Documentation Practical Thesis
Michael Zünd
Master Film

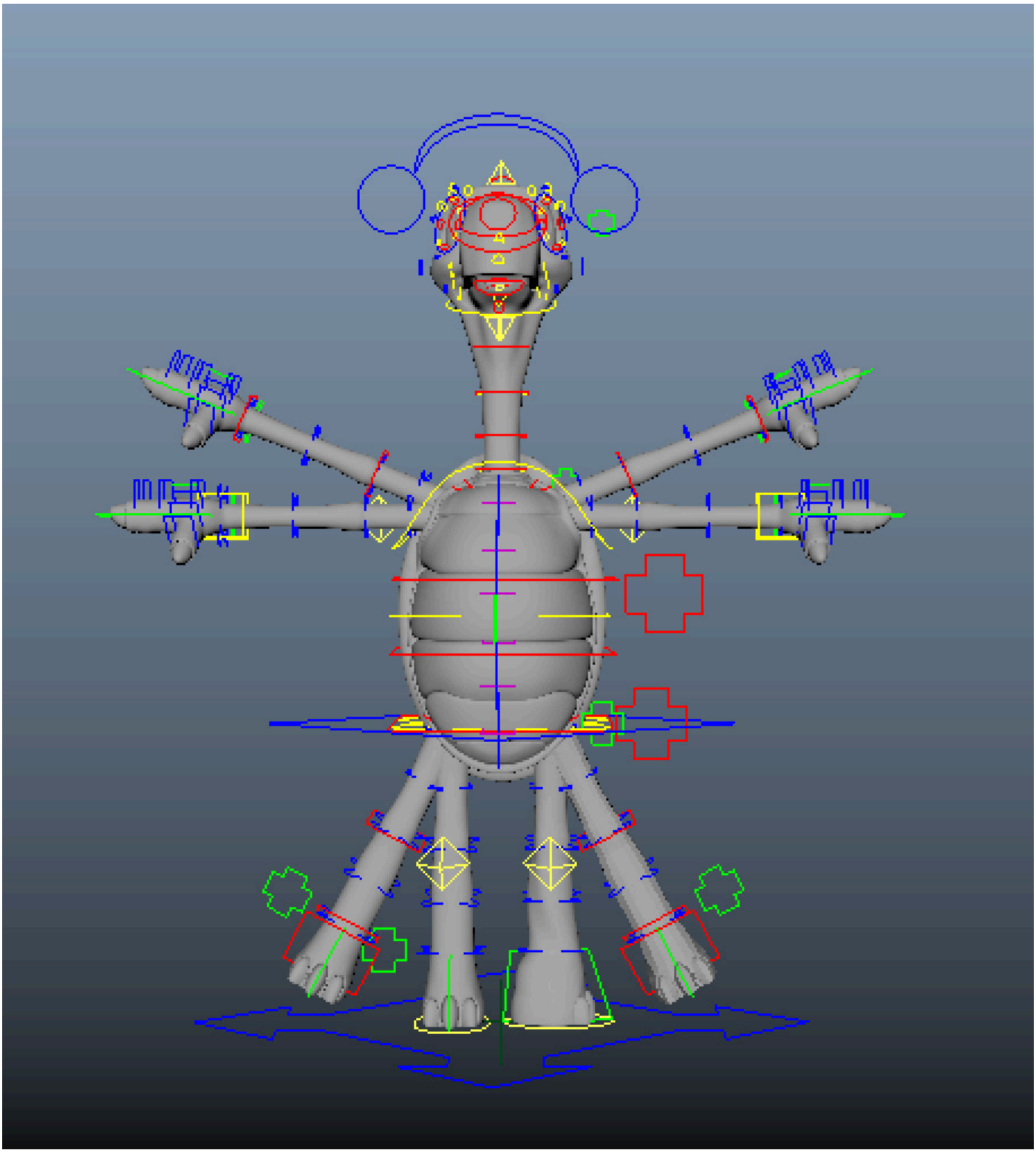


Table of Contents

1. Preface

I started the master's program because I wanted to study rigging for stylised 3D animation—how to mimic the look and feel of the source material when animating characters from a classic cartoon or a comic book.
I initially thought that I would create a short film that features a lot of different settings and characters, so I could develop my skills while also working creatively on a practical project. However, I quickly ran into problems during the story development, mainly because I wanted many different types of characters to explore a wide range of styles. This caused the story to become extremely fragmented and hard to understand. I also wanted my main characters to move like characters from an old slapstick cartoon, but this didn't really fit my normal style of storytelling, as I am normally drawn to projects that aim to convey a message or a feeling. After struggling with this story for more than six months, rewriting it multiple times and creating an animated storyboard, it became obvious that my heart wasn't in this project.
At the same time, I was working on my written thesis on rigging for stylised characters and I was fascinated by the technical aspects of the rigging setups. Through the research and particularly through two amazing interviews with industry professionals, I was able to get a much clearer picture of where I still had gaps in my knowledge and which of my existing skills I needed to develop further to become truly proficient. I realised that I would learn more if instead of working on a short film, I would create a rigging reel with characters that pose different technical and stylistic challenges. I also decided to program a fast rigging tool, which would allow me to directly translate my new acquired skills into code—as Richard Feynman said: what I cannot create, I do not understand.

2. The Awesome_Fastrig (Working title)

2.1 Motivation

The question

The fastrig was by far the biggest project I had tackled during my master, as well as the one I received the most pushback on for various reasons. The question „why?“ accompanied me pretty closely during the first few months of this project.
My regular lecturers asked it because, admittedly, this wasn't a project that was suited for a master film. Moreover, while they were very supportive, they couldn't really offer much support. I also received the question from my technical mentors, who asked why I wanted to write a fastrig when there are so many professional options from companies that had invested more time and resources than I could during the masters.

So why?

There were two main reasons why I wanted to tackle this project, although I knew from the beginning that I still would work with „Human IK“ and „Advanced Skeleton“ even if this experiment was successful.
On one hand, I wanted to challenge myself. I had just come down from the first big wave of confidence in my Python skills and the interviews for my written thesis had made clear how crucial strong programming skills are to really push the boundaries as a rigging professional. I was pretty good at writing small scripts for everyday animation work, but I was very slow and I had never tackled a bigger project that required me to create multiple versatile functions and iterate through different objects using their relations.
On the other hand, I had a genuine need for a fastrig. More and more often, I found myself digging into rigs that I just had created with a fastrig to exchange setups with more reliable or robust

ones. That cost me a lot of time, so I started to write my own module library so I could just create modules and link them to other rigs. This turned into a fastrig designed to create fully connected rigs using the techniques I preferred for my projects.
As an added bonus, I also gained experience working with the „script only“ rigs—rigs that don’t use a UI—that many big companies rely on. Job candidates for rigging positions, routinely get asked if they have experience using such rigs, but since those tools are only used in-house and not available to the public, gaining experience with such rigs is basically impossible short of writing one yourself.

2.2 Goals

The concrete objective of this project was to create a fastrig that could create cartoony rigs for bipedal characters. The rigs should have a robust squash and stretch system for every module that utilises joint chains, as well as bend options for the limbs and spine. The fastrig should also give users the ability to choose to only have an IK or FK mode if the other isn’t needed.
A further goal was to rely on matrix-based constraints instead of the common Maya constraints for constraint connections.
Finally, I set myself the goal of connecting this project to my other projects by creating one of the body rigs in my reel with this script.

2.3 Technical process

Language

The fastrig was created using Python 2.7 and is based on the maya.cmds and pymel.core libraries. This means that it is utilising MEL commands called from Python instead of Maya’s Python API. I decided to use this approach over using the Python or C++ API because it allows me to get direct feedback from the Maya MEL Script Editor‘s command history.
I also decided to use Python instead of MEL for two key reasons. On one hand, Python allows for more dynamic coding as the language allows the dynamic growing/shrinking of lists and dyna-

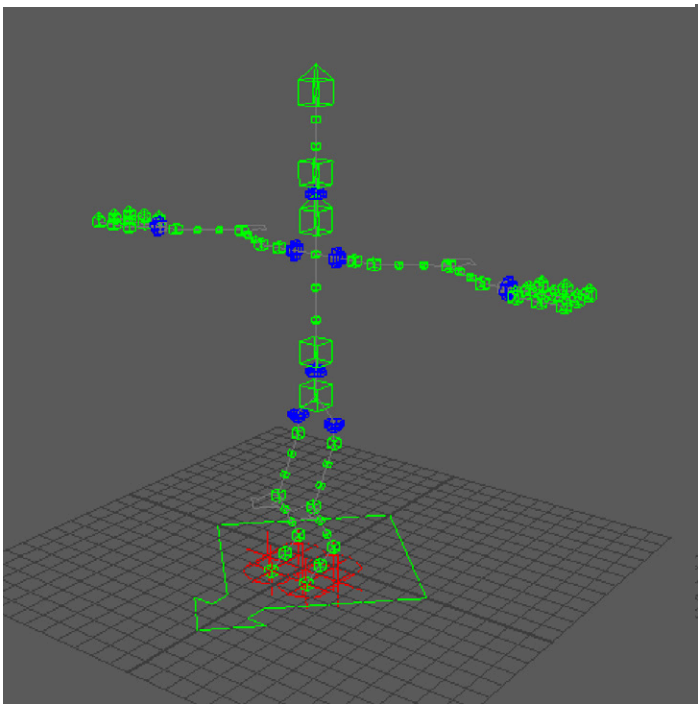
```
671  def translateCurveUser(trans = (None,None,None)):
672      """translates cv's of selected curve to match the given parameters (rot) argument type tuple (float,float,float)"""
673
674      sel = mc.ls(selection = True)
675
676      for crv in sel:
677          shapes =mc.listRelatives(crv, s= True, c = True) or []
678          for shape in shapes:
679              if mc.attributeQuery("spans",n =  shape, ex = True) == True:
680                  degs = mc.getAttr( '{0}.degree'.format(crv))
681                  spans = mc.getAttr( '{0}.spans'.format(crv))
682
683                  cvs = degs+spans
684
685                  mc.move(trans[0],trans[1],trans[2],'{0}.cv[0:{1}]'.format(crv,cvs), os = True, co = True)
686
687  def scaleCurveUser(sc = (None,None,None)):
688      """scales cv's of selected curve to match the given parameters (rot) argument type tuple (float,float,float)"""
689      sel = mc.ls(selection = True)
690
691      for crv in sel:
692          shapes =mc.listRelatives(crv, s= True, c = True) or []
693          for shape in shapes:
694              if mc.attributeQuery("spans",n =  shape, ex = True) == True:
695                  degs = mc.getAttr( '{0}.degree'.format(crv))
696                  spans = mc.getAttr( '{0}.spans'.format(crv))
697
698                  cvs = degs+spans
699
700                  mc.scale(sc[0],sc[1],sc[2],'{0}.cv[0:{1}]'.format(crv,cvs), os = True, ocp = True)
701
702  def weightedMatrixConst( constName = '', mo = True, parent1 = None, parent2 = None, poc = None,  child = None, w1 = 1, w2 = 1, trans = None, rot = None, scale = None):
703      '''creates a weighted constraint between two parent objects using matrixs nodes. Takes inputs:constName(String) mo (maintain Offset, Boolean), parent1 (name first pa
704      poc (name of the childs direct parent, string), child (name of child, string), w1 (weight of parent1, float), w2 (weight of parent2, float), trans (specifies transla
705      rot (specifies rotation Channel, takes strings 'ALL', 'X','Y','Z', input None skips rotation), scale (specifies scale Channel, takes strings 'ALL', 'X','Y','Z', inpu
706      For this function to work child needs a parent node'''
707
708      channelInputList  = [trans, rot, scale]
709      channelList  = [['Translate','translate'], ['Rotate','rotate'], ['Scale','scale']]
710
711      # create weight blend node
712      wtAddMatrix = mc.createNode('wtAddMatrix', name ='{0}_wtAddM'.format(constName))
713
714      #create matrix constraint
715      parentList = [parent1, parent2]
716      for p in range(0, len(parentList)):
717          n = 0
718          multMatrix = mc.createNode('multMatrix', name ='{0}_{1}_multM'.format(constName, parentList[p]))
719
720          if mo == True:
```

Cutout from the code of the fastrig

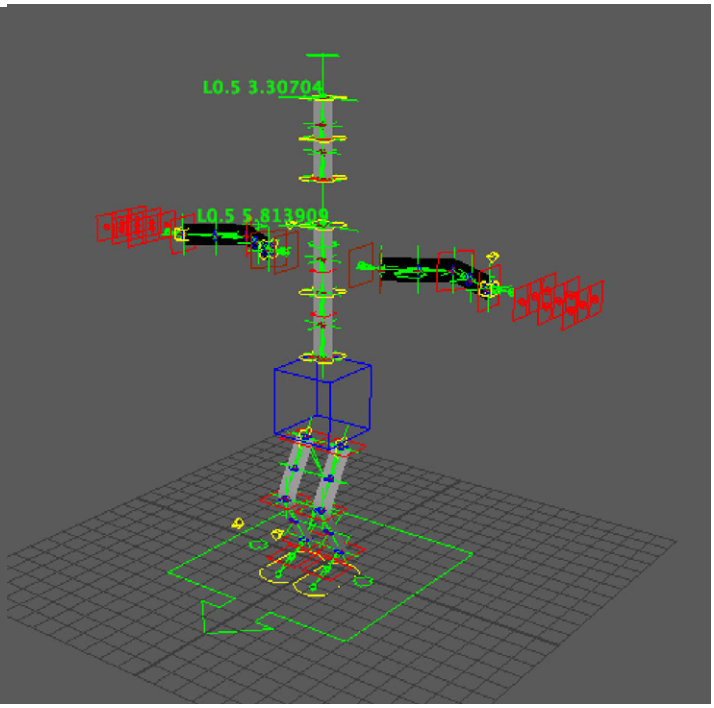
mic assignment of variables. Amongst other advantages, this makes it easy to track and organise created objects or to set up dictionaries to organise the connections that need to be made between these objects. On the other hand, Python is a widely used language that is compatible with almost every 3D software on the market and with many other applications, unlike MEL, which is only compatible with Maya and Maya LT. This enables the use of third party command libraries to speed up the development process because many functions are already implemented, for example, matrix operations for basic linear algebra.

Proxies

In fastrigs, the objects that are placed in 3D space to indicate where a rig's important landmarks need to be during creation are referred to as proxies.
In the fastrigs that I looked at during my research, I found that there are three common ways of indicating where in 3D space joint will be created. In my first tests with the fastrig, I tried out all methods to get a feeling of the pros and cons of each approach.
The first is to allow the user to define their own proxies. Meaning that a module requires the names of X amount of objects to function. For example, an arm module might require three points (shoulder, elbow and wrist), and the user can select any three objects and run the command to create the module. This approach has the upside of being very quick and dynamic; the user can create or select the objects without having to be aware of their object type or how they got created. The drawback is that since those objects can be anything, it is very easy to lose track of which objects represent what landmark, and it is hard to establish relationships between the modules. So this technique is mostly used in rigs that are created one module at a time and then connected manually.
The second approach is directly creating joints as proxies. The user creates a joint chain with the required amount of joints (or runs a script to create a proxy with a joint chain). Then the rig gets created on the existing joint chain. The advantage is that the fastrig doesn't have to be concerned with joint orientation or joint creation since it can just use and duplicate the existing joints. However, the user needs substantial knowledge of correctly creating, moving and orienting joints to achieve good results with the rig.



Proxies created with the fast rig



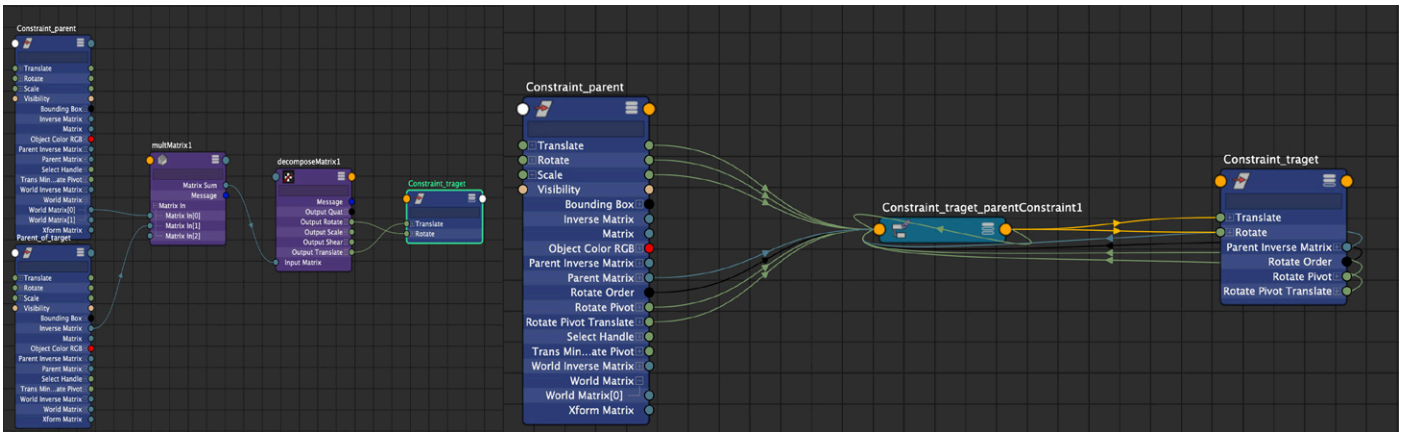
Rig created from proxies

The third approach, the one I ended up using, is to create distinctive stand-in objects via a script. Those objects already stand in relation to each other and indicate what they are representing. For my rig, I decided to use nurbs-curves as stand-ins. The design of those curves and the way they

relate to each other was heavily inspired by „Rapid Rig Modular“, a popular fastrig for beginners because of its intuitive way of representing proxies which makes them easy to track. The upside of this approach is that it gives full control to the developer. She decides how the hierarchy or the proxies function and can use this knowledge to dynamically find the objects she needs during rig creation. It also allows for very complex rigs, since the hierarchy is clearly designed and logical for the fastrig. The downside of giving this freedom to the developer usually means taking certain freedoms from the user: to safeguard against unintended behaviour of the rig, nodes in setups like this are normally locked so the hierarchy can’t easily be changed. Additional scripts must be provided for the user to safely change the parenting or the settings of a module, which are key functions of a fastrig.

Matrix Constraints

I decided to rely on matrix constraints because matrix constraints tend to compute faster than regular constraints and I wanted to learn more about the matrix calculations in Maya. Matrix constraints keep the outliner clean—the window giving an overview over the hierarchy in your scene—, since they rely on mathematical nodes that don’t have a position in 3D space instead of creating DAG (directed acyclic graph) nodes. They also lead to considerably cleaner connections in the node graph, since they don’t rely on cyclical connections like the common constraint.



Matrix-based constraint (left) and common constraint (right) in the Node Editor

However, the common constraints also have their advantages, such as not needing to know where the object stands in the hierarchy and the ability to easily reparent objects with a constraint connection without breaking the calculations. I also decided to create the aim constraints using common constraints, since creating an aim constraint with matrix constraints requires you to create an extra world-up-object, an object that defines the plane the secondary aim axis has to be in, which defies the point of keeping the outliner clean and adds a lot of overhead calculations. While it is possible to create matrix aim constraints with different world-up-types, they tended to be significantly slower and less stable than the common aim constraint.
Nevertheless, I’m glad that I set myself this restriction and decided to work with matrix constraints. It taught me a lot about how Maya calculates the position and orientation of an object. If set up correctly and the hierarchy is respected, the additional speed can make a significant difference in the speed of a complex rig.

Tagging System

Fastrigs require a system to keep track of the different objects within the script, know what they are used for and how to further use them. It is easy to store such data in code using tuples, lists or dictionaries. However, this information is only stored in active memory inside the program. This works well for objects that need to be used multiple times during the same executed function, but the information is lost as soon as the program is closed or, as tends to occasionally happen with Maya, it crashes.
There are two common ways to store this information so it wouldn't be affected when the program is closed. The first way is to store it in a real-text-file with a light format, for example, „.js" or „.pickle". This keeps your Maya scene clean and prevents the user from accidentally deleting it within Maya, but it has the downside that the project then relies on the file. The user then has to copy this file with the project when changing machines—a file they probably don't even know gets written on their hard drive.
The alternative approach is to use attributes to tag the objects within Maya. These are often string-type or message-type attributes, since those allow for storing precise data. Some scripts I analysed used a single set-object to store long strings for each tag, which contained the name of every object that needed this tag separated by a semicolon. While this has the benefit of having all your data in one easily accessible place, it brings the risk of the user deleting that data accidentally. Instead, I decided to directly tag the objects with attributes as outlined in the table below:

Tag	Type	Storing	Stored on	Used for
specification	String	The function of the object within the rig or proxy	Every object that is supposed to be manipulated by the user	Queried existence: Checks if legal object is selected Queried value: Checks if an object is affected by a running function
module_type	String	Type of the module that belongs to object	Module bases	Queried value: Checking which function needs to be called when creating rig from proxies
is_paired	String	empty	Proxy module bases that have a counterpart	Queried existence: Checking if module has counterpart to mirror values onto

Tag	Type	Storing	Stored on	Used for
parentObjectT	String	Name of object that it will constrain tranformation	Proxy module bases Module bases IK controls	Queried value: On proxy module base: To convey information from proxy to rig On module base and IK controls: To find constraint parent when connecting modules to each other
parentObjectR	String	Name of object that it will constrain rotation	Proxy module bases Module bases IK controls	Ignored if: Value matches parentObjectT Queried value: On proxy module base: To convey information from proxy to rig On module base and IK controls: To find constraint parent when connecting modules to each other

This tagging system not only keeps the user from making mistakes while creating proxies, but it is essential when creating a rig from the proxies. Since the proxies are built in a hierarchy that reflects their dependency, modules that follow other modules are directly parented underneath them. This makes the proxies faster and helps to query their relationships when the rig is created and ensures that modules get only called for creation when the module they will follow already got created. That said, I created this tagging system very early on in the process, so some of the tags could be solved more elegantly or made redundant and I will probably rework the system in the near future.

Common Modules

Most modules in this fastrig are pretty standard so they don't have to be discussed in great detail. The arm and leg use a three-chain IK/FK setup connected by direct connections. Stretching is translation based, to avoid the problems with dual quaternion weighting. The arm also comes with an FK/IK hybrid clavicle.

The head and auxiliary modules are single joints that are directly parented to the controls they follow. For the head, there is an option to create a two joint FK chain as a jaw.

The look-at module is a single joint that is aim constrained to its control

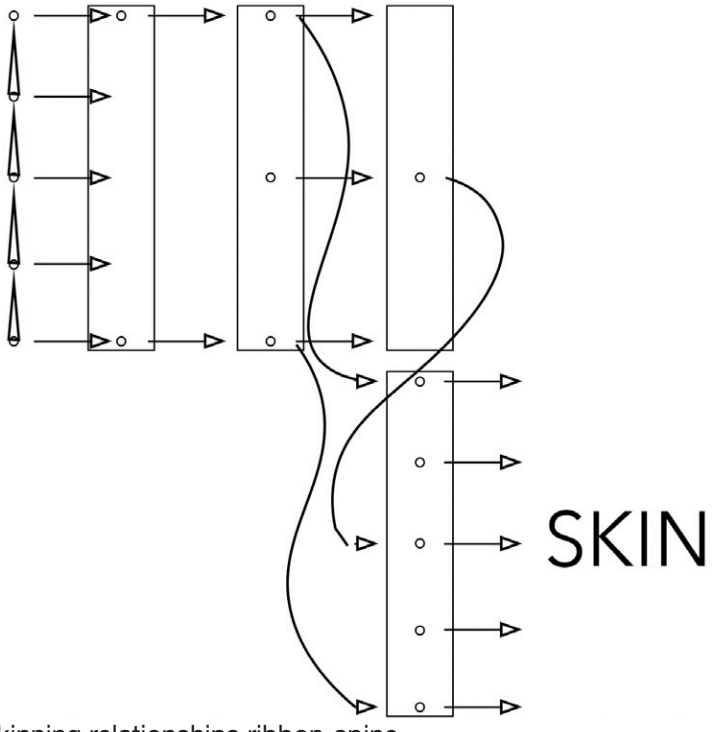
The FK chain is a single joint-chain that is directly connected to the controls in an identical hierarchy.

Two Ribbons

There are two ribbon-based systems that I will discuss in a bit more detail here.

The general definition of a ribbon, when talking about CGI-rigging, is a nurbs or polyon plane that contains joints that are bound to its surface and which is then skinned to control joints. This allows for a setup that behaves similar to IK-spline setups with automatic stretch without having to be in a chain hierarchy. In ribbons, the rotation of the skinning joints also affects the orientation of the surface joint, unlike in similar spline/curve based setups.

The first system I would like to discuss is the IK/FK-ribbon-spine module. As mentioned above, creating a simple pseudo-IK-spline setup with a ribbon is rather easy. The challenges arise if the setup should not only have an end- and a start-control, but also a middle-control. When the start or end are moved, the middle won't move since it has no hierarchical connection to those joints. Simply constraining the middle to start and end will give the desired effect when translating start or end, but not when rotating them. When the start or end are rotated, the middle should stay in place and rotate with them to some degree. Instead, the middle will move outward, to the frustration of the animator. Some setups constrain the middle to start and end in translation and use blended direct connections for rotation, but those setups won't work as soon as an FK functionality is introduced.
On top of that, introducing an FK-functionality is also tricky. Doing this in a proper IK-spline is rather easy since in those setups all the calculation is done between the end of the chain and the start of the chain, so all that is required is for the start and the end to follow the FK-chain to function properly. This, however, doesn't work in a ribbon that has free-floating joints without proper inverse-kinematic calculations.
Thus, the challenge in creating a ribbon setup with IK and FK functionality is how to connect the different elements. I looked at many cartoony rigs to find out how other people do it. During this search, I found the „Tomas" Rig by Artem Dubina[1], which contained an IK/FK-ribbon-spine like the one I wanted to build.
The way to connect the different setups is rather simple: skinning. The setup contains four ribbons. At the start of the chain is an FK chain that the first ribbon is skinned to, and each ribbon contains joints that are the skinning joints for the next ribbon in the setup. The diagram below

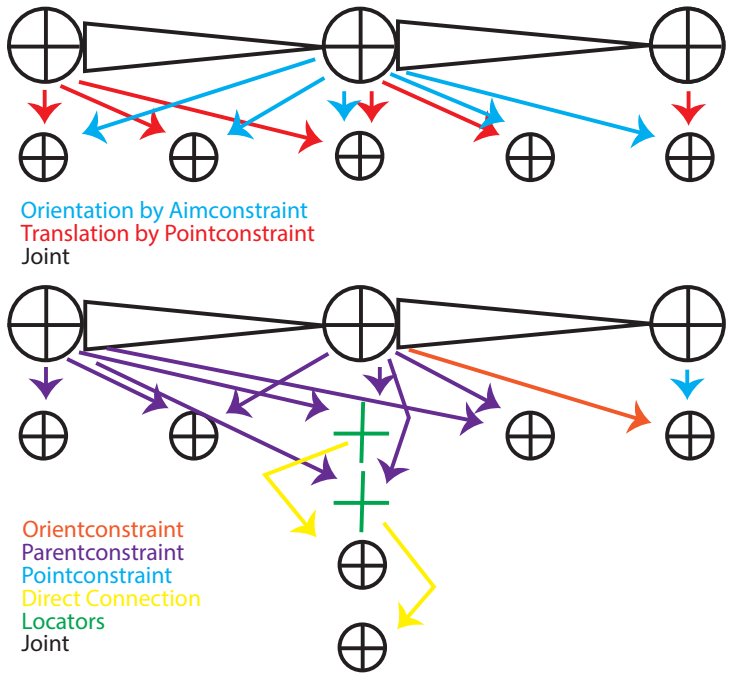


Skinning relationships ribbon-spine

1 https://animationbuffet.blogspot.com/2020/08/tom-free-maya-rig.html

explains the relationships in detail.

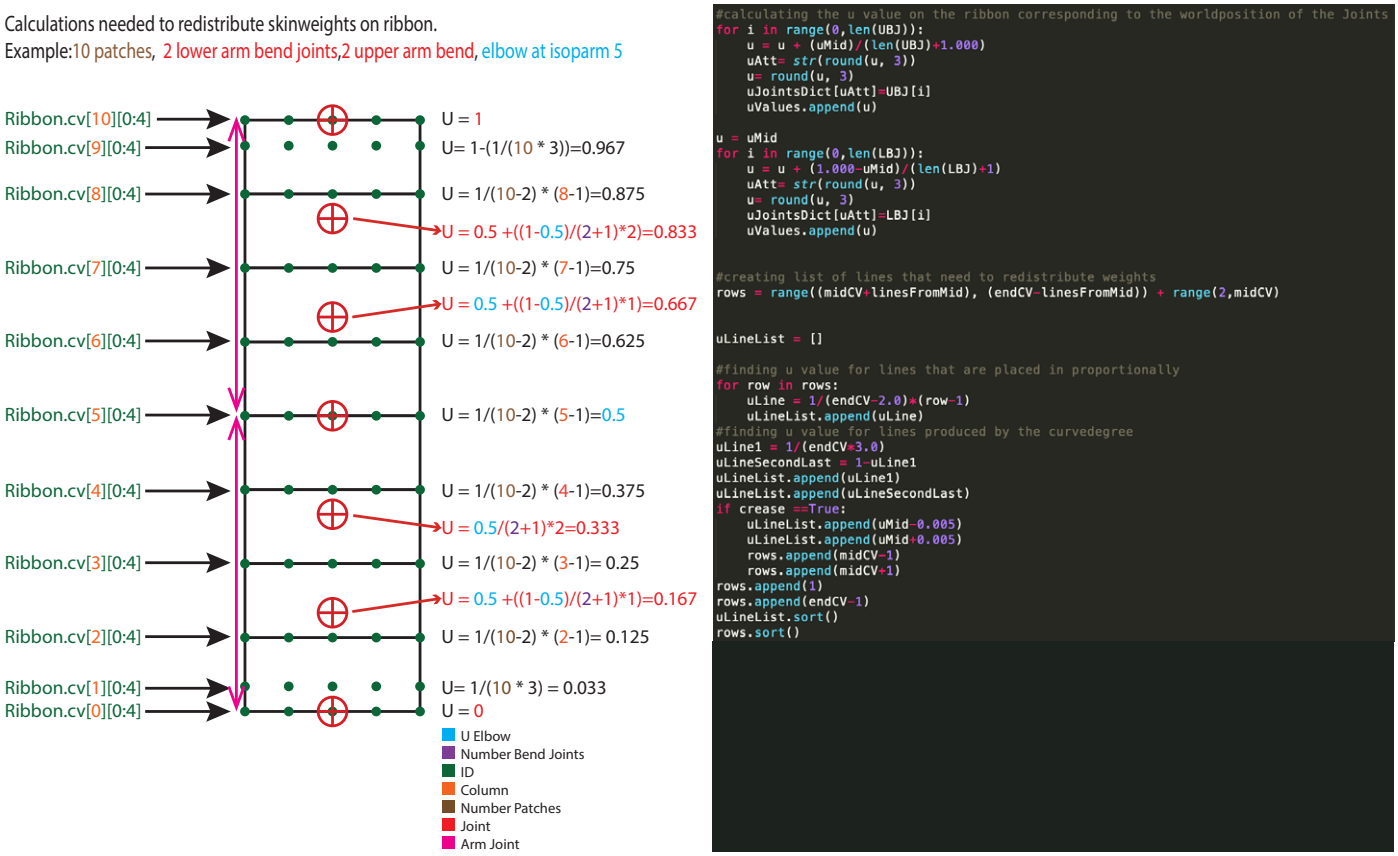
The other ribbon setup I would like to discuss is the bend setup for the limbs. It contains either two ribbons, one for the upper and one for the lower part of the limb (the „Split Ribbon“ mode) or a single ribbon, that reaches from start to end (the „Continuous Ribbon“ mode). For this documentation, I will focus on the „Continuous Ribbon“ since it brings more challenges with it and I will use an arm to illustrate it.
The ribbon initially gets created with two surface patches and in linear mode, so that it can be positioned in the correct points in space. The first isoparm (nurbs-surface equivalent to edge) gets snapped to the shoulder, the second to the elbow, and the third to the wrist. Next, the ribbon gets rebuilt in third-degree-cubic mode for softer deformations. Before that, however, it needs to be decided how many patches the ribbon requires since the ribbon optimally has patches that are all of equal surface area. First, the ratio between the lengths of the limb's parts must be considered, because, in order to stick to the elbow, the ribbon needs to have an isoparm right at the position of the elbow or very close to it. This is not the case if, for example, the upper arm is one unit long and the lower arm is seven units long, but the ribbon has only three patches. So in the case of a one-unit upper arm and a seven-unit lower arm, the ribbon would need at least eight patches: one for the upper and seven for the lower limb. However, if the lengths were two and six units instead, only four patches would be required.
Then the amount of skinning joints on each part of the limb needs to be considered since the ribbon needs to have at least as many patches as all skinning-joints combined to function properly. So in order to still have the correct ratio of patches, the number of patches previously defined gets multiplied by increasing integers until the result exceeds the total amounts of skinning joints. With this number of patches, the ribbon can finally be rebuilt.
For the bend system, four joints get created: one joint at the shoulder, two joints at the elbow (one that belongs to the upper arm and one for the lower arm), and one joint at the wrist. In each part of the arm, the number of control joints specified gets built and evenly spaced out over the length of the part of the limb. In order to introduce a curve function that bends stronger the more the limb is bent, each control joint gets constrained to both the upper and lower arm. The constraints' weights initially are set to one for the part of the limb they belong to and zero for the other part of the limb. Then attributes can be introduced that control the weight that is initially set to



transform based bendy limb setup and arc based bendy limb setup

zero: the higher the weight gets, the more the joint will be pushed outwards when the limb bends.

As I mentioned before, single-ribbon setups for bendy limbs are something that is not often seen in fastrigs, despite being a pretty common thing to do when rigging by hand. While having to calculate the number of patches certainly plays a role in this, the even harder part of a single-ribbons setup is that the weights have to be repainted with code. In a „Split Ribbon“ setup the joint's weight can be automatically assigned to the skincluster for the correct part of the arm, so the initial weight painting will be quite good to begin with. For the smoothing out that might need to be done, most fastrigs just assign a deltamush deformer, which is designed to keep tensions low in the mesh. However, this method can't be used for a single-ribbon setup. In a single ribbon setup, the two joints at the elbow will both have influences in the wrong part of the arm. Also, if the arm is bent on creation, there might be joints from the lower arm, that are closer to control vertexes on the upper arm than the next upper arm joint. So the skin-weights need to be redone to assure that the arm bends correctly.
This was probably the part that took me the longest to figure out. I needed to find out how to address control vertexes by their ID via code and then create the formula to find their U value on the ribbon, as well as how to convert the joint transformation into the same U value so I could com-




```
#calculating the u value on the ribbon corresponding to the worldposition of the Joints
for i in range(0,len(UBJ)):
    u = u + (uMid)/(len(UBJ)+1.000)
    uAtt= str(round(u, 3))
    u= round(u, 3)
    uJointsDict[uAtt]=UBJ[i]
    uValues.append(u)

u = uMid
for i in range(0,len(LBJ)):
    u = u + (1.000-uMid)/(len(LBJ)+1)
    uAtt= str(round(u, 3))
    u= round(u, 3)
    uJointsDict[uAtt]=LBJ[i]
    uValues.append(u)


#creating list of lines that need to redistribute weights
rows = range((midCV+linesFromMid), (endCV-linesFromMid)) + range(2,midCV)


uLineList = []

#finding u value for lines that are placed in proportionally
for row in rows:
    uLine = 1/(endCV-2.0)*(row-1)
    uLineList.append(uLine)
#finding u value for lines produced by the curvedegree
uLine1 = 1/(endCV*3.0)
uLineSecondLast = 1-uLine1
uLineList.append(uLine1)
uLineList.append(uLineSecondLast)
if crease ==True:
    uLineList.append(uMid-0.005)
    uLineList.append(uMid+0.005)
    rows.append(midCV-1)
    rows.append(midCV+1)
rows.append(1)
rows.append(endCV-1)
uLineList.sort()
rows.sort()
```

How to get U values of controlvertices and joints. In concept (left) and in code (right)

pare them and assign the right weights.

3. The Rigging Reel

3.1 Motivation

Since I started with 3D animation, rigging has always fascinated me—finding different ways to solve problems, figuring out how to achieve hard effects. I love learning new techniques from the online community, trying to apply them to my own projects, and figuring out how to improve on setups or finding alternative ways to implement them.
For me, being a good rigger means having an in-depth knowledge of all the tools at your disposal—how you can use and, occasionally, misuse them to achieve the desired effect, understanding how things interact and always having a plan B ready if the first solution doesn't work. It's

fun to identify problems in rigs and solve them. It's rewarding to be able to use my knowledge to help other people out, and to learn more things every day, be it through trial and error or by learning from someone else. No two rigs are the same, especially when it comes to facial rigs—the facial form and topology always requires new skills to function as good as possible. I also find it extremely satisfying when something just works. An animator's job can be made so much easier if the rig allows them to work with it instead of against it.
Before I started the masters, I had fairly good rigging skills already, but I had settled into a routine of creating all rigs with the same scheme. I didn't leave my comfort zone and stayed with effective and fast techniques instead of maybe risking having to do a process multiple times in the process of figuring out the optimal solution. „Good enough" was good enough. The master's program enabled me to branch out, try new things, and approach problems that I had an established solution to from a different angle—not always successfully, but always improving.

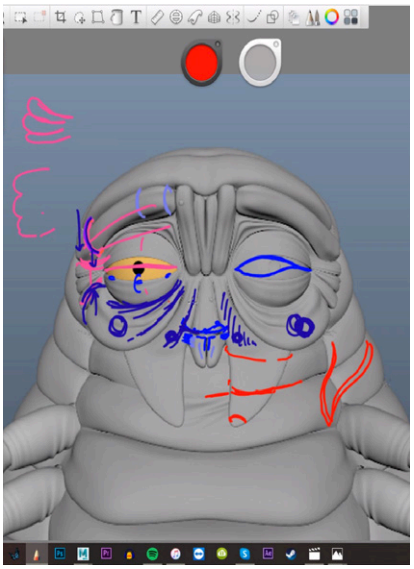
3.2 Goals

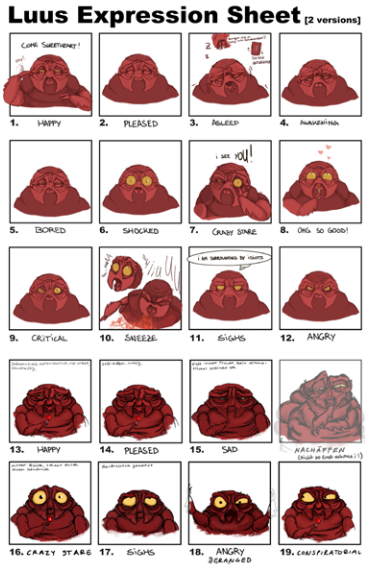
The simple technical goals I set for the rigging reel were to produce a 1.5-2.5 min reel, showcase four or five rigs, try to minimise screen recordings, and replace them with other ways to exhibit my rigs. With an eye towards potential employers, I also decided to include a non-bipedal character and to showcase facial expressions as well as body deformations.
The goal of any reel is first and foremost to testify to the skills of the person who created it. I wanted to dive deep into the different rigs for cartoony or stylised characters and more common rigs, utilise and learn new techniques with each rig, and demonstrate the versatility of my skills as a rigger to an employer.
Also, I was never really happy with my reels in the past. They lacked style, so I wanted to try and mix simple animations with the conventional controller wiggling to show the handling of the rigs and to make it entertaining. I think I achieved this goal. To this day, I never had a reel that represented me as well as this one.

3.3 The Rigs

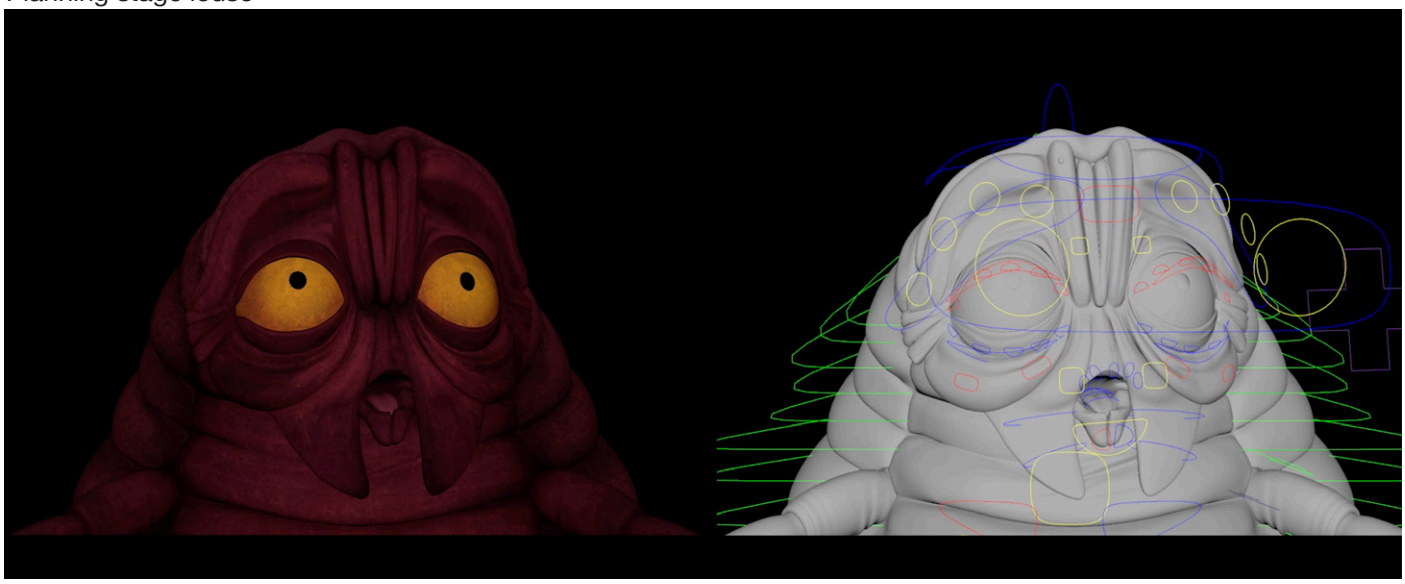
Louse - Planty of Love

During my second semester in the master's program, I was approached by Rosemarie E. Benson to rig for the art project „Planty of Love by Badel/Sarbach". The character that I needed to rig was a louse of the family dactylopius coccus, known for the red pigment carmine used in different spirits and sweets.
The model was great to work with: clean topology and very appealing to look at. The model just looked friendly, even in neutral pose. It is always great to work directly with Rosie, who was head of animation on this project. She knew exactly how she wanted the louse to move and provided me with a pose-sheet of the extremes the louse should be able to do early on, as well as some later meetings in which we discussed further changes.
The body rig for the louse was fairly standard. We knew from the beginning that she would only sit on a chair and emote, so there wasn't really a reason to do anything special with the limbs. The thing that stands out most about the body rig is the additional controls that allow her fat rolls to be deformed and moved.
The facial rig was created with a joint based setup spread out over three live blendshapes and the main face setup. Using an additional twelve corrective blendshapes to create better deformations around the skin flaps that hang over her mouth corners and around the inner part of her eyebrows. Most notable in this rig's facial topology is probably the middle section: she has no nose and instead of a lower lip she has two pincers. There are also four permanent folds between the eyebrows, in which I engineered three joint based controls that allowed them to deform. This brought the expressiveness of the rig to the next level.





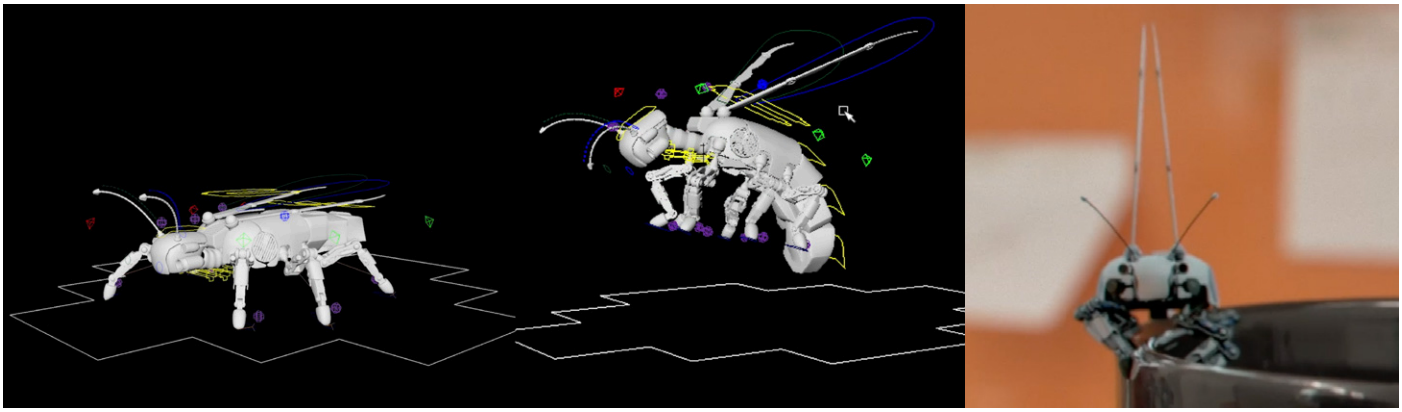
Planning stage louse



Louse finished rig

Robo Bug - Fried

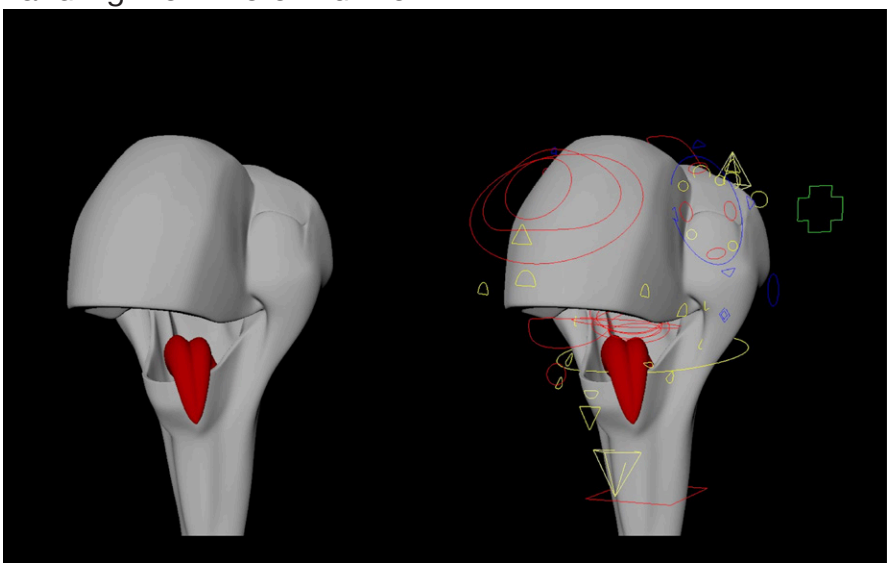
Fried was a short I created together with Liam Carter during the TM1 in the masters. The character is a robot bug heavily inspired by fireflies and mayflies.
The rig is pretty straightforward; it has an IK-ribbon neck, FK-tail, FK-wings, and FK-antenna.
The Legs are IK/FK, double knee legs, using a layered IK-rotate-plane solver and IK-spring solver solution for the IK functionality.
The legs are the main reason that the Bug is included in the reel. A good rigging reel should always include a character with double knees since this is something that every quadrupedal mammal requires.

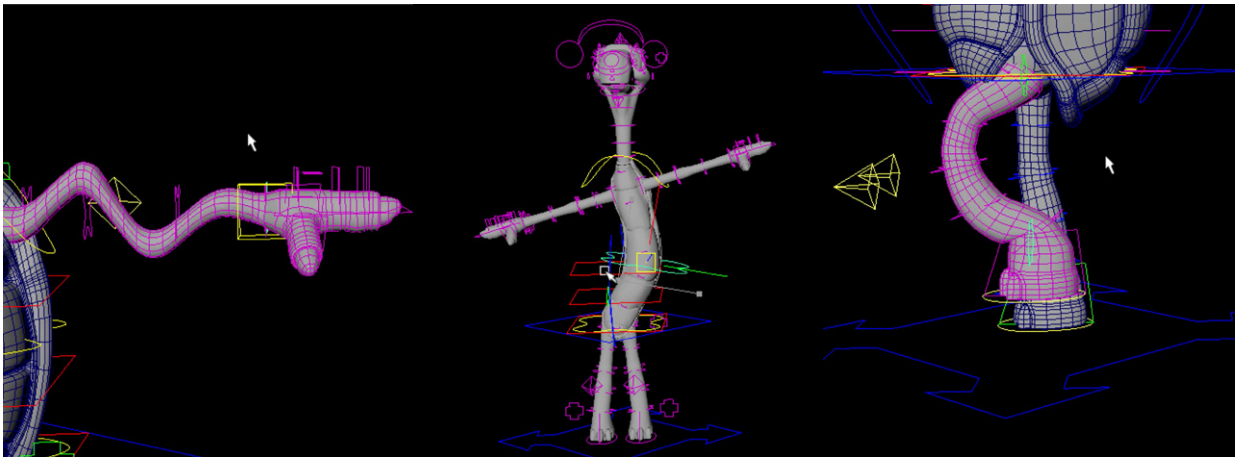


Robo bug rig and final render

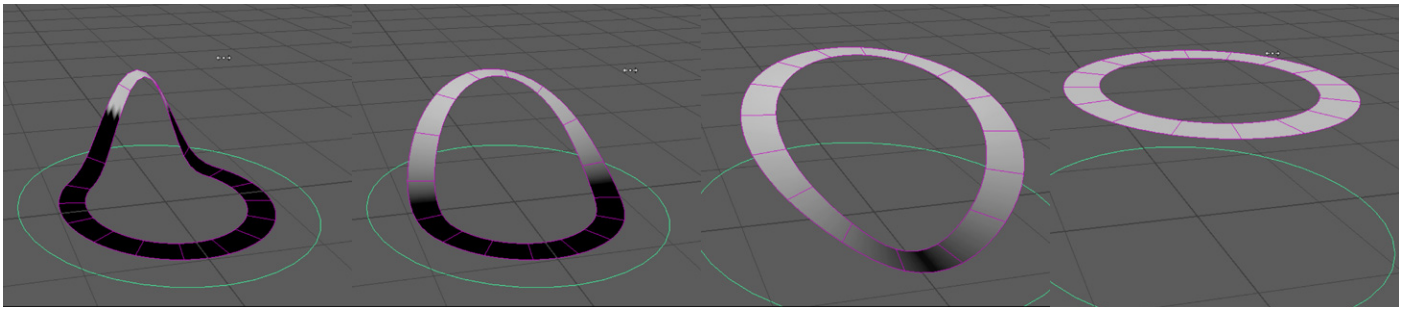
Theodore

Theodore was initially designed for the film project I worked on in the first year of my master's degree. I wanted him to be able to behave extremely cartoony—stretching, squashing, being able to rotate his body more than 180 degrees like a cork-screw, bendy limbs with additional deformers—the whole seven yards. I created this rig mostly during my rigging-dojo apprenticeship course with Hector Abraham Torres. Hector works at DreamWorks Animation and was very knowledgeable and helpful when it comes to cartoony rigging. I learned a lot during these five weeks of training.
Theodore has a lot of features I never created before. There's the arc-based bend system for arms and legs that reacts to the bending of the limb rather than the bend controls being moved. He has scalable hands and feet that allow for shots with forced perspective. He also was the first rig that I used the ribbon-spine from my fastrig on. He also comes with separate files for multiples that can be loaded in if a cartoony animation requires multiple arms, legs, or heads, as well as a running multiple: white multiple feet on the same set up that make it seem like he is running fast. The upper half of his facial rig is joint-based and uses a spline-based eye rig, while the mouth is blendshape-based and uses a joint-based ribbon setup for tweaking. Since this was the first time I created a blendshape-based facial rig that had joint-based controls on top of it rather than the other way around, it was also the first time I had to work with rivets and transform negation to keep some of the controls close to the mesh they deform.
The facial rig also has post-skin deformers. It has a total of six lattices that deform the mesh after the skin cluster deformations: one for the top half of the face, one for the lower half of the face and two to deform each eye. The lattices are also set up using local blendshape rigs to make handling them more intuitive.

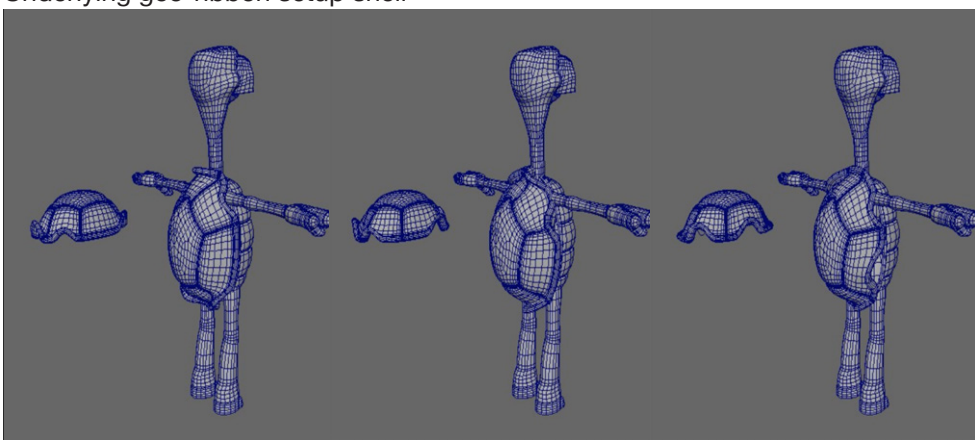

Theodore facial deformations


Theodore body features

Lastly, I should talk about Theodore‘s shell. The shell was the module that I probably spent the most time on. Initially, I had the idea to have the holes in the shells be completely free moving: they could open, close and merge with each other, making the number of holes variable and giving it potential for a lot of cartoony jokes. I first tried to achieve this using sine-deformers on splines which then would deform the mesh using a wire deformer. The problem with this setup was that the sine-deformers would cumulate when they started overlapping, which gave undesirable results. I encountered the same problem with all the other non-linear deformers. This was when I found a tutorial for a dynamic ribbon[2], which used a texture to deform the ribbon, using the red-value of a texture to determine the movement of the control vertexes. This concept was very interesting, but the author had marked the tutorial as „unfeasible“ since he needed expressos to read out the colour values which would slow down rigs to a crawl when multiple of those ribbons were in there. Apart from that, it seemed unproblematic, so I tried to come up with my own version of this setup without using expressos. This ultimately worked by using a mesh-ribbon instead of a nurbs-ribbon and reading out the texture values using a texture deformer. This was significantly faster than the expresso setup. However, I had come to realise that this setup wasn’t great for as many holes as I wanted in the shell, because there would have to be extremely high mesh resolution on both the shell and the ribbon and I had to locate 80+ joints on the ribbon to make the deformations of the shell smooth at any given position. This was again unfeasible on top of a full rig, so I ended up making the holes open, close and scale up using blendshapes. A lot less versatile than the other setup, but robust and fast enough to not slow down the rig.



Underlying geo-ribbon setup shell



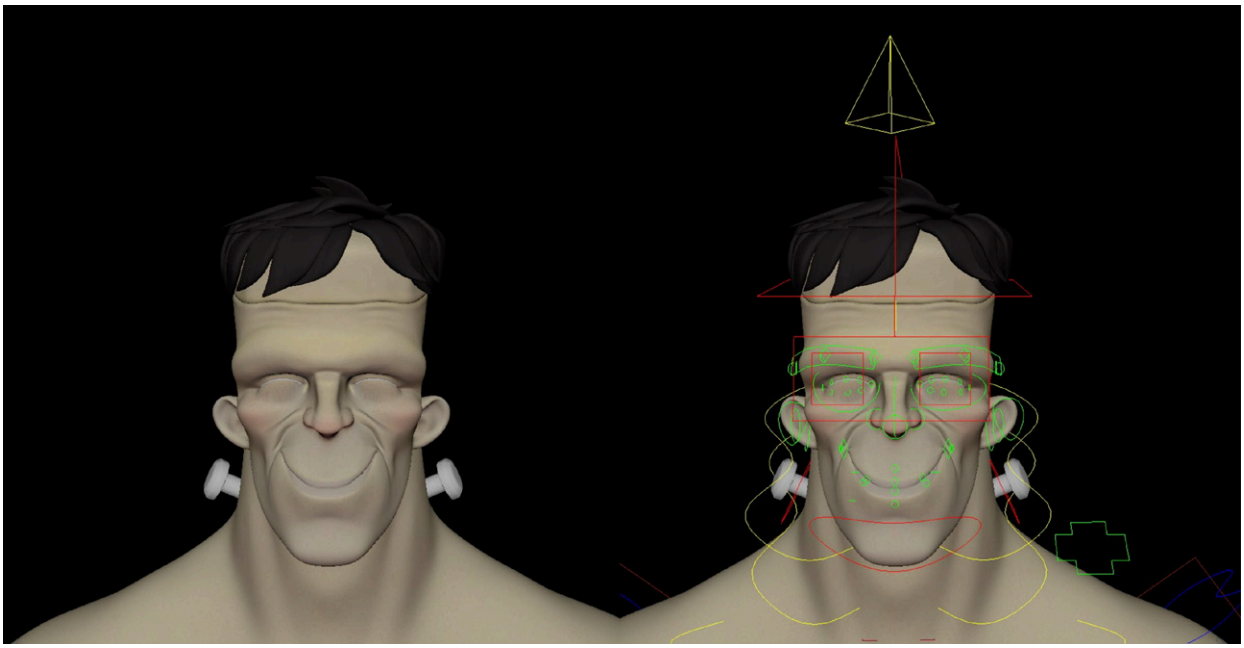
Proof of concept shell

Frank

While working on Theodore with Hector, we realised that Theodore’s model simply didn’t allow for good front on facial deformation and readable lip-synch—turtles aren’t known for their facial dexterity. So he suggested a model from a friend of his, to use a similar technique to the one I used with Theodore to exhibit nice facial deformations in my reel. Frank is based on Frankenstein’s monster and is a model by Makar Maliki.
When I saw the model, I knew right away that I wanted to rework the geometry so he could take

2 source has been deleted since (blogpost that references it: https://lesterbanks.com/2011/09/creating-a-flexible-ribbon-in-maya/)

the top of his head off and reveal his brain—perform some more surgery on him, so to speak. Franks facial rig is blendshape-based in both mouth and eyebrows. The blendshapes allow for better control in the skinfolds and a good base deformation which then can be tweaked using the joint-based ribbon controls around the mouth and free-floating joint controls in the eyebrows. The eye rig is based on the same principle as Theodore, using nurbs curves and wire deformers. He also has a lattice to deform the upper part of his head. The top of his head can be taken off using a joint-based control and the brain inside his head can be moved using a joint-based control and deformed using a lattice-based setup. The top of his head, the brain, and the upper part of the face are all connected to the same lattice setup so they deform with each other, but connected through separate deformer nodes so the influence can be turned off individually.
Franks body rig brings both of my projects together since it was the first full-body rig that was created using my fastrig. He has continous-ribbon setups in the arms and legs for bending and twisting, an IK/FK Ribbon spine and scalable hands and feet. He's proof that the prototype of my fastrig works.



Frank facial setup

4. Reflection and Outlook

During my master's degree, I started out with creating a short film project, but ultimately realised that I wanted to spend the time on improving my skills in rigging, learning new things and gaining more experience in the ones I already knew. It was great that I could take the time to focus on the gaps in my knowledge without having to think about project plans or picture locks.
My fastrig and my rigging reel are both successful projects that are testaments to the things I learned and the ways I improved over the last 2.5 years. I'm hoping that my reel will help me to find a position in the industry as a rigger where I can contribute to pushing the frontiers of the craft. It is certainly more marketable than the last reel I had.
My fastrig is far from done. Creating a script-only prototype for bipeds was a good goal for the masters, but I would like it to ultimately be able to rig any kind of creature. For that, I will have to add new modules, like double knee legs, wings and so on. I also would like to publish my fastrig for people who want to use it, but I'll need to create a comprehensive UI for that, as well as ironing out some of the kinks that it currently has and that impact its user-friendliness. I'm looking forward to improving it and to learning more in the field of automating rigging processes.