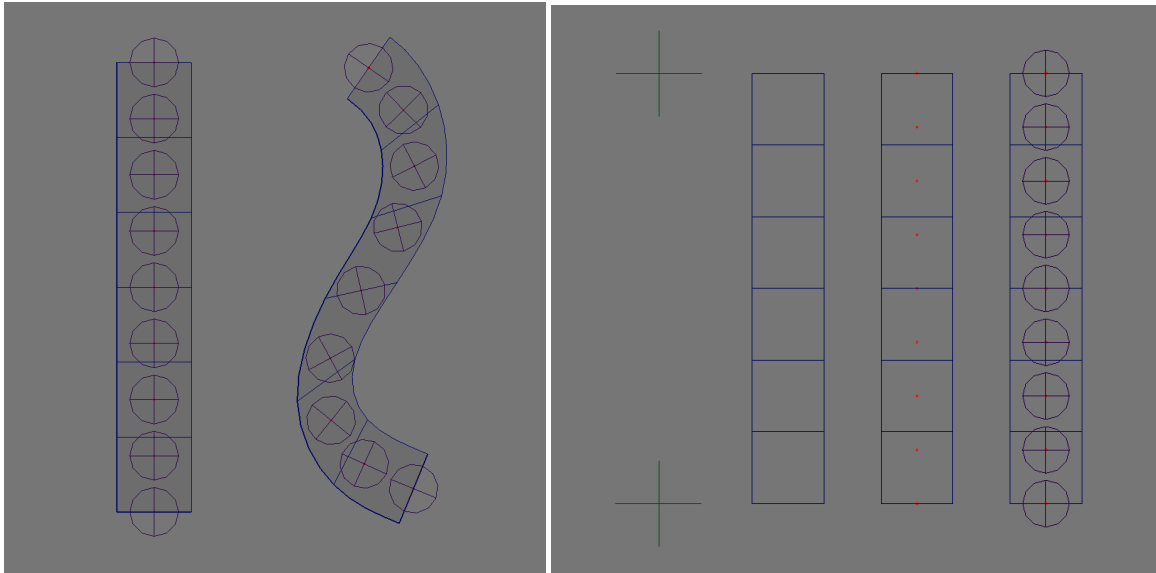
3.1.3.2 RibbonIk

The RibbonIk block was designed for tails and tentacles, but could also be used for the spine of a humanoid or other character. The block creates bind joints that remain distributed between controls that can be translated and rotated into position. Figure 3.2a shows an example of a finished block. An undeformed ribbon is shown on the left, with a deformed ribbon on the right.

In addition to the attributes inherited from the base block, the RibbonIk block adds the following attributes (construction parameters are marked with *):

- startLoc*: a locator at the starting point of the RibbonIk
- endLoc*: a locator at the ending point of the RibbonIk
- numSpans*: the number of spans and control objects associated with the ribbon
- numJoints*: the number of evenly spaced bind joints along the ribbon
- ribbonIkPlane: the NURBS surface plane created by the block
- follicleGrp: the group containing the created follicle nodes.

In its build method, the block creates a NURBS plane and positions it between the start and end locators. Follicles are placed along the plane using the UV information built



(a) (Left to right) An un-posed ribbon IK joint chain and a posed joint chain.

(b) (Left to right) The input locators, NURBS plane, follicles, and bind joints.

Figure 3.2: The ribbonIk block in a Maya scene.

into the NURBS surface. Joints are then parented under each follicle, which allows them to follow the position of the surface. Figure 3.2b shows a deconstructed view of the final block. This method is based on [Jos10].

3.1.3.3 InlineOffset

The InlineOffset block was designed to create a control object around a single joint. This type of control is useful when the rigging artist needs to create offset controls—that is, controls that directly manipulate the joints bound to the mesh. Animators use offset controls to finesse the silhouette and performance of a character. For example, controls on a ribbonIk block could be used to define the rough shape of a limb, then offset controls could be used to further refine the position of specific joints in the pose (Figure 3.3). The only extra parameter this block requires is the joint that will be affected by the control. The block then creates a control object and inserts it directly into the hierarchy above the joint.

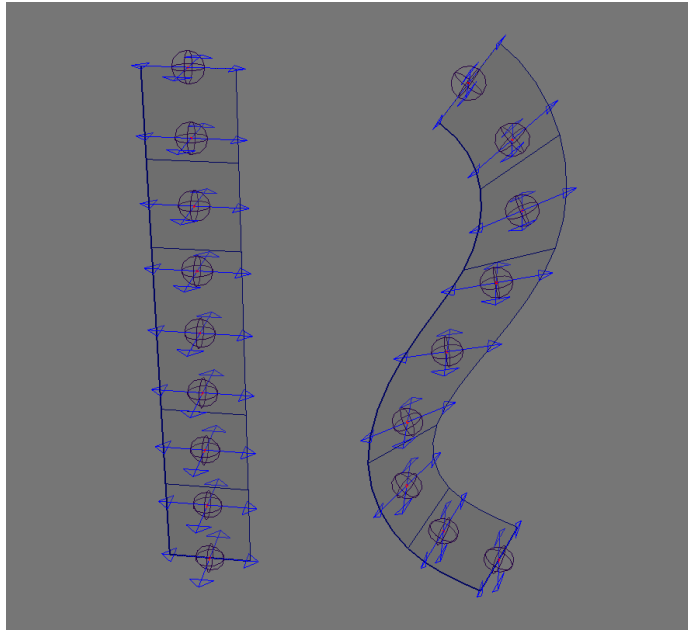


Figure 3.3: InlineOffset controls (shown in blue) added to a ribbonIk block.

3.1.3.4 LinearSkin

At times creating control shapes that are directly attached to a piece of geometry is useful, especially in a layered rig approach where one piece of geometry will guide another. The LinearSkin block creates such a system. In addition to the parameters inherited from the base block, the LinearSkin block inputs the following:

- mesh*: the mesh that will be skinned for this block
- startLoc*: a locator at the starting position for the joint chain the block code will create
- endLoc*: a locator at the end position of the joint chain
- numControls*: the number of control objects to be created.

In the Maya scene, the LinearSkin block creates a linear joint chain starting at the startLoc position and ending at the endLoc position. It attaches the created joints to the

geometry specified by the mesh parameter. Finally, it uses the InlineOffset block to create control shapes on each joint.

3.1.3.5 IkSC and IkRP

Inverse kinematics (IK) is useful when an animator needs to set the position of the end of a joint chain and have the rest of the joint chain automatically fall into place. For example, an animator sets a character's feet at a certain position and allows the legs to automatically follow, based on the goal set by the IK system's end effector. The IkSC and IkRP blocks both create IK handles from the start and end joints passed as parameters. The IkSC block uses Maya's "Single Chain" IK solver, which attempts to solve for the position and rotation of the end effector (allowing the system to be completely controlled from a single control object). The IkRP uses Maya's "Rotate Plane" IK solver, which only attempts to match the position of the end effector. A second object, or pole vector, is used to set the rotation of the rotate plane system [Aut11].

3.1.3.6 IkSpline

Maya's IK spline handle constrains joints such that they follow a curve [Aut11]. IK splines are typically used to produce the curved spine of a character's back and in cases where a joint chain needs to follow a specific shape without explicitly setting the rotational values on every joint. Maya's IK spline tool can create the guide curve, or it can be specified when creating the IK spline handle. Figure 3.4 shows two joint chains with IK splines. The one on the right has been positioned by translating a control vertex on the guide curve. In addition to the attributes it inherits from the base block, the IkSpline block also contains:

- startLoc*: a locator at the desired start location of the joint chain
- endLoc*: a locator at the desired end location of the joint chain
- numSpans*: the number of spans in the guide curve
- numJoints*: the number of joints to place along the curve

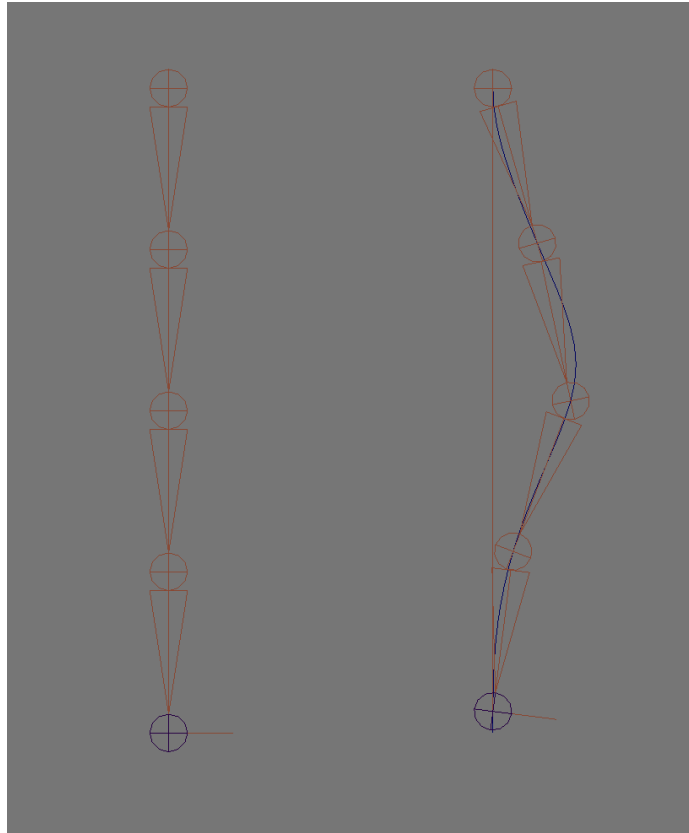


Figure 3.4: Joint chains with IK splines applied.

- ikCrv: the guide curve created by the block
- ikHandle: the IK handle created by the block
- ikEffector: the end effector created by the block.

3.1.3.7 ManDynHair

To create a believable animated character, all aspects of the character must respond to the environment around it. Adding secondary, overlapping motion to hair, tails and tentacles is often difficult and time-consuming; therefore, simulating some or all of their movement can be useful. The ManDynHair block creates a hair system that can be blended with manual, key-framed controls to animate a NURBS curve. The block is based on the

process outlined by Zeth Willie in his tutorial on rigging a strand of hair [Wil09].

Parameters (in addition to those inherited from the base block) of the ManDynHair block include:

- `curveIn`: the curve to which the system will be applied
- `startLoc`: a locator at the base of the curve
- `endLoc`: a locator at the end of the curve
- `hairSystem` (optional): if provided, the hair system node.

This block then creates three copies of the input curve. The block adds a series of inline offset controls evenly spaced along one of the new curves, all of which are connected to the input curve with a blend shape. In manual mode, the controls will directly manipulate the input curve.

The block then attaches the manual control curve to the second copy of the original curve, which acts as the input to a Maya hair dynamics system. A hair follicle is created and the third curve is connected as the output curve on the follicle. A hair system node is created to connect the dynamic input curve to the follicle. The output curve from the follicle is then added to the blend shape that controls the input curve.

The system allows an animator to use one set of controls to explicitly position the final curve and to position the input curve of the hair system. The system can be set to be fully simulated, fully hand-animated, simulated with a strong bias toward the position set by the control, or a blend of all the options.

After a ManDynHair block has been successfully built in the scene, it will have the following attributes:

- `manCrv`: the manual control curve to which the control objects are connected
- `dynInCrv`: the input curve of the hair system
- `dynOutCrv`: the output curve from the follicle

- manDynBs: the blend shape node that connects the manCrv and dynOutCrv to the input curve
- shapeBs: the blend shape node that connects the manCrv to the dynInCrv
- hairSystem: the hair system used to drive the simulation,
- follicle: the follicle that connects the dynOutCrv to the hair system.

3.2 Layering Blocks

The major advantage a scripted system, such as the one proposed here, is the endless possibilities in which blocks can be combined and assembled. Since the blocks are modular, a rig can be as complex or as simple as needed. Additionally, scaling a rig in either direction does not result in hours of lost time. Finally, blocks can be built from within other blocks, thus reducing duplicate code.

3.3 Rigging a Character

The first step in any rigging process is planning the rig. A rigger needs to know details about the appearance of the character and the range of movement the character will need in the story. Figure 3.5 shows the rigging pose sheet for a jellyfish character named Pirate Captain. Based on the rigging requirements presented in the pose sheet and the storyboard (Figure 3.6), the jellyfish required a robust rig that could handle floating, swimming, walking and grasping movements. Each of these can be achieved by utilizing one or more blocks in the proposed system.

3.3.1 Outer Tentacles

One of the most important elements of creating a believable underwater animation for a jellyfish is the secondary overlapping animation on the tentacles, which should portray a believable buoyant floating motion. An effective technique for achieving this goal in Maya

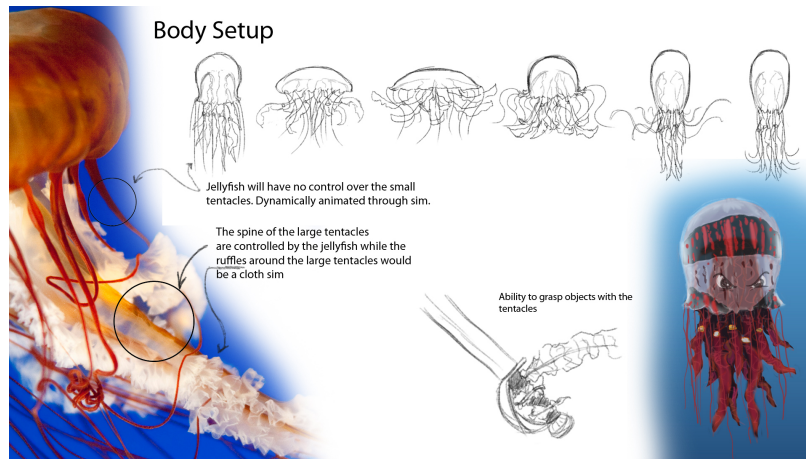


Figure 3.5: Rigging spec sheet for the Pirate Captain character.

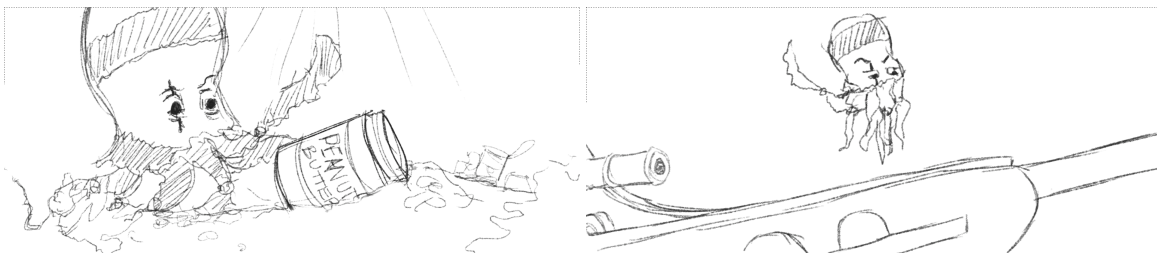
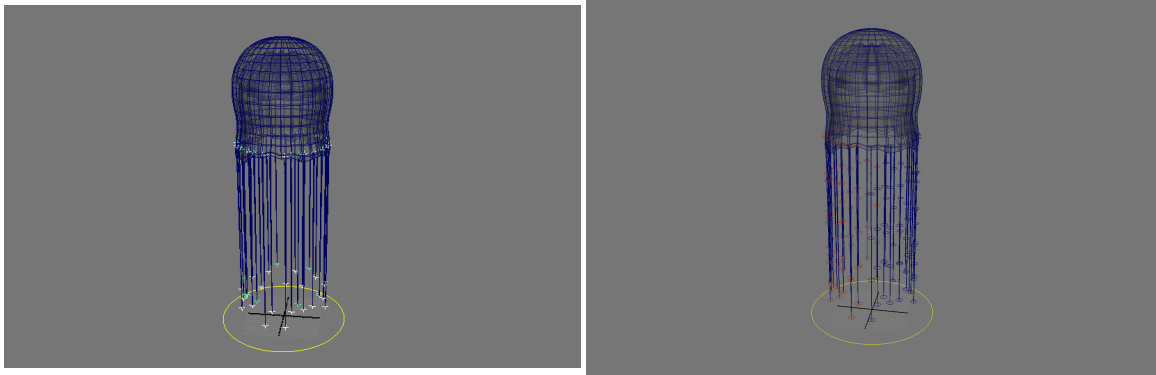


Figure 3.6: Storyboards showing the Pirate Captain's movement.



(a) Locators at the base and end of each tentacle.

(b) The finished rig

Figure 3.7: The outer tentacle rig.

is a hair simulation. The `ManDynHair` block provides an efficient method for controlling a hair system; therefore, it is well suited as the primary control system for the outer tentacles on this character. The `ManDynHair` block does not, however, create bind joints to attach to the geometry. Layering an IK spline under the hair system will appropriately translate the curved shape of the hair curve in a joint chain that can be bound to the character’s mesh.

The first step is to place locators at the top and bottom of each tentacle (Figure 3.7a), each with a unique name. In this example, each tentacle has been assigned a locator at its base (named “`outerTentacle<tentacle number>_top_loc`”) and a locator at its end (named “`outerTentacle<tentacle number>_bottom_loc`”). To ease looping over all the outer tentacles of the character, a list of nested dictionaries specify the mapping of tentacles to locators. Each dictionary was keyed on the name of the tentacle. The value of that key leads to another dictionary with keys (and their corresponding values), representing the attributes passed to the `ManDynHair` block code. The dictionary was prepared with the following form:

```
outerTentacle_locs = [
    {'outerTentacleS3_0001':{
        'start_loc':'outerTentacleS3_0001_top_loc',
        'end_loc':'outerTentacleS3_0001_bottom_loc',
        'color':Ctrl.NAVY,
        'parent':None,}},
    {'outerTentacleS3_0002':{
```

```

        'start_loc': 'outerTentacleS3_0002_top_loc',
        'end_loc': 'outerTentacleS3_0002_bottom_loc',
        'color': Ctrl.NAVY,
        'parent': None,}},
    ...
]

```

Looping over the tentacles to create the IkSpline and ManDynHair blocks was relatively simple and efficient (as shown below). By using a loop to generate the IkSpline and ManDynHair blocks, the code for the thirty tentacles need only be written once, and every tentacle is guaranteed to behave uniformly with its neighbors.

```

hairSys = None
for dict_ in outerTentacle_locs:
    mesh = dict_.keys()[0]
    attrs = dict_.values()[0]

    name = mesh.split('_')
    name = name[0] + name[-1]

    splineBlock = IkSpline(
        startLoc = attrs['start_loc'],
        endLoc = attrs['end_loc'],
        name = name + '_spline',
        numSpans = 10,
        numJoints = 20)

    mdhairBlock = ManDynHair(
        curve = splineBlock.ikCrv,
        startLoc = splineBlock.startLoc,
        endLoc = splineBlock.endLoc,
        numControls = 5,
        name = name,
        color = attrs['color'],
        hairSystem = hairSys,
        radius = 1
    )

```


3.3.2 Inner Tentacles

The inner tentacles present a different challenge. They must be able to move and twist in very specific, art-directable ways. The RibbonIk block is a good candidate for this type of motion. The broad, general shape of the tentacles can be set with the control shapes the block creates, while grasp motions can be fine-tuned by directly manipulating offset controls on the bind joints. Again, a nested dictionary is used to help automate the process. For the inner tentacles, the list of dictionaries is represented in the following way:

```
innerTentacle_locs = [  
    {'tentacleR1_ring_grp_0001':{  
        'start_loc':'tentacleR1_ring_grp_0001_top_loc',  
        'end_loc':'tentacleR1_ring_grp_0001_bottom_loc',  
        'color':Ctrl.BLUE,}},  
    {'tentacleR1_ring_grp_0002':{  
        'start_loc':'tentacleR1_ring_grp_0002_top_loc',  
        'end_loc':'tentacleR1_ring_grp_0002_bottom_loc',  
        'color':Ctrl.RED,}},  
    ...  
]
```

Again, the loop to create the control structure for the tentacles in the Maya scene can be trivially written as follows:

```
for dict_ in innerTentacle_locs:  
    meshGroup = dict_.keys()[0]  
    attrs = dict_.values()[0]  
  
    meshName = group.split('_')  
    name = meshName[0] + meshName[-1]  
  
    ribbonBlock = RibbonIk(  
        startLoc = attrs['start_loc'],  
        endLoc = attrs['end_loc'],  
        name = name + '_ribbonIk',  
        numSpans = 9,  
        numJoints = 20)  
  
    linearSkinBlock = LinearSkin(  
        mesh = ribbonBlock.ribbonIkPlane,  
        startLoc = ribbonBlock.startLoc,
```

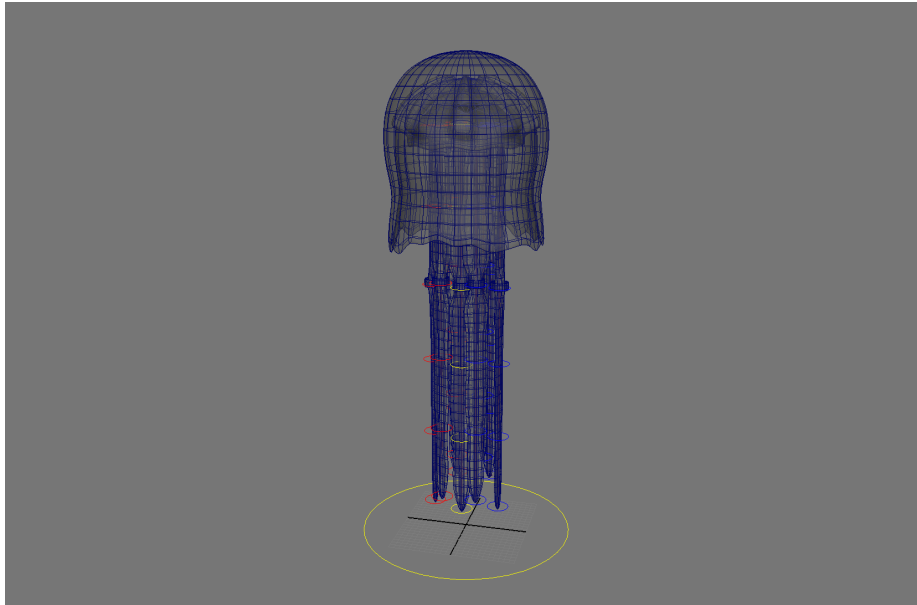


Figure 3.8: The inner tentacles after creating the RibbonIk blocks.

```
endLoc = ribbonBlock.endLoc,  
name = name,  
color = attrs['color'])
```

The above code creates a RibbonIk block for each tentacle, then adds an additional set of joints on top of the ribbon. This additional joint set is given controls for the animator to shape the ribbon. The final rig section can be seen in Figure 3.8.

3.3.3 Bell

The bell of the Pirate Capitan serves as the jellyfish's face, and as such must offer the animator a great deal of control over the final shape. Giving the animator a control object on every vertex would allow the maximum influence over the mesh, but doing so would create a rig that is difficult to set up and unwieldy to animate. To simplify the problem, the bell of the jellyfish is decomposed into two pieces: the lower bell and the upper bell.

Movement in the lower bell must focus on creating the expanding and contracting motion that the jellyfish needs to swim. To allow an animator to exactly position the vertices

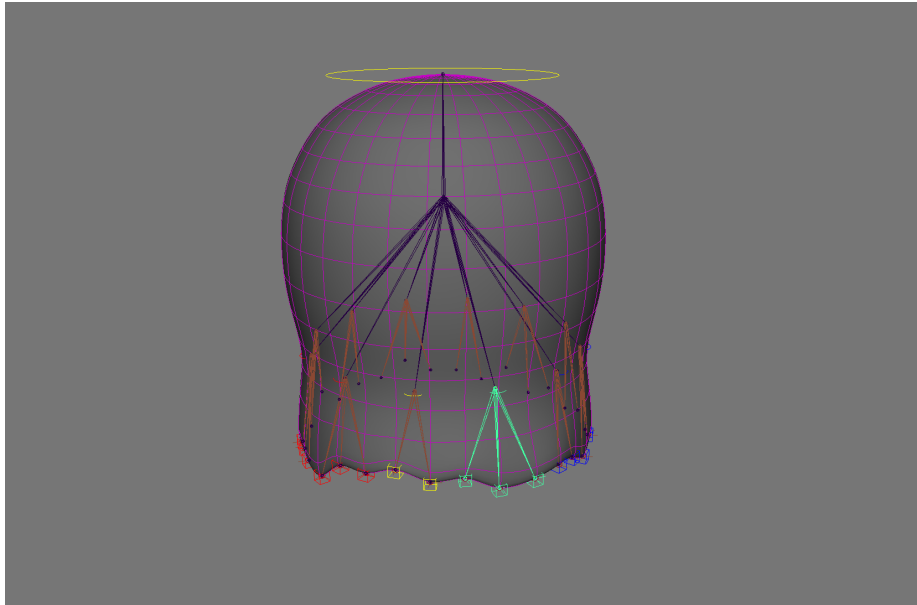


Figure 3.9: The joints and control shapes for the jellyfish’s lower bell.

of the edge of the bell, a ring of IK controllers is used. To control larger sections of the lower bell, the IK controllers are parented to joints higher in the bell. This scheme allows animators to create the general shape with rotation controls, then explicitly position the IK controls to tune the pose. The lower bell controls are created with the IkSC block and the upper rotational controls are created with the InlineOffset block (Figure 3.9).

The rigging system proposed here allows riggers to efficiently create rig scripts to generate customized character setups. PyMel is used to create a library of rig pieces, called blocks, which can be added together and layered to form countless character architectures. Finally, the jellyfish character is analyzed and rigged using the system. Next, the performance and features of the jellyfish rig are explored.

Chapter 4

Results

The resulting rig for the jellyfish character, Pirate Captian, contains 758 joints and 249 control shapes (Figure 4.1). By using the IkSpline block, a full range of tentacle poses can be achieved (Figure 4.2). Additionally, the combination of FK and IK controls used on the lower bell provide a flexible control system (Figure 4.3). The dynamics system used on the outer tentacles presents animators with a method of synthesizing motion without hand-animating every appendage. Other pieces of the Pirate Captain’s geometry can be specified as collision objects, which will repel the curves in the outer tentacles and enhance the believability of the tentacle movement (Figure 4.4).

The blocks system was also used to solve non-character rigging problems. The blocks system was truly tested when a rig was needed for a sea anemone that would populate the background of some shots. Due to the relatively small size of the anemone and the budget for the film, animating hundreds of anemone tentacles by hand was not feasible.

The ManDynHair block was used to create a hair system for each anemone. Another rigging artist was able to incorporate a one-line call to the block code in a procedural modeling script. In under four hours, the script had been refined and could be repeated for any number of anemone needed for a shot. Figure 4.5 shows two frames of the resulting animation.

These results show various ways the block-based rigging framework can be used

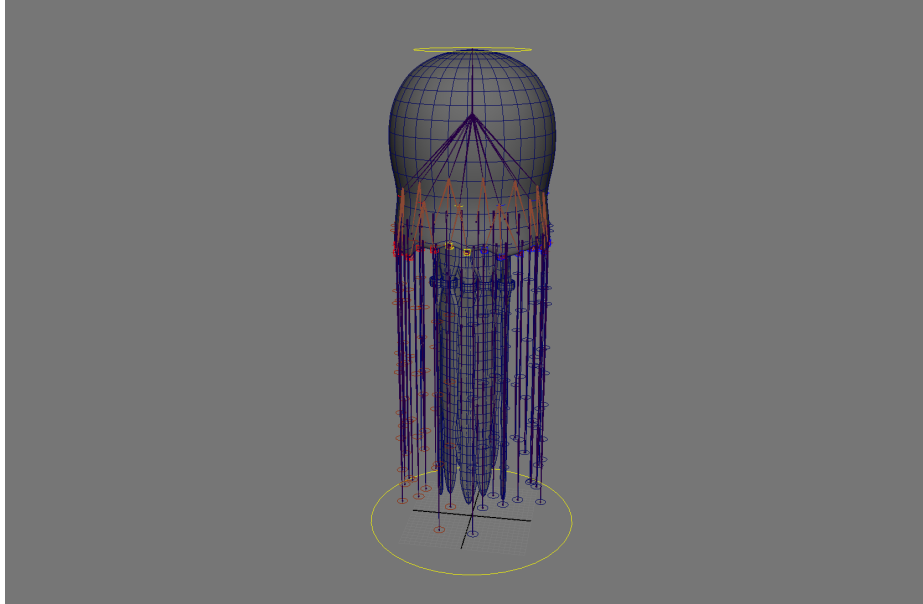


Figure 4.1: The full Pirate Captain rig with all controls activated.

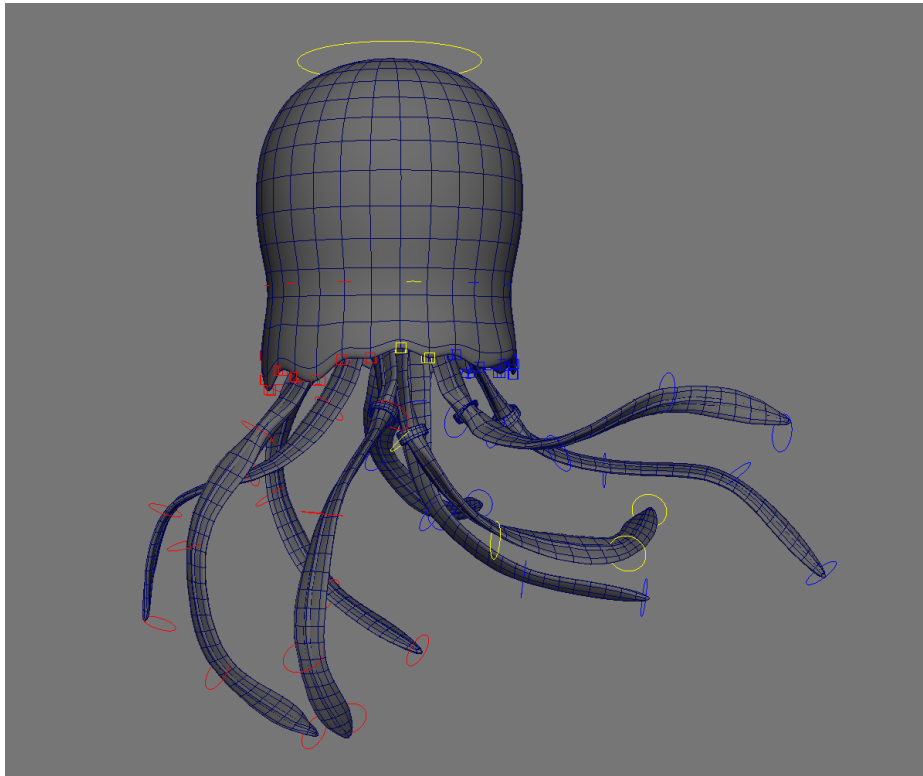


Figure 4.2: The fully posed Pirate Captain rig.

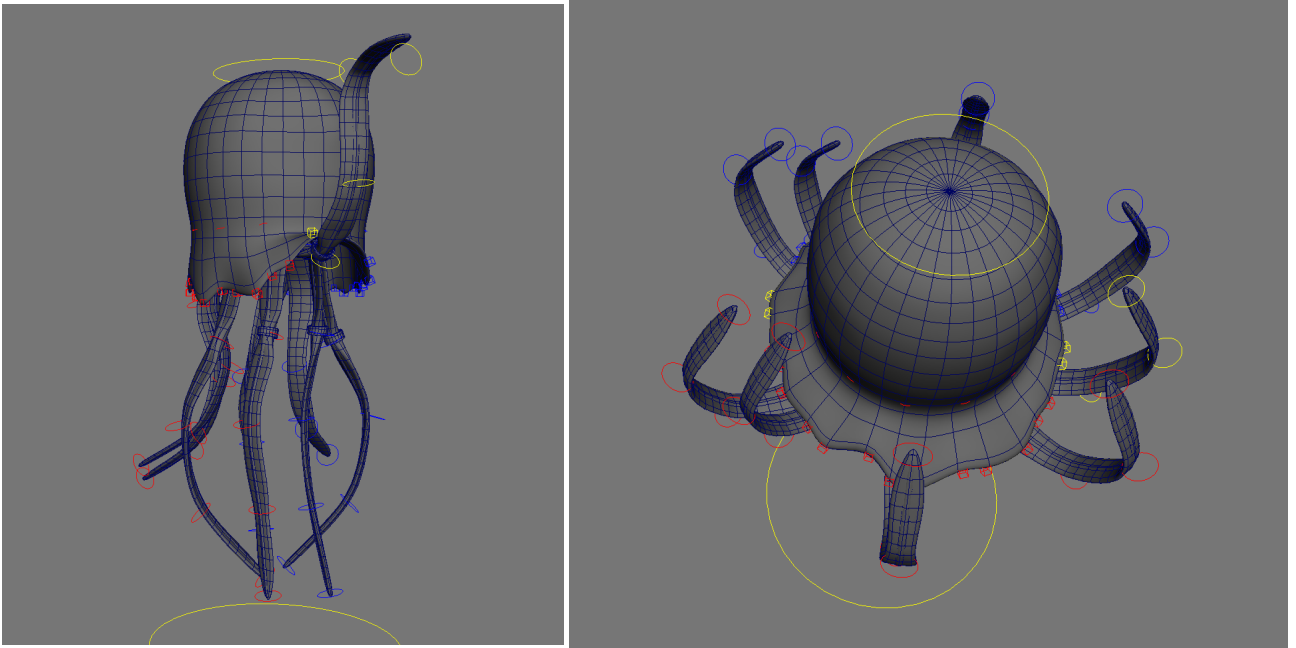


Figure 4.3: Lower bell range of motion tests.

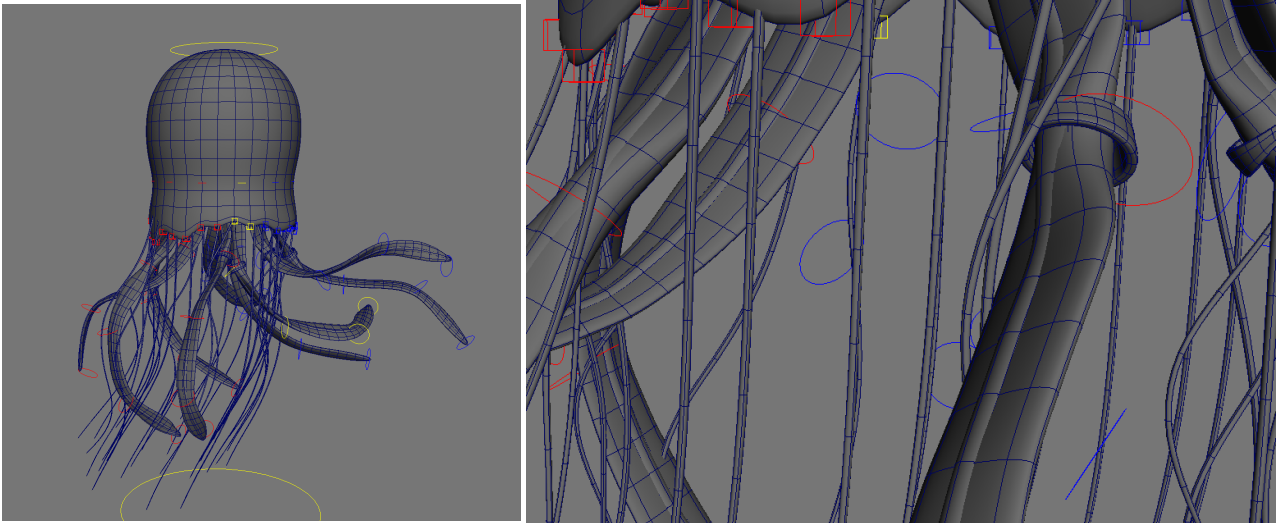


Figure 4.4: The outer tentacles hair simulation.

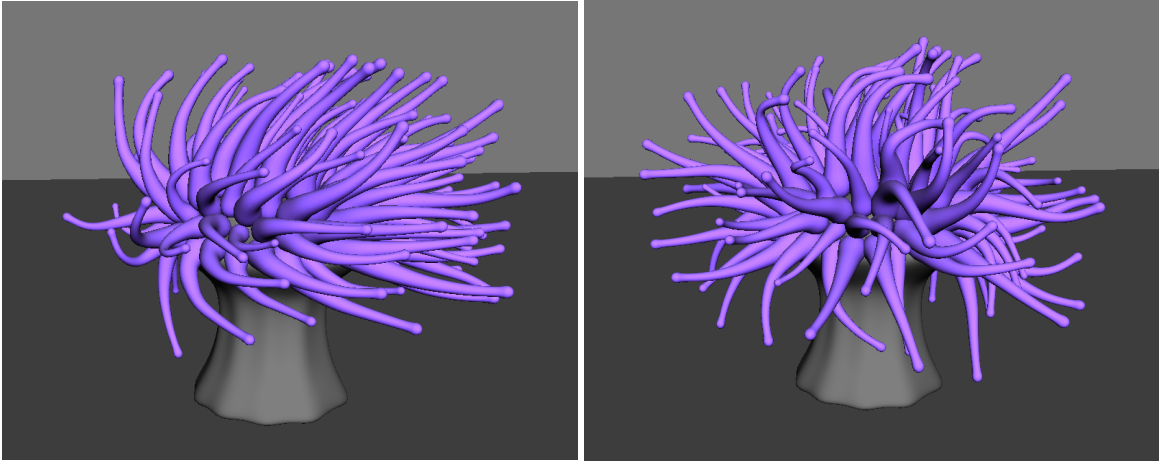


Figure 4.5: Sea anemone rigged using the ManDynHair block (rigging and animation by Alex Beaty).

on multiple characters across many situations. The rigging scripts for each character are portable and can be transferred among artists. Through small alterations, the scripts can be used to create countless sea creatures.

Chapter 5

Conclusions

The blocks rigging system proposes a framework for automating character setup. Its modular design allows for extensive rig types of varying complexity without the need for a complex code base. The script-based (as opposed to GUI-based) nature of the system provides a powerful tool set for rigging artists to quickly apply on a variety of characters.

The benefits of the blocks system became apparent during production. A scripted rigging solution allows changes to be made in upstream departments (such as surfacing) without worry about affecting the rig. The rigging script can simply be rerun, thus recreating an exact replica of the original rig.

The blocks system also created consistency across multiple rigs. Because the blocks system relies on a small handful of pre-scripted blocks, every character uses the same style of controls. This uniformity allows animators to move quickly between characters without learning new control systems. Rig consistency also allows for the creation of better animation tools. Since all the control shapes are named in similar ways across characters rigged with the system, tools can rely on predictable behavior of those shapes on every character.

Several of the characters in the “Peanut Butter Jelly” short would not have been possible without the use of the blocks system. For instance, the Pirate Captain (used as an example in the previous chapters) has 39 individual tentacles and over 750 individual joints. Placing every joint by hand would not have been as accurate, or as fast, as using the tools

the blocks system makes available.

As with any tool in the visual effects and animation pipeline, the proposed rigging system is a work in progress. The extensible nature of Python allows for the addition of new blocks and new features. Further work on this project would expand the library of available blocks. While the code base should remain small and easily maintained, the blocks currently in the system only scratch the surface of possible rig features. Industrial Light and Magic's BlockParty 2 has fewer than 25 individual blocks [Coo13], which facilitates propagating changes and improvements in the block code.

The blocks system currently neglects the skinning process. Each block records the joints it creates, and which of those joints require skinning to a mesh, easing the creation of scripts that handle binding the character's geometry to the joints. The blocks system itself, however, does not include that functionality.

The data in the block objects is not persistent between Maya sessions, which is largely not an issue, since the rigging artist generates a rig script that can recreate the rig from scratch at any time. This approach, however, requires the entire rig to be destroyed every time an update is made to one of the blocks. A more elegant solution might create nodes in Maya to store the information in the blocks for future rigging sessions.

Automated build systems have become the standard for production-quality character rigs. The blocks system provides the framework rigging artists need to quickly produce solid, animation-ready rigs, and the consistent control shape interface helps animators quickly move from character to character.

Appendices

Appendix A Code Examples

A.1 blocks.py

```
import pymel.core as pm

import utils as u
reload(u)

import defines as d
reload(d)

class Ctrl(object):
    CIRCLE = 'circle'
    BOX = 'box'
    ARROWS4 = 'arrows4'

    CRIMSON = 4
    NAVY = 5
    BLUE = 6
    BRICK = 12
    RED = 13
    GREEN = 14
    YELLOW = 17
    AQUA = CYAN = 18

    # shapes
    def circle(name, radius, normal, color):
        circ = pm.circle(
            object=True,
            name=name,
            normal=normal,
            radius=radius,
            constructionHistory = False)[0]
        circ.getShape().overrideEnabled.set(True)
        circ.getShape().overrideColor.set(color)
        return circ

    def box(name, radius, normal, color):
        r = radius
        points = [(-1*r, -1*r,-1*r),
                  (-1*r, -1*r, 1*r),
                  ( 1*r, -1*r, 1*r),
                  ( 1*r, -1*r,-1*r),
                  (-1*r, -1*r,-1*r),
                  (-1*r,  1*r,-1*r),
                  (-1*r,  1*r, 1*r),
```

```

        (-1*r, -1*r, 1*r),
        (-1*r, 1*r, 1*r),
        ( 1*r, 1*r, 1*r),
        ( 1*r, -1*r, 1*r),
        ( 1*r, 1*r, 1*r),
        ( 1*r, 1*r,-1*r),
        ( 1*r, -1*r,-1*r),
        ( 1*r, 1*r,-1*r),
        (-1*r, 1*r,-1*r)]

    b = pm.curve(
        point=points,
        name=name,
        degree=1,
        worldSpace=True)
    u.aimNormal(b, normal=normal)
    pm.makeIdentity(b,apply=True)
    b.getShape().overrideEnabled.set(True)
    b.getShape().overrideColor.set(color)
    return b

def arrows4(name, radius, normal, color):
    r = radius
    points = [(0.25*r, 0*r, 0.75*r),
              (0*r, 0*r, 1*r),
              (-.25*r, 0*r, 0.75*r),
              (0.25*r, 0*r, 0.75*r),
              (0*r, 0*r, 1*r),
              (0*r, 0*r, -1*r),
              (0.25*r, 0*r, -.75*r),
              (-.25*r, 0*r, -.75*r),
              (0*r, 0*r, -1*r),
              (0*r, 0*r, 0*r),
              (-1*r, 0*r, 0*r),
              (-.75*r, 0*r, 0.25*r),
              (-.75*r, 0*r, -.25*r),
              (-1*r, 0*r, 0*r),
              (1*r, 0*r, 0*r),
              (0.75*r, 0*r, -.25*r),
              (0.75*r, 0*r, 0.25*r),
              (1*r, 0*r, 0)]

    a = pm.curve(point=points, name=name, degree=1, worldSpace=True)
    u.aimNormal(a, normal=normal)
    pm.makeIdentity(a,apply=True)
    a.getShape().overrideEnabled.set(True)
    a.getShape().overrideColor.set(color)
    return a

build_shape = {
    CIRCLE: circle,

```

```

    BOX: box,
    ARROWS4: arrows4,
}
def __init__(self,xform=None, name=d.CTRL, shape=CIRCLE, radius=1.0,
             normal=[1,0,0], group=True, color=0, lock=None):
    self.name = name
    self.shape = shape
    self.radius = radius
    self.normal = normal
    self.xform = xform
    self.group = group
    self.color = color
    self.lock = lock

    self.ctrl = None

    self.build()

def build(self):
    self.ctrl = Ctrl.build_shape[self.shape](self.name, self.radius,
                                             self.normal, self.color)

    posObj = self.ctrl
    if self.group:
        pos_obj = pm.group(self.ctrl, world=True,
                          name=self.ctrl.name() + 'Grp')

    if self.xform:
        pm.delete(pm.orientConstraint(self.xform,posObj))
        pm.delete(pm.pointConstraint(self.xform,posObj))

    pm.makeIdentity(self.ctrl,apply=True)

    if type(self.lock) != list:
        self.lock = []
    for attr in self.lock:
        pm.Attribute(self.ctrl.name() + '.*s' % attr).lock()

    return self.ctrl

# blocks

class Block(object):
    '''Abstract Block base class'''
    def __init__(self, name, controlRadius=1.0, controlColor=1,
                 controlShape=Ctrl.CIRCLE, jointPostfix=d.BIND, parent=None,
                 **kwargs):
        self.name = name
        self.controlRadius = controlRadius
        self.controlColor = controlColor

```

```

self.controlShape = controlShape
self.jointPostfix = jointPostfix
self.parent = parent

self.bindJoints = []
self.controls = []

def build(self):
    raise NotImplementedError

class RibbonIk(Block):
    '''Joints equally positioned along the middle of a NURBS plane.

    Useful for tails, tentacles and spines.
    '''

    def __init__(self, startLoc, endLoc, numSpans, numJoints,
                 name='ribbonIk', **kwargs):
        super(RibbonIk, self).__init__(name=name, **kwargs)
        self.startLoc = u.toPmNodes(startLoc)[0]
        self.endLoc = u.toPmNodes(endLoc)[0]
        self.numSpans = numSpans
        self.numJoints = numJoints
        self.ribbonIkPlane = None
        self.follicleGrp = None

        self.build()

    def createFollicle(self, shape, posU=0.5, posV=0.5, name=d.FOLLICLE):
        follicle = pm.createNode('follicle', name=name+'Shape')
        shape.local.connect(follicle.inputSurface)
        shape.worldMatrix[0].connect(follicle.inputWorldMatrix)

        follicle.outRotate.connect(follicle.getParent().rotate)
        follicle.outTranslate.connect(follicle.getParent().translate)
        follicle.parameterU.set(posU)
        follicle.parameterV.set(posV)
        follicle.getParent().t.lock()
        follicle.getParent().r.lock()

        follicle.getParent().rename(name)

    return follicle.getParent()

    def build(self):
        # make the nurbs plane
        self.ribbonIkPlane = pm.nurbsPlane(
            axis = (0,0,1),
            degree = 3, #cubic
            constructionHistory = False,

```

```

        name = self.name + '_' + d.PLANE,
        patchesU = 1,
        patchesV = self.numSpans,
        lengthRatio = self.numSpans)[0]
pm.rebuildSurface(self.ribbonIkPlane, degreeU=1,
                  spansU=1, direction=0)
u.makeNonRendering(self.ribbonIkPlane)
pm.setAttr('%s.%s'% (self.ribbonIkPlane.name(), 'visibility'), 0)
pm.setAttr('%s.%s'% (self.ribbonIkPlane.name(),
                    'inheritsTransform'), 0)

# move the pivots to the top of the plane
self.ribbonIkPlane.setPivots((0, (self.numSpans/2.0), 0))

# place and scale the plane in 3d space
pm.delete(pm.pointConstraint(self.startLoc, self.ribbonIkPlane))
pm.delete(pm.orientConstraint(self.startLoc, self.ribbonIkPlane))
#pm.delete(pm.aimConstraint(self.endLoc, self.ribbonIkPlane,
#                           aimVector=(0, -1, 0)))
#                           #skip      =('x', 'z'))
height = u.distance(self.startLoc, self.endLoc)
scale = (height / self.numSpans)
self.ribbonIkPlane.scaleBy((scale, scale, scale))

# create and attach follicles
follicles = []
for i in range(1, (self.numJoints+1)):
    follicle = self.createFollicle(
        shape = self.ribbonIkPlane.getShape(),
        posV = (i-1.0)/float(self.numJoints-1.0),
        posU = 0.5,
        name = self.name + "_%s%02d" % (d.FOLLICLE, i))
    pm.setAttr(follicle.visibility, False)
    follicles.append(follicle)

self.follicleGrp = pm.group(follicles, name='%s_%s_grp' % \
                            (self.name, d.FOLLICLE))
pm.setAttr('%s.%s'% (self.follicleGrp, 'inheritsTransform'), 0)

# create the bind joints
for i, f in enumerate(follicles):
    self.bindJoints.append(u.placeJoint(
        position = u.getWsLocation(f),
        name = '%s%s%02d_%s'%(self.name, d.JOINT.title(),
                               i+1, self.jointPostfix),
        parent = f))

# parent
self.ribbonIkPlane.setParent(self.parent)

```

```
class InlineOffset(Block):
```

```

'''Inserts control shapes into the scene hierarchy above the given jnts.
'''

def __init__(self, joints, name='offset', controlShape=Ctrl.ARROWS4,
             **kwargs):
    super(InlineOffset,self).__init__(name=name,controlShape=controlShape,
                                     **kwargs)

    if type(joints) != list:
        joints = [joints]

    self.joints = joints

    self.build()

def build(self):
    for joint in self.joints:
        joint = pm.PyNode(joint)
        parent = joint.getParent()
        ctrl = Ctrl(xform = joint,
                  name = joint.name() + '_' + d.CTRL,
                  shape = self.controlShape,
                  radius = self.controlRadius,
                  normal = [0,1,0],
                  color = self.controlColor,
                  group = False).ctrl
        ctrl.setParent(parent)
        pm.makeIdentity(ctrl, apply=True)
        joint.setParent(ctrl)
        self.controls.append(ctrl)

class LinearSkin(Block):
    '''Places a joint chain in the scene and skins the given mesh to it.
    '''
    def __init__(self, mesh, startLoc, endLoc, numControls=3, name="ls",
                 controlShape=Ctrl.CIRCLE, jointPostfix=d.DRVR, **kwargs):
        super(LinearSkin,self).__init__(name=name, controlShape=controlShape,
                                       jointPostfix=jointPostfix, **kwargs)

        self.meshIn = mesh
        self.startLoc = startLoc
        self.endLoc = endLoc
        self.numControls = numControls

        self.controls = []
        self.drivers = []
        self.skin = None

        self.build()

    def build(self):
        # place joints in the scene
        self.drivers = u.placeJointChain(

```



```

        self.startLoc,
        self.endLoc,
        numJoints = self.numControls,
        parent = None,
        name = '%s_%s' % (self.name, self.jointPostfix)
pm.delete(pm.orientConstraint(self.startLoc, self.drivers[0]))

self.skin = pm.skinCluster(
    self.drivers, self.meshIn,
    toSelectedBones = True,
    maximumInfluences = 2)

self.controls = InlineOffset(
    self.drivers, controlRadius=self.controlRadius,
    controlColor=self.controlColor,
    controlShape=self.controlShape).controls

self.controls[0].setParent(self.parent)

class IkSC(Block):
    '''Makes an IK system (using Maya's SC solver).
    '''
    def __init__(self, startJoint, endJoint, name=d.IK, controlShape=Ctrl.BOX,
        **kwargs):
        super(IkSC, self).__init__(name=name, controlShape=controlShape,
            **kwargs)
        self.startJoint = pm.PyNode(startJoint)
        self.endJoint = pm.PyNode(endJoint)

        self.ikHandle = None

        self.build()

    def build(self):
        self.ikHandle = pm.ikHandle(
            startJoint = self.startJoint,
            endEffector = self.endJoint,
            name = self.name + '_%s' % d.IK_HANDLE,
            sticky = 'sticky',
            solver = 'ikSCsolver')[0]
        self.ikHandle = pm.PyNode(self.ikHandle)
        ctrl = Ctrl(xform = self.endJoint, normal=[0,1,0],
            name=self.name+'_%s'% d.CTRL, shape=self.controlShape,
            radius=self.controlRadius, group=True,
            color=self.controlColor, lock=['rx','ry','rz']).ctrl
        self.ikHandle.setParent(ctrl)
        self.controls.append(ctrl)

```

```

class IkSpline(Block):
    '''Places a joint chain in the scene and applies an IK spline system.
    '''
    def __init__(self, startLoc, endLoc, numSpans, numJoints,
                 name='ikSpline', **kwargs):
        super(IkSpline, self).__init__(name=name, **kwargs)
        self.startLoc = startLoc
        self.endLoc = endLoc
        self.numSpans = numSpans
        self.numJoints = numJoints
        self.name = name

        self.joints = []
        self.ikCrv = None
        self.ikHandle = None
        self.ikEffector = None

        self.build()

    def build(self):
        self.joints = u.placeJointChain(
            startLoc = self.startLoc,
            endLoc = self.endLoc,
            numJoints = self.numJoints,
            parent = None,
            name = self.name)
        self.joints[0].setAttr('visibility', False)

        self.ikHandle, self.ikEffector, self.ikCrv = pm.ikHandle(
            startJoint = self.joints[0],
            endEffector = self.joints[-1],
            name = self.name + '_' + d.IK_HANDLE,
            solver = 'ikSplineSolver')
        self.ikCrv = pm.PyNode(self.ikCrv)
        self.ikHandle = pm.PyNode(self.ikHandle)

        self.ikHandle.inheritsTranform = False
        self.ikHandle.setAttr('visibility', False)

        self.ikCrv.rename(self.name + '_' + d.CURVE)
        self.ikCrv.setAttr('visibility', False)

        pm.rebuildCurve(self.ikCrv,
            degree = 3,
            spans = self.numSpans,
            keepRange = 0,
            constructionHistory = False)

class ManDynHair(Block):
    '''Creates a system to blend between manual and dynamic ctrl of a crv.
    '''

```

```

def __init__(self, curve, startLoc, endLoc, numControls=3,
             name='manDynHair', hairSystem=None,
             controlRadius=2, **kwargs):
    super(ManDynHair, self).__init__(name=name, controlRadius=controlRadius,
                                     **kwargs)

    self.curveIn = pm.PyNode(curve)
    self.startLoc = pm.PyNode(startLoc)
    self.endLoc = pm.PyNode(endLoc)
    self.numControls = numControls

    self.manCrv = None
    self.dynInCrv = None
    self.dynOutCrv = None
    self.manDynBs = None
    self.shapeBs = None
    self.driverJoints = []
    self.hairSystem = hairSystem
    self.follicle = None

    self.build()

def build(self):
    self.curveIn.setAttr('inheritsTransform',False)
    # create a crv and ctrls to act as a manual driver for the sys
    self.manCrv = pm.duplicate(self.curveIn, returnRootsOnly=True,
                              name = '%s_man_%s' % (self.name,d.CURVE))
    self.manCrv[0].inheritsTransform = False
    self.manCrv[0].setAttr('visibility',False)

    linearSkin = LinearSkin(
        mesh = self.manCrv,
        startLoc = self.startLoc,
        endLoc = self.endLoc,
        numControls = self.numControls,
        name = self.name,
        controlColor = self.controlColor,
        controlRadius = self.controlRadius,
        controlShape = self.controlShape)

    self.driverJoints = linearSkin.drivers
    self.controls = linearSkin.controls

    self.dynInCrv = pm.duplicate(self.manCrv,returnRootsOnly=1,
                                name = '%s_dynIn_%s' % (self.name,
                                                         d.CURVE))[0]

    self.dynInCrv.setAttr('visibility',False)
    # drive the dynamic input curve with the manual curve
    self.shapeBs = pm.blendShape(self.manCrv, self.dynInCrv,
                                name = '%s_shape_%s' % (self.name,
                                                         d.BLENDSHAPE))
    pm.blendShape(self.shapeBs, edit=True, weight=[[0,1]])

```

```

# make a dynamic curve that is driven by a hairSystem
self.dynOutCrv = pm.duplicate(
    self.dynInCrv, returnRootsOnly=1,
    name = '%s_dynOut_%s' % (self.name, d.CURVE))[0]
self.dynOutCrv.inheritsTransform = False
self.dynOutCrv.setAttr('visibility',False)
self.follicle = pm.createNode('follicle', skipSelect=1,
    name = '%s_%sShape' % (
        self.name,d.FOLLICLE))
self.follicle = pm.rename(self.follicle.getParent(),
    '%s_%s' % (self.name,d.FOLLICLE))
self.follicle.setAttr('visibility',False)
self.follicle.restPose.set(1)

if self.hairSystem == None or not pm.objExists(self.hairSystem):
    self.hairSystem = pm.createNode(
        'hairSystem', skipSelect=1,
        name='%s_%sShape' % (self.name,d.HAIR_SYS))
    self.hairSystem = pm.rename(self.hairSystem.getParent(),
        '%s_%s' % (self.name,d.HAIR_SYS))
    pm.PyNode('time1').outTime >> \
        self.hairSystem.getShape().currentTime
    self.hairSystem.setAttr('visibility',False)

hairSystem = pm.PyNode(self.hairSystem)
hairIndex=len(hairSystem.getShape().inputHair.listConnections())

pm.parent(self.dynInCrv, self.follicle)
self.dynInCrv.getShape().worldSpace[0] >> \
    self.follicle.getShape().startPosition
self.follicle.getShape().outCurve >> \
    self.dynOutCrv.getShape().create
self.follicle.getShape().outHair >> \
    hairSystem.getShape().inputHair[hairIndex]
hairSystem.getShape().outputHair[hairIndex] >> \
    self.follicle.getShape().currentPosition

# drive the input curve with a blendshape to blend btwn
# the manual and dynamic curves
self.manDynBs = pm.blendShape(
    self.manCrv, self.dynOutCrv, self.curveIn,
    name='%s_manDyn_%s' % (self.name, d.BLENDSHAPE))[0]

pm.addAttr(self.controls[0],
    longName='manDynBlend', attributeType='float',
    minValue = 0.0, maxValue = 1.0, keyable=True)

pma = pm.createNode('plusMinusAverage', skipSelect=1,
    name = '%s_%s' % (self.name, d.PMA))

```

```

pma.setAttr('operation','Subtract')
pma.setAttr('input1D[0]',1)

self.controls[0].manDynBlend >> self.manDynBs.weight[1]
self.controls[0].manDynBlend >> pma.input1D[1]
pma.output1D >> self.manDynBs.weight[0]

class Lattice(Block):
    '''Creates a lattice around a given vertices.
    '''

    def __init__(self, geometry, latticeDivisions=[5,5,5], name='lattice',
                 **kwargs):
        super(Lattice,self).__init__(name=name, **kwargs)
        self.geometry = geometry
        self.latticeDivisions = latticeDivisions

        self.lattice = None

        self.build()

    def build(self):
        pm.select(clear=True)
        pm.select(self.geometry)
        self.lattice = pm.nt.Lattice(objectCentered = True,
                                     divisions = self.latticeDivisions,
                                     frontOfChain = True,
                                     name = self.name)
        self.lattice.getParent().setParent(self.parent)

```

References

- [Atk08] D. Atkinson, “A Short History (part II),” http://minyos.its.rmit.edu.au/aim/a_notes/anim_history_02.html, 2008.
- [Aut11] Autodesk, Inc., “Autodesk Maya 2012 User’s Guide,” http://download.autodesk.com/global/docs/maya2012/en_us/index.html, 2011.
- [Aut14a] Autodesk, Inc., “Autodesk Maya Product Website,” <http://www.autodesk.com/products/autodesk-maya/>, accessed March 2014.
- [Aut14b] Autodesk, Inc., “Autodesk Softimage Last Release Annoucement,” [press release]. <http://www.autodesk.com/products/autodesk-softimage/overview>, March 7 2014.
- [Bad87] N. Badler, K. Manoochehri, and G. Walters, “Articulated Figure Positioning by Multiple Constraints,” *Computer Graphics and Applications, IEEE*, 7(6):28–38, June 1987.
- [Bod97] B. Bodenheimer, C. Rose, S. Rosenthal, and J. Pella, “The Process of Motion Capture: Dealing with the Data,” *Computer Animation and Simulation*, 1997.
- [Bor12] P. Borosán, M. Jin, D. DeCarlo, Y. Gingold, and A. Nealen, “RigMesh: Automatic Rigging for Part-based Shape Modeling and Deformation,” *ACM Trans. Graph.*, 31(6):198:1–198:9, November 2012.
- [Boz95] J. S. Bozman, “SGI Acquires 3-D Graphics Firms; Company Targets Interactive Media Market, Attempts to Beat Out Microsoft,” *Computerworld*, 29(9):32, February 27 1995.
- [Bur75] N. Burtnyk and M. Wein, “Computer Animation of Free Form Images,” in “Proceedings of the 2nd Annual Conference on Computer Graphics and Interactive Techniques,” SIGGRAPH ’75, pp. 78–80, ACM, New York, NY, USA, 1975.
- [Bur76] N. Burtnyk and M. Wein, “Interactive Skeleton Techniques for Enhancing Motion Dynamics in Key Frame Animation,” *Commun. ACM*, 19(10):564–569, October 1976.
- [Car07] W. Carlson, “A History of Bell Labs,” <http://design.osu.edu/carlson/history/tree/bell.html>, 2007.

- [Cha89] J. E. Chadwick, D. R. Haumann, and R. E. Parent, “Layered Construction for Deformable Animated Characters,” in “Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques,” SIGGRAPH ’89, pp. 243–252, ACM, New York, NY, USA, 1989.
- [Coo13] C. Cooper, Creature Technical Director, Industrial Light and Magic. [Personal Communication], July 2013.
- [Cus11] R. Cushman, *Open Source Rigging in Blender: A Modular Approach*, Master’s Thesis, Clemson University, May 2011.
- [Dre12] T. Dreher, “History of Computer Art,” <http://iasl.uni-muenchen.de/links/GCA-IV.2e.html>, April 2012.
- [Fet14] W. Fetter, “Fifty Percentile Human Figures related to Cockpit,” <http://dada.compart-bremen.de/item/artwork/244>, accessed March 2014.
- [Gia98] M. Giambruno, “Maya Manifest,” *Interactivity*, 4(10):51, October 1998.
- [Hel13] R. Helms, Character Technical Director. PDI|DreamWorks Animation. [Personal Communication], May 2013.
- [Jos10] Jose Antonio Martin Martin, “Ribbon Spline Rig,” <http://www.creativecrash.com/maya/tutorials/character/c/ribbon-spine-rig>, June 2010.
- [KA03] Z. Kačić-Alesić, M. Nordenstam, and D. Bullock, “A Practical Dynamics System,” in “Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation,” SCA ’03, pp. 7–16, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2003.
- [Lew64] J. A. Lewis and E. E. Zajac, “A Two-Gyro, Gravity-Gradient Satellite Attitude Control System,” *Bell Systems Technical Journal*, 43(6):2705–2765, November 1964.
- [Lum14a] Luma Pictures, “People: Chad Dombrova,” <http://www.lumapictures.com/people/?433>, accessed March 2014.
- [Lum14b] Luma Pictures, “pymel - Python in Maya Done Right,” <https://code.google.com/p/pymel/>, accessed March 2014.
- [McL11] T. McLaughlin, L. Cutler, and D. Coleman, “Character Rigging, Deformations, and Simulations in Film and Game Production,” in “ACM SIGGRAPH 2011 Courses,” SIGGRAPH ’11, pp. 5:1–5:18, ACM, New York, NY, USA, 2011.
- [Mea11] S. P. Means, “Pixar founder’s Utah-made ‘Hand’ added to National Film Registry,” *The Salt Lake Tribune*, December 2011.
- [Mec12] A. Mechtley and R. Trowbridge, *Maya Python for Games and Film: A Complete Reference for Maya Python and the Maya Python API*, Morgan Kaufmann, Waltham MA, 2012.

- [Num11] N. Numaguchi, A. Nakazawa, T. Shiratori, and J. K. Hodgins, “A Puppet Interface for Retrieval of Motion Capture Data,” in “Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation,” , August 2011.
- [O’N08] R. O’Neill, *Digital Character Development: Theory and Practice*, Morgan Kaufmann, 2008.
- [Par12] R. Parent, *Computer Animation, Third Edition: Algorithms and Techniques*, Morgan Kaufmann, 2012.
- [Pla03] “An Oscar for Alias/Wavefront,” *Playback*, 17(9):2, January 20 2003.
- [PRN08] “Autodesk Completes Acquisition of Softimage; Acquisition Complements Autodesk’s 3D Animation Portfolio,” *PR Newswire US*, November 18 2008.
- [Rob05] M. Robinson, “Autodesk acquires rival software maker Alias,” *Playback*, p. 17, October 24 2005.
- [Ros13] R. Rose, M. Jutan, and J. Doublestein, “BlockParty 2: Visual Procedural Rigging for Film, TV, and Games,” in “ACM SIGGRAPH 2013 Talks,” SIGGRAPH ’13, pp. 8:1–8:1, ACM, New York, NY, USA, 2013.
- [Smi06] J. Smith and J. White, “BlockParty: Modular Rigging Encoded in a Geometric Volume,” in “ACM SIGGRAPH 2006 Sketches,” SIGGRAPH ’06, ACM, New York, NY, USA, 2006.
- [Smi12] G. Smith, M. McLaughlin, A. Lin, E. Goldberg, and F. Hanner, “dRig: An Artist-friendly, Object-oriented Approach to Rig Building,” in “ACM SIGGRAPH 2012 Talks,” SIGGRAPH ’12, p. 18:1, ACM, New York, NY, USA, 2012.
- [Tel04] “Silicon Graphics Sells Alias Software Business; Transaction Increases Focus on SGI Core Systems Business, Boosts Liquidity,” *Television*, p. 55, April 15 2004.
- [Wad00] L. Wade, *Automated Generation of Control Skeletons for Use in Animation*, Ph.D. Dissertation, The Ohio State University, 2000.
- [War97] M. Ward, “A (Spotty) History and Who’s Who of Computer Graphics,” <http://web.cs.wpi.edu/~matt/courses/cs563/talks/history.html>, 1997.
- [Wil09] Z. Willie, “Maya Rigging: Dynamic/Manual Hair Setup,” <http://vimeo.com/8240821>, December 18 2009.
- [WSJ95] “Business Brief – Silicon Graphics: Mergers With Wavefront And Alias Are Completed,” *Wall Street Journal*, p. B5, June 19 1995.
- [Zel82] D. Zeltzer, “Motor Control Techniques for Figure Animation,” *Computer Graphics and Applications, IEEE*, 2(9):53–59, November 1982.