Media Engineering and Technology Faculty
German University in Cairo

Character Rigging in Maya Using Python

Bachelor Thesis

Author: Omar Zaki
Supervisor: Assoc. Prof. Dr. Rimon Elias
Submission Date: 12 June, 2022

Media Engineering and Technology Faculty
German University in Cairo

Character Rigging in Maya Using Python

Bachelor Thesis

Author: Omar Zaki
Supervisor: Assoc. Prof. Dr. Rimon Elias
Submission Date: 12 June, 2022

This is to certify that:

(i) the thesis comprises only my original work toward the Bachelor Degree

(ii) due acknowledgement has been made in the text to all other material used

---

Omar Zaki
12 June, 2022

Acknowledgments

This project would not have been possible without the support of many people. Many thanks to my supervisor, Assoc. Prof. Dr. Rimon Elias, who read my numerous revisions and helped me make some sense of the confusion. Thank you for your patience, guidance, and support. I have benefited greatly from your knowledge and meticulous editing. I would also like to thank the teacher's assistant, Eng. Menrit for her dedication and guidance.

I am thankful for my friends and colleagues who helped critique my work and provide insightful advice during the time spent finishing this. Their help was vital to finishing this paper.

Most importantly, I am grateful for my family's unconditional, unequivocal, and loving support.

Abstract

In recent years, the application of animated characters has diversified in multiple projects and businesses. Including video games, virtual reality, and animated movies, the technology sector day by day escalate their rigged skins demand, since they need to be incorporated in their work. To uphold this demand, this paper insists on providing a sustainable automatic character rigging system that complies with the companies needs, to provide excellent working skin mechanisms in a brief period of time.

Contents

Chapter 1

Introduction

1.1 Problem Definition

Character rigging artists will take the static image from a modeller to then create separate rigs for each part of the character, such as their facial expressions and limb movements [6]. The rigging of a skin or a character whether to implement in a video game, an animation or a movie can take up a lot of time and effort. Adjusting the bones and joints to the correct dimensions of the provided skin is definitely not an easy task. Using AutoDesk Maya [5] (Maya), artists will attempt to create a rig manually using Maya's interface, which takes a protracted period of time.

1.2 Aim

Animating characters is already a heavily time-consuming process. With making use of Python, we will create an auto rigger that is not only a time saviour, but it will also keep glitches and errors to a minimum.

Our goal for this project is to create an Auto Rigger which will immensely ease the job required by character rigging or animation artists. The point of the rig is to make use of Maya's script editor and the Python language to create a friendly User Interface (UI) with clear buttons and instructions to seamlessly construct a skeleton rig for the provided character. It is vital for the constructed rig to match the vertices of the skin perfectly to minimize the possibilities of bug or glitches when parts of the skeleton transition or rotate during animation. If applied incorrectly, the skin may stretch and deform when moved.

1.3 Organization

We will first start by giving some information about rigging in Maya within the Background chapter of this thesis. Moreover, different terms related to this topic will be thoroughly explained so as to easily keep up during reading the rest of the thesis.

Afterwards, in the implementation chapter, we will go through step by step on how the UI was constructed, the `Auto Rigger` class, its helper classes and the important lines of code within the scripts. These are categorized into different subsections beneath the chapter so as to easily follow the methods that were used.

Chapter 2

Background

2.1 AutoDesk Maya

During this project, we will be using AutoDesk Maya to create an Auto Rigger for skins imported into the aforementioned program. Maya is an animation and modelling application for creating full-motion three-dimensional (3-D) effects. To govern the behaviour of virtual objects in computer animation, Maya uses natural physics rules. The program is capable of creating videos that are more lifelike than previously achievable with less advanced software. Maya enhances facial expressions and the realism of body language, making it easier to represent emotions in animated characters. The virtual software has been used in numerous movies, including Twister and Stuart Little.

2.2 MEL & Python

The script language Maya Embedded Language (MEL) is not only a feature of Autodesk Maya software, it is the foundation. Every function in Maya is a MEL command that can be accessed using menus, icons, buttons, marking menus, and other controls. However, in this project we will be using Python, a high-level, general-purpose programming language, instead of MEL. In Python, more data structures are available, it is better for file i/o and string manipulation, tons of other libraries available for maths, network connections, file i/o, and more. It is also universally relevant; and as a result, other inexperienced MEL users can understand the coding without having to learn a new programming language. Maya makes this feature easily accessible as we only have to open a new tab in the script editor and two options will be presented, MEL and Python, and we will select Python. From now on, every line of code will be treated as Python syntax.

2.3 Skeletal Rig

Skeletal animation or rigging is a technique in computer animation in which a character (or other articulated object) is represented in two parts: a surface representation used to draw the character (called the mesh or skin) and a hierarchical set of interconnected parts (called bones, and collectively forming the skeleton or rig using joints), a virtual armature used to animate (pose and key-frame) the mesh. While this technique is often used to animate humans and other organic figures, it only serves to make the animation process easier and more intuitive, and the same technique can be used to control the deformation of any object—such as a door, a spoon, a building, or a galaxy. However, in our case, we will be rigging a humanoid character.

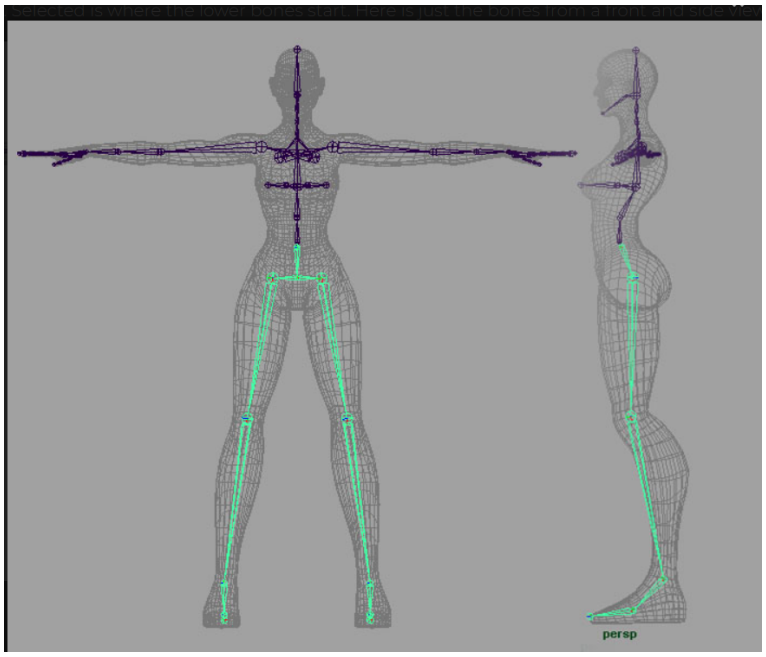

Figure 2.1: Skeletal Rig

2.4 Joints

Maya joints are a type of deformer that are designed to transfer rotational values to points on a model based on each joint's influence on those points. Joints are most often arranged in a hierarchy–meaning that one joint is a child or farther down on the hierarchy than its parent. It is important that each joint points down a selected axis towards its child so when the rotation by the animator or user is applied to the parent joint, its children move in the correct rotational axis accordingly.

Bones do not have nodes, and they do not have a physical or calculable presence in your scene. Bones are only visual cues that illustrate the relationships between joints, they cannot be controlled or manipulated. The behaviour of joints is determined by a variety of joint characteristics. You can limit how far a joint can rotate or restrict which planes it can rotate around by modifying its properties. If the rotational value of the shoulder joint is limited to 270 degrees and the joint is set up correctly in the X-axis, this means that when the joint is rotated, the whole arm will stop rotating any point that exceeds 270 degrees to the starting position, since that is similar to what a human arm can do; thus, avoiding glitches and maintaining realism.

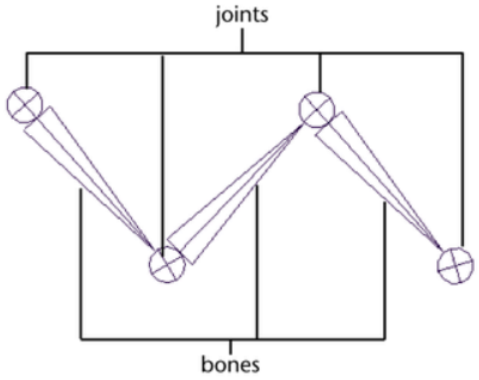

Figure 2.2: Bones and Joints Illustration

2.5 Connections and Constraints

The natural step to follow on from creating the skeletal rig would be to begin looking into controlling those joints. ideally, with any rig, you do not want the animator to be able to directly select the joints, they should be hidden away and instead controlled with custom icons or other attributes. We will be discussing how to create custom control icons, but those controls need to be able to drive the joints. This is where Connections and Constraints come in.

In Maya, you can use one attribute to control another or many other attributes. To create a connector, you design a circle or any other shape using nodes primitive in Maya and use the node editor, which presents an editable schematic of the dependency graph displaying nodes and the connections between their attributes, to link it with the joint. Hence, you can connect all the rotational and transitions or select between them. You can also connect different axes together or single them out. On the other hand, constraints are more reliable, because the node editor can fail when trying to maintain offsets between the joint and the control. Moreover, constraint functions are marginally simpler than the node editor. Furthermore, transforms cannot be frozen on joints, when trying to freeze transforms on the control, it will reset the joint back to 0,0,0 in the world view.

This can be solved using a utility node; however, implementing constraints from the get go avoids this problem and is way less time-consuming.

There are different types of constraints, Point A point constraint connects the translate values of one object to another, For example:

- Orient constraint that connects the rotate values of one object to another.
- Scale constraint that connects the scale values of one object to another.
- Aim constraint that aims one object at another object, in which it is useful for the rigging of the eyes to look at objects as an example.
- Parent constraint works similar to the parent function in Maya, that allows you to temporary associate objects with other objects, but lets you offset the child object from the parent. Moreover, this can be turned on and off and select the certain attributes to be affected, which couldn't be done using the original parent function.
- Geometry constraint that ties the position of one object to the surface of another. The pivot point of the constrained object determines where the constrained object sits on the surface.
- Normal constraint that constrains an object's orientation values so that they align with the normals of another surface. When used in conjunction with a geometry constraint, the normal constraint is very useful for moving characters or vehicles over bumpy surfaces.
- Tangent constraint that constrains an object's orientation in the direction of a curve. if the constrained object is attached to a path, a tangent constraint ensures that the object always points in the direction of the path curve.
- Pole Vector constraint that allows you to use an object to control the pole vector value of an IK Rotate Plane handle.

2.6 Movement and Kinematics

For fluid and realistic movement, we should incorporate Inverse Kinematics (IK).
Kinematics is the study of motion without considering the cause of the motion, such as forces and torques. Inverse kinematics is the use of kinematic equations to determine the motion of a robot to reach a desired position. For example, to perform automated bin picking, a robotic arm used in a manufacturing line needs precise motion from an initial position to a desired position between bins and manufacturing machines. The grasping end of a robot arm is designated as the end-effector. The robot configuration is a list of joint positions that are within the position limits of the robot model and do not violate any constraints the robot has [4].

Given the desired robot's end-effector positions, IK can determine an appropriate joint configuration for which the end-effectors move to the target pose.

2.7 Human Skeleton

The supreme reference of this project is the human skeleton. The joints allow it to rotate and move, and its bones carry the flesh. In a similar case, the Maya joints provide rotation and movement and all the skin's vertices are bound to the bones. The pelvis is regarded as our main joint in which all other joints connect to, since it is the biggest and connects the most.

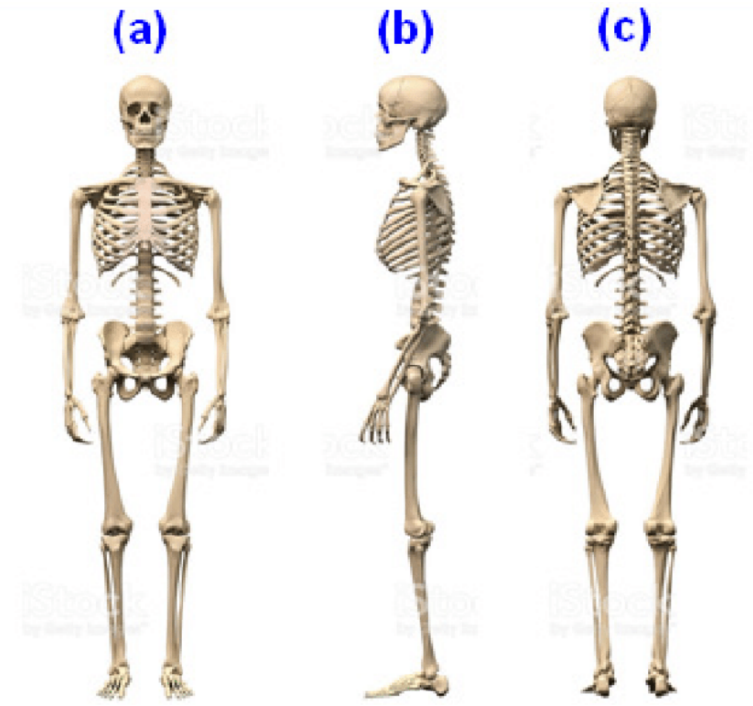

Figure 2.3: Bone structure and skeletal system of human body (a) Anterior (front side) view, (b) Left side view and (c) Posterior (back side) view

Chapter 3

Implementation

3.1 Using Python

To start the project, we begin coding on the Maya script editor provided. We open up a new tab in the script editor and choose Python. In order to use Python's library in Maya, its bindings for all the native Maya commands are in the `maya.cmds` module. Therefore, in order to access these commands, we will enter the following code:

```
import maya.cmds as base
```

3.2 Making the Initial GUI Window

To make it as easy as possible for the user to work with the auto rigger, we will have to create a UI window for the user to interact with to rig his chosen mesh. We then create a window to be able to place buttons and further instructions on it to create the rig and then further down the project editing it, providing constraints for the skeleton, then creating the IK controls to manage its movement.

3.3 Locators

Initially, it would be best to create what we call Locators. Since the user inputs a mesh or skin of different size according to his needs, it is not practical for the auto rigger to create a rig with one size only since only then the skeleton would be too large or small for the mesh and hence the skin would not bind correctly; resulting in the stretching of the skin when moving the joints or clipping and outputting weird results in the animation. Locators are then used to pinpoint where in the mesh each joint is placed.

A button called `Create Base Locators` is then fixed in the Rig Window, which starts the process by first creating a root locator using the following code:

```
root = base.spaceLocator(n='Loc_ROOT')
base.scale(0.1,0.1,0.1, root)
base.move(0,1.5,0, root)
```

Where `base` is the Python CMD and `Loc_` is the way to keep track of all locators created if needed to be dealt with later on in the project. Since we are using a script, it is easier to loop around the code and classifying items and editing them if they are tagged correctly. We then set the preference to meters in the Maya editor and scale them accordingly using the scale method. We then create a group called `Loc_Master`, and we parent any locators created to this group, so it would be organized in a hierarchy. Therefore, if we need to grab any of the locators, we just use the following method that places all the locators in a list of our choosing:

```
nodes = base.ls("Loc_*")
```

The `Root` locator will connect to the spine; hence, it is vital for all the locators to be parented to the `Root` or a child of the `Root`, in case the user decides to move all the locators at once. They can just use the `Root` locator to displace the whole body, as to keep the dimension of the rig the same.

We then request the user to input the number of spine joints required to allow for better mobility of the mesh, with a minimum of 1 and maximum of 10 joints, with the default amount set to 4. This is all done in the Rig Settings tab of the window.

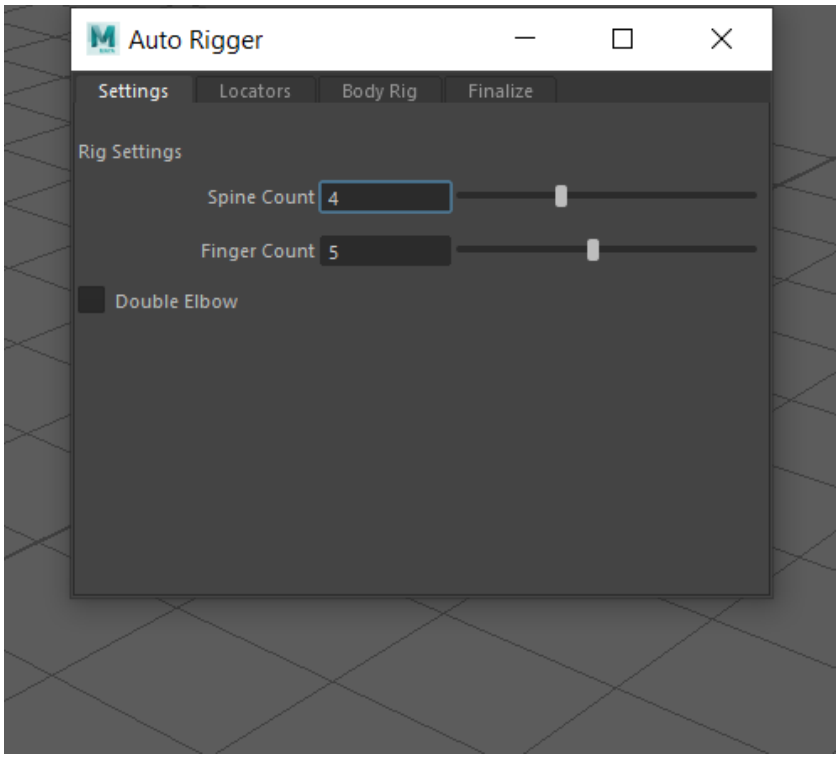

Figure 3.1: Settings Window

The user then inputs his number and the amount of spine locators will be saved upon creation, with one parented to the `Root` and the others parented to the spine locator constructed before it.

```
base.parent(spine, 'Loc_SPINE_' + str(i - 1))
```

They will all be separated by 0.25 meters intervals as to separate them enough for the back to be able to bend forwards or backwards without having to move too many joints, but maintaining the human back arc that it displays while bending.

We then proceed to call the method responsible for creating arm locators, making a left group with its locators getting an `L_ prefix` and a right group with a `R_prefix`. Our next step is to construct the clavicle locators; however, we need to figure out the amount of spine joints the user selected, since the clavicle should be place right next to the top spine joint. Therefore, we use the following code:

```
clavicle = base.spaceLocator(n = 'Loc_L_Clavicle')

base.scale(0.1,0.1,0.1, clavicle)

base.parent(clavicle, 'Loc_SPINE_'+ str(spineCount -1))

base.move(0.1 * side, 1.5 + (0.25 * spineCount), 0.1, clavicle)
```

Where `spineCount` is the variable containing the amount of spine joints selected by the user, and `side` dictates whether this is the left or right clavicle.

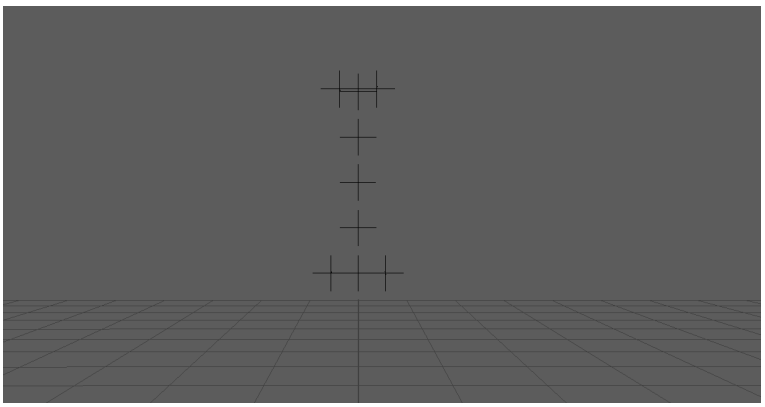

Figure 3.2: Spine, Pelvis and Clavicle Locators

Afterwards, we create the upper arm locators, the shoulders in the human skeleton. These locators are in turn parented to the clavicles. Subsequently, we make the elbow locators and parent them to the upper arms. We have added a `Double Joint` tick box in the `Settings` window. Certain hinge joints are not actually rotating around a fixed point. Double joints in rigs help to add this offset. It's not exactly true to life, but it's better than a single hinge joint, and stops the squashing up of the elbow as it reaches its most bent poses. Then we create the wrist locators and parent them to the elbow.

The next step is to create the fingers. Another slider is projected on the `Settings` window in case the user wants to select the number of fingers, ranging from 1 to 10 and the default set to 5. This is due to the fact that many characters may have different sets of fingers. In each finger, we make four locators to simulate the real thing. Our fingers only consist of three joints; however, in Maya, we need a fourth joint to represent the tip of the finger. The first locator is parented to the wrist, and subsequently the next locators created are parented to the ones made before it.

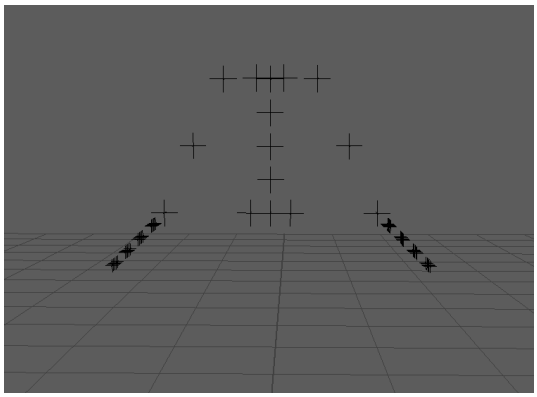

Figure 3.3: Finger Locator Frontal View

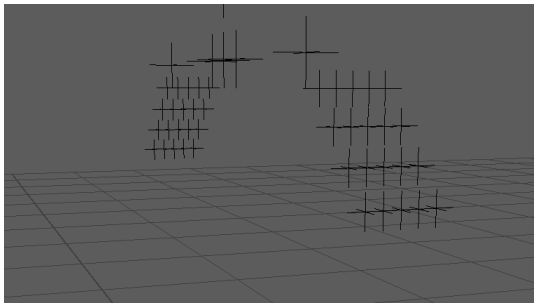

Figure 3.4: Finger Locator Side View

We then proceed to make the neck locators. There exists two, a `Neck Start` and a `Neck End`, to account for the neck bending whether upwards or downwards.

Consequently, we make the head and parent it to the `Neck End`. In addition to the head, we create jaw locators. Constructed of a `Jaw Start`, which is parented to the head, and a `Jaw End`, it is important to include these joints in case the skin accommodates a mouth. Accordingly, the jaw will be used to mimic a talking character.

As for the legs, we create an `Upper Leg` locator, which is parented and adjacent to the `Root` locator, in both X-axis directions. We then add a `Lower Leg` locator to simulate the knee and parent it to the `Upper Leg`. The foot is then constructed using three locators, the `Foot`, the `FootBall` and the `Toes`. Connected in a straight line, these locators are

responsible for representing the way in which a foot moves in real life. The ball of the foot helps add a curve that we naturally possess in our feet.

We will also implement a reverse foot roll. The reverse foot allows us to create all the necessary motion needed for a foot to plant and take off during a walk, a run or any other type of locomotion where the foot comes in contact with a ground or wall plane [3]. This works by adding four new locators in each foot. The heels, ankles, toes and foot balls.

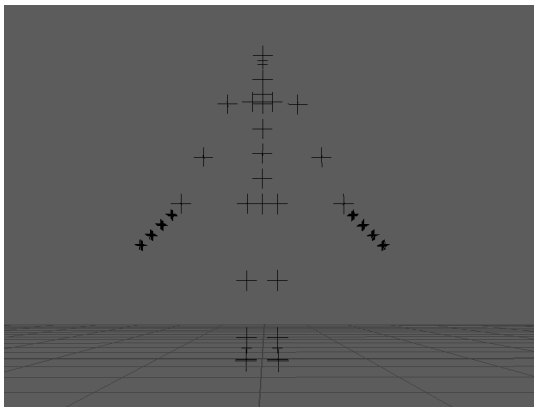

Figure 3.5: Final Locators Front View

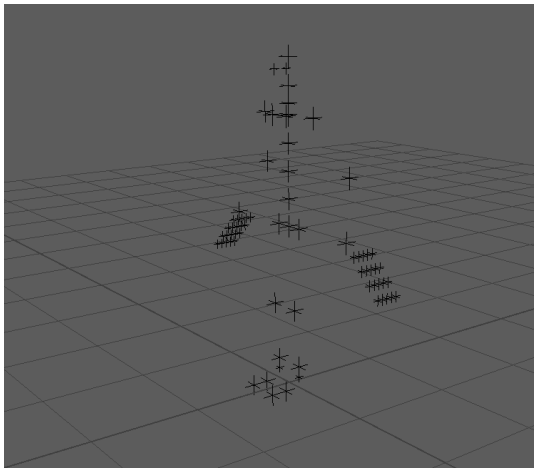

Figure 3.6: Final Locators Side View

Moreover, there is a `Mirror Locators L --> R` button, which calls a method, `Mirror Locators`, that retains the world space coordinate of every locator within the left locators group. Then, it inverts the value of each joint's X-position; hence, mirroring them all. This succeeds at saving the user time and help improve accuracy by ensuring the symmetry between both sides of the character. Furthermore, it accounts for the rotation of the fingers, if any.

In case the user would like to start over, there is a `Delete All Locators` button in the `Locators` window, which would delete the `Loc_Master` group.

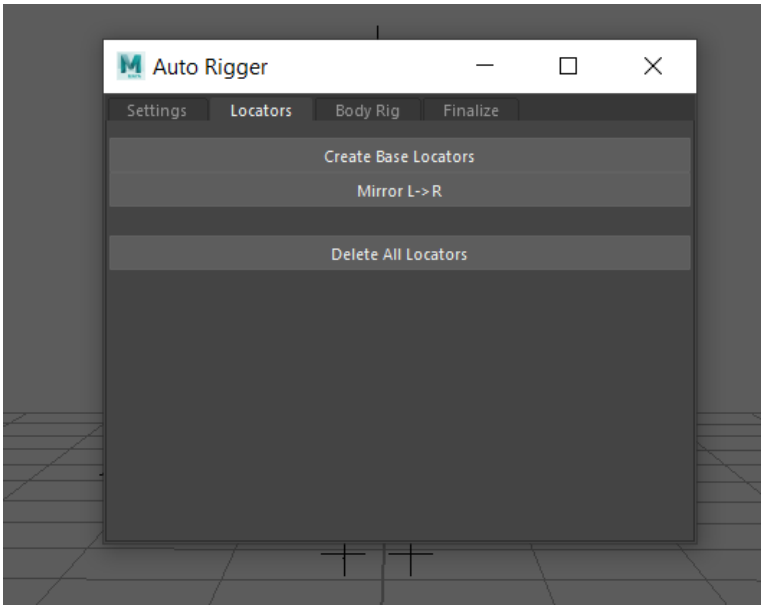

Figure 3.7: Locators Window

3.4 Joint Creation

Now that locators are done and dusted, our next step is to implement the skeleton. In the `Body Rig` window, we will add a `Create Joints` button and a `Delete Joints` button.

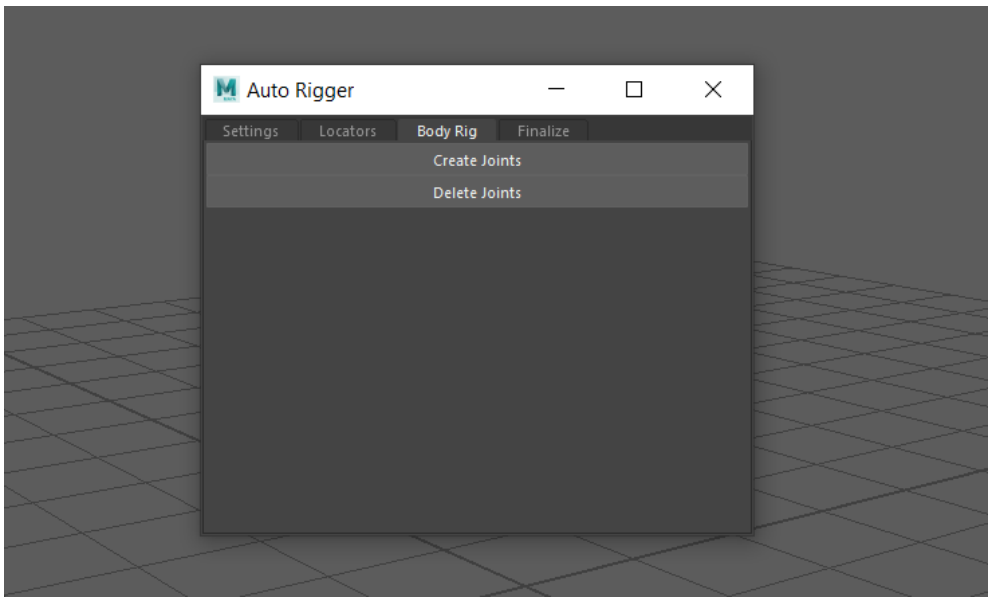

Figure 3.8: Joints Window

The method, called by the `Create Joints` button, initially creates a group called `RIG`, which contains all the joints that will be later created. This helps clean up the scene and organize our objects in the outliner.

We then start with the `Root` locator and make a root joint. The root joint represents the pelvis in the human body. We get the `Root` locator position and then use the following method to create the joint:

```
rootJoint = base.joint(radius = 0.1, p = rootPos, name = RIG_ROOT)
```

As a result, the locator will be replaced with a joint in the same position within the scene and the `Root` joint will be parented to the `RIG` in the outliner. Concerning the spine and fingers locators, we must retrieve the objects in a list using the following code:

```
allSpines = base.ls("Loc_SPINE_*", type='locator')

spine = base.listRelatives(*allSpines, p = True, f = True)
```

The `ls` function collects all the locator objects in the scene. However, it also crams the `allSpines` list with the shapes of the locators.

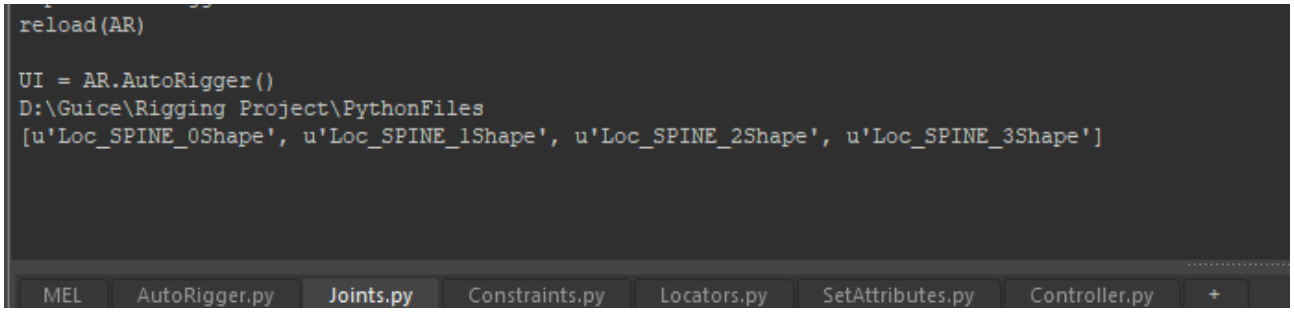

Figure 3.9: Shapes List Printed

To overcome this, we resort to the `listRelatives` function. It retrieves the actual objects because they are classified as the parents of the shapes within the list. We then create the spine joints and parent them to the root joint, all while setting their absolute value to true, since it is the only way to output their correct arrangement in the hierarchy. Afterwards, we repeat the aforementioned process with the `Clavicles`, `Upper and Lower Arms Elbows`, `Wrists`, `Hands`, `Upper and Lower Legs` and `Feet`.

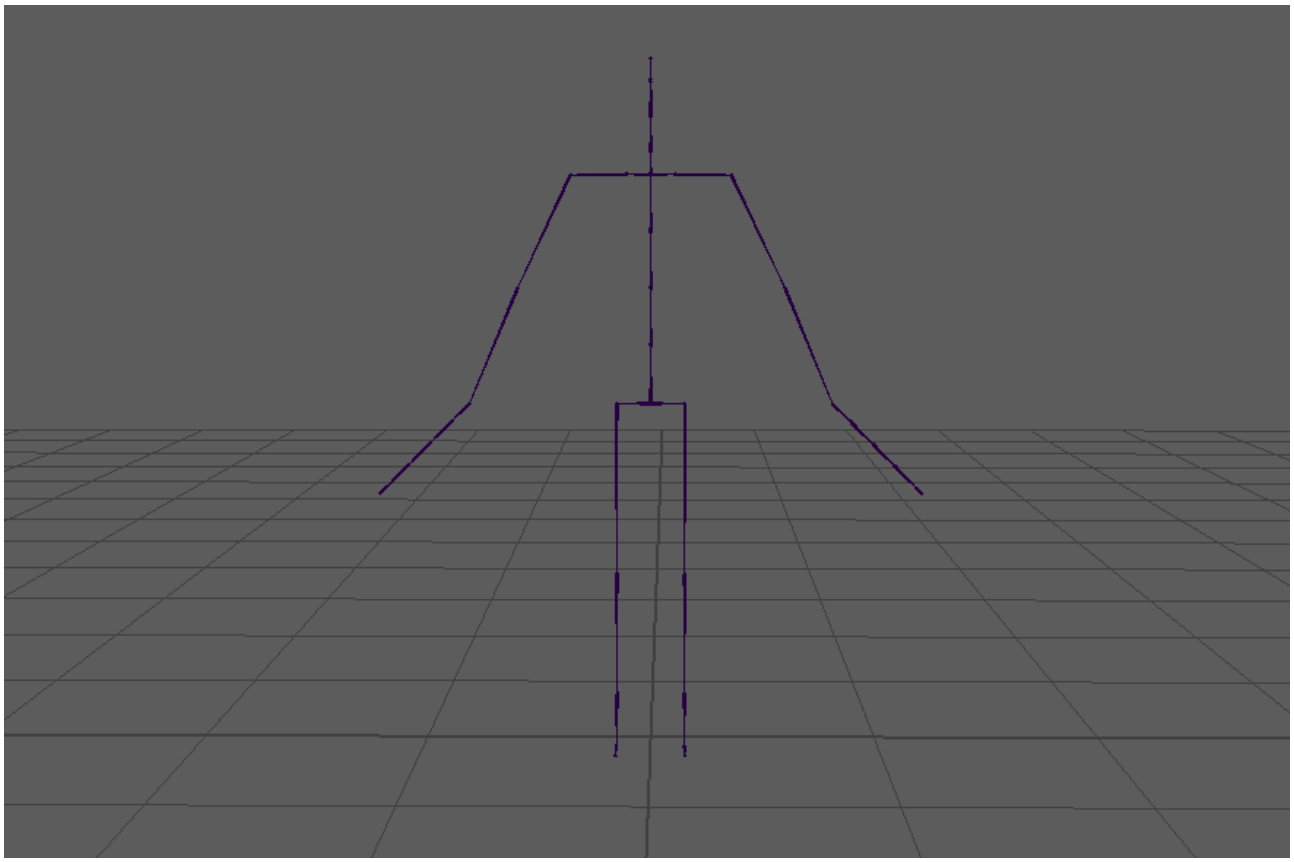

Figure 3.10: Rig

3.5 Joint Orientation

As a default, the joint orientation is set to the world's coordinates, meaning its x, y and z axes align with that of the scene's. That is fine for motion capture rigs; however, that is not the purpose of this project. Every joint's axis has to be customized. When the `RIG` is implemented, each joint will point its X-axis to its child. To do this via code, we select the joint then set the `edit mode` and the `channel mode` to true, and we set the `Joint Orientation (OJ)` to 'xyz'.

3.6 Shelf

Since we are adding more lines of code to our class, we need to organize and refactor, so it is easier to detect bugs and fix errors. Our main class is the `AutoRigger()`. It displays the UI, retrieves information from the user, and calls other classes with each button press. We then make a constructor known as `__init__(self):`. Inside this constructor, we call our main method `Build UI`, which then builds the UI to start the auto rigging process. We make other classes such as the `Locators`, `Joints`, `Create IK`, `Set Attributes`, `Controllers` and `Constraints`. Each class contains the method responsible for their main function.

We create a new icon on the custom tab on the shelf editor and insert the following code into it:

```
Import Autorigger as AR

reload(AR)

UI = AR.AutoRigger()
```

The reload function is vital for our code. When a module is reloaded, its dictionary (containing the module's global variables) is retained [2]. Thus, we do not have to run every other class or script every time we want to run our code. Once we click on the icon, the implemented code runs.

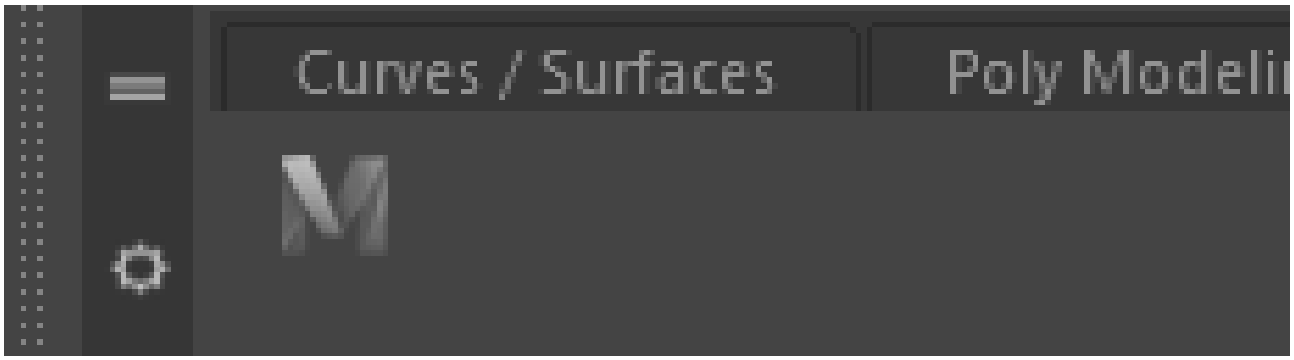

Figure 3.11: Rig

3.7 Inverse Kinematics

Now that the skeleton is done, it is time to start moving it. We begin with implementing it in the arm using the following code:

```
base.ikHandle(name = "IK_L_Arm", sj = base.ls("RIG_L_UpperArm")[0],
ee = base.ls("RIG_L_Wrist")[0], sol = 'ikRPsolver')
```

Where `Starting Joint (SJ)` is the `Upper Arm`, `End Effector (EE)` holds the `Wrist`. There are different types of `Solver (SOL)` required to dictate the motion in each limb.

- `ikRPsolver`:The Rotating Plane method is ideal for joint chains such as arms and legs, where it is vital to remain in the same plane. For example, as the elbow rotates, the shoulder, elbow, and wrist joints of an arm should stay in the same plane.
- `ikSCsolver`: The Single Chain equation handles the orientation of the joints included in the IK chain.
- `ikSplinesolver`:It is used for posing long chains of joints that are present in spines, tails and snakes, for example.

For the spine to work, we need to incorporate a curve, since the `ikSplinesolver` works with a custom curve. Initially, we retrieve the `Root` position and mark it as the start point of our curve. Afterwards, we get the positions of the spine joints and add curve points in their location and append them. Moreover, we set the `degree` to 1; ergo, Maya does not attempt to round off the straight line. The curve is now set; nevertheless, we can not control it. Accordingly, we transform the curve's vertices into control vertices and insert them into a cluster using this piece of code:

```
curveCV = base.ls('SpineCurve.cv[0:]', fl = True)

for k, cv in enumerate(curveCV):

  c = base.cluster(cv, cv, n = "Spine_Cluster_"+str(k)+"_")

  if k > 0:

    base.parent(c, "Spine_Cluster_"+str(k - 1)+"_Handle")
```

Where `Flatten (FL)` retrieves the objects' name only. Furthermore, we parent each cluster to the one before it; consequently, each vertex on the curve affects the others. Ultimately, we should add the IK handle for the spine, and it is executed by this line of code;

```
base.ikHandle(n = "IK_Spine", sj = "RIG_ROOT",
e = "RIG_SPINE_" + str(len(spineAmount) - 1), sol = 'ikSplineSolver',
c = 'SpineCurve', ccv = False)
```

We set `Create Curve (CCV)` to false, since we implemented our own custom one.

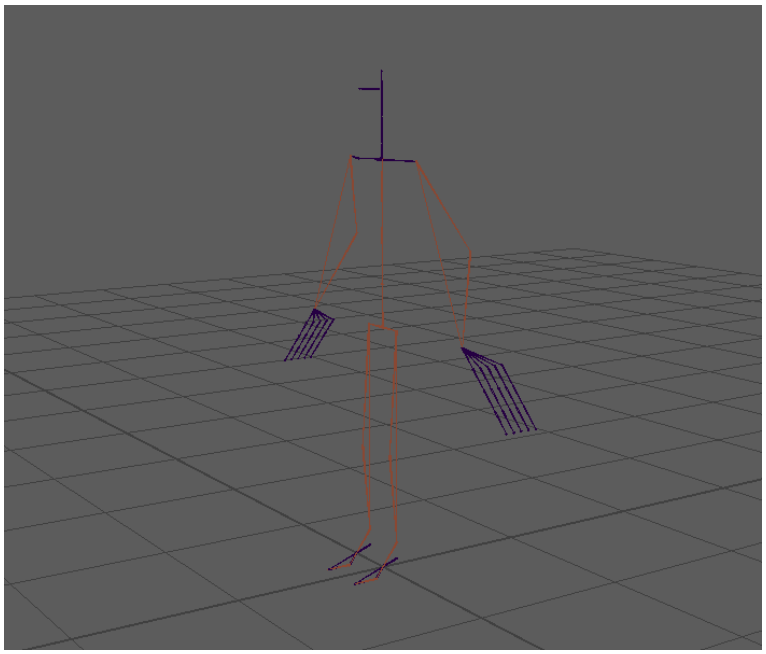

Figure 3.12: IK Rig

3.8 Controllers

We construct the controller class and start designing various shapes to place on our rig. First and foremost, we make the `Master Controller`, which is responsible for moving the skeleton as a whole.

```
master_ctrl = base.circle(nr = (0,1,0), c = (0,0,0), radius = 1,
degree = 1, s = 16, name = "MASTER_CONTROLLER")
```

```
selection = base.select("MASTER_CONTROLLER.cv[1]","MASTER_CONTROLLER.cv[3]",
"MASTER_CONTROLLER.cv[5]","MASTER_CONTROLLER.cv[7]","MASTER_CONTROLLER.cv[9]",
"MASTER_CONTROLLER.cv[11]","MASTER_CONTROLLER.cv[13]", "MASTER_CONTROLLER.cv[15]")
```

```
base.scale(0.7, 0.7, 0.7, selection)
```

```
base.scale(1, 1, 1, master_ctrl)
```

```
base.makeIdentity(master_ctrl, apply = True, t = 1, r = 1, s = 1)
```

The `Sections (S)` in the first method helps us divide the circle by 16 vertices; ergo, we can transform the circle into a star, for cosmetic purposes. This is achieved by scaling down the `selection` of every other vertex. Ultimately, we freeze the attributes of the all the controllers with the makeIdentity built-in method in Maya.

```
base.makeIdentity(master_ctrl, apply = True, t = 1, r = 1, s = 1)
```

Where `t` is transformation, `r is rotation` and `s` stands for scale.

The next step is to create a controller for every start or end effector joint. We create a `Pelvis Controller` and parent it to the `Master Controller`. Each controller from here on out will be parented to the master or its children.

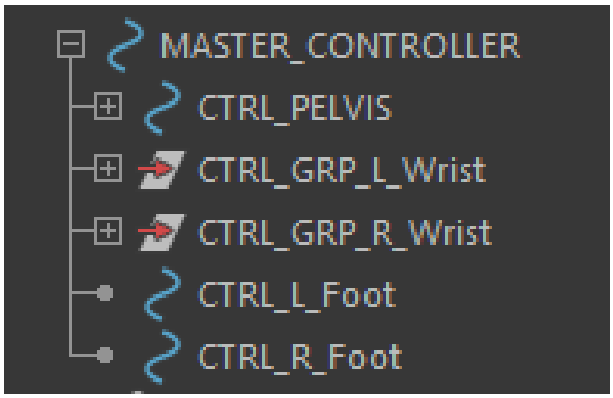

Figure 3.13: Controller Hierarchy in the Outliner

We then create a controller for the head, neck and the clavicles, all with various shapes. The wrist; however, requires more details. We try to adjust the wrist control circle to match the rotation of the wrist joint, so it is easier to locate and move.

```
base.rotate(0,0, -wristRotation[2], wristGrp)
```

Furthermore, we add an `Elbow PV` attribute, which is present on the `Channel Box` on Maya's right side. It is responsible for rotating the elbow, being as we have no controller placed there. It is also more accurate for the user to manipulate the attribute values than moving an elbow around.

Regarding the fingers and spines, we place a controller on top of every joint present within these figures. As for the feet, we place an arrow shaped controller beneath each foot. Furthermore, we add four attributes to our shape:

- `Knee Fix`: Responsible for very minimal knee movement in the Z-axis. This is important to adjust the knee posture. When we stand up, the knee is not fully straightened and has a slight bend to its shape. Hence, the attribute's default is set to 90. With a minium of 0, fully straight, and a maximum of 100, full minimal bending.
- `Knee Twist`: This dictates the knee rotation on the Y-axis.
- `Foot Roll`: It represents foot bending with the foot ball and toes remaining on the ground.
- `Ball Roll`: With only the toes on the ground, this simulates tip toeing as the whole foot stretches as the attribute's value increases.

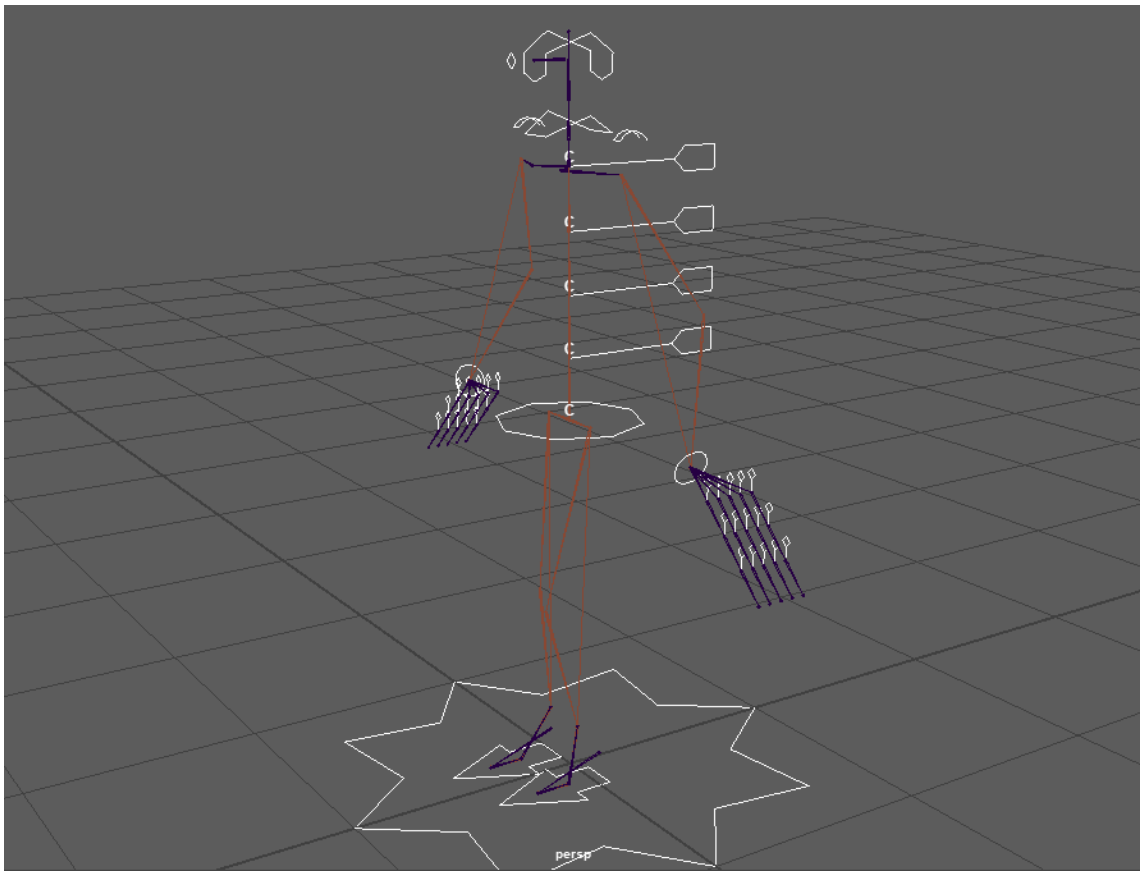

Figure 3.14: Controllers

Finally, we set all the controller colours to white; ergo, simplifying the UI. Moreover, the controllers now are easily distinguishable from the rig.

3.9 Constraints

Constraints are applied to connect the controllers with their respective IKs and joints. For the joints, we use the `orientConstraint` method to handle the rotations. Afterwards, we use the `pointConstraint` method to integrate the IKs, while setting the Maintain Offset (MO) variable to true in both functions. Concerning the custom attributes, such as `ElbowPV`, we connect it to its similar attribute within the IK handle, using this line of code:

```
base.connectAttr("CTRL_L_Wrist.Elbow_PV", "IK_L_Arm.twist")
```

The `orientConstraint` is applied to the joints that only rotate, as the transitional values only integrate using the `pointConstraint` function. For example, we orient the neck, clavicles, head and jaw controllers only as they only need to rotate and do not displace in a human skeleton. Nevertheless, we use both constraints in other controllers and joints that might need both actions.

Regarding the feet and legs, we also connect the custom attributes with their IKs. Moreover, we apply both the orient and point constraints on the foot joints. We also utilize the `pointConstraint` method to connect their IKs

```
base.connectAttr("CTRL_L_Foot.Foot_Roll", "RIG_L_INV_Ball.rotateX")
base.connectAttr("CTRL_L_Foot.Ball_Roll", "RIG_L_INV_Toes.rotateX")
```

Taking advantage of its original attributes, we connect the Y-axis of the spine to the `IK_Spine.twist`. Hence, only the spine and their children can rotate on the Y-axis, but also utilizing the IK to move all the associated joints correctly. Furthermore, we connect the Z and X-axes to the finger controllers, as they are the only axes to represent their movement in real life. The constraints and controllers are all applied when clicking the `Finalize Rig` button in the `Finalize` window.

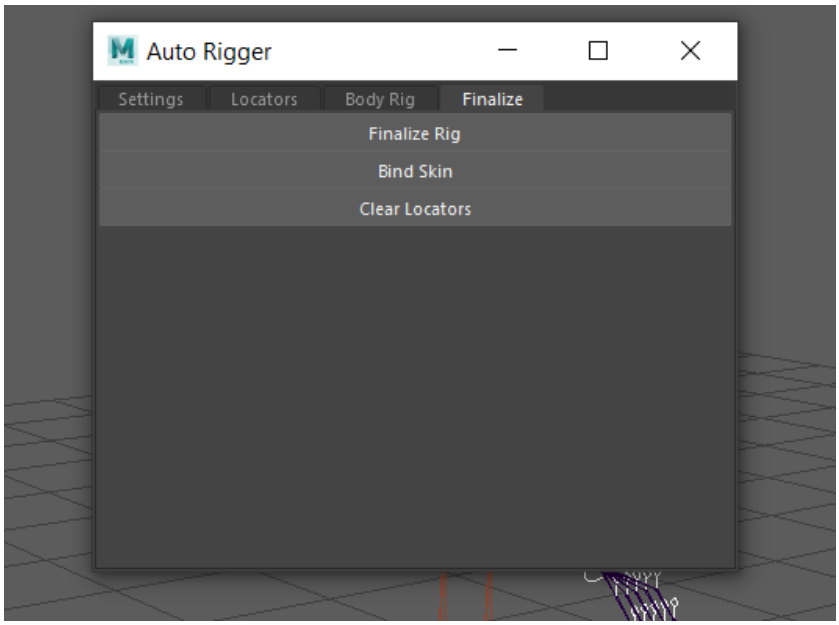

Figure 3.15: Finalize Window

3.10 Set Attributes

The final class we will create is the `Set Attributes` class. Vital for eliminating errors, this class will focus on locking unnecessary attributes for controllers. Moreover, it will also initialize the attributes of another.

First and foremost, we lock the scaling of all the controllers one by one, using this method:

```
base.setAttr('CTRL_PELVIS.scale'+axe, lock = True, k = False)
```

Where axe is the list of the three axes, X, Y and Z, as we loop and lock them all. `K` is the keyable state in the channel box; ergo, we deactivate it, so the user will not be able to manipulate them.

We lock the translation on the spine, clavicles, neck, head, jaw and the finger controllers, since only rotation is sufficed. Additionally, the jaw controller Y and Z axes are locked.

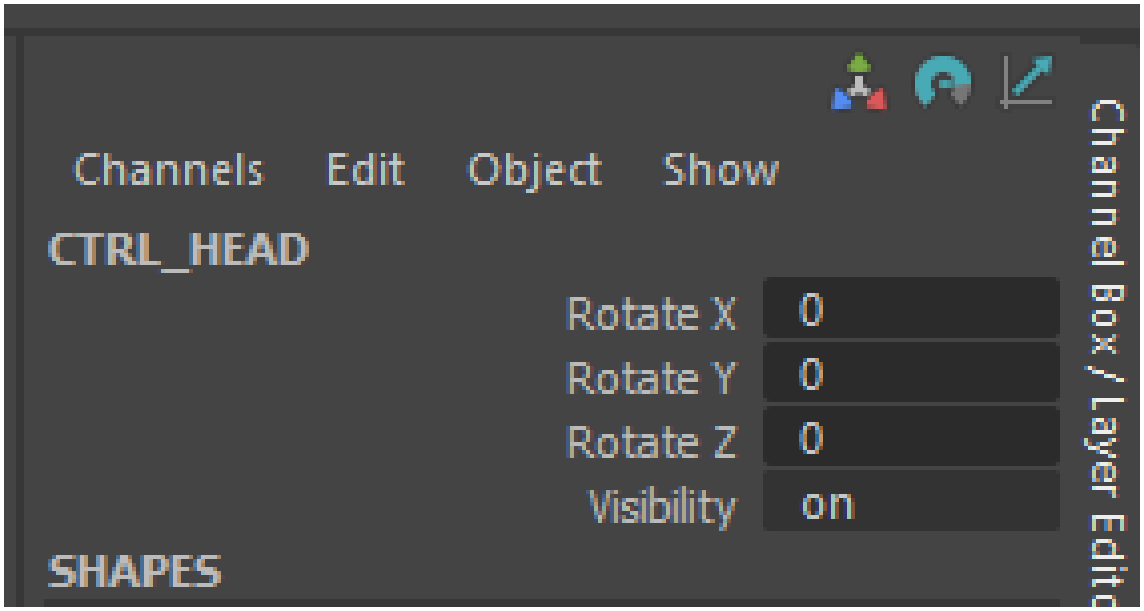

Figure 3.16: Head Attributes in the Channel Box

Figure 3.16 illustrates the absence of all other `CTRL_HEAD` attributes except the rotation.

3.11 Binding the Skin

Ultimately, we need to combine our rig and the user's skin. The user has to select a mesh and click the `Bind Skin` button in the `Finalize` window, which calls a `BindSkin` method. The function then begins by checking if the user selected a mesh. Otherwise, a dialogue prompts up requesting a skin selection. If the mesh was selected, we call the `skinCluster` function. There are multiple ways to choose when binding the skin; ergo, we have chosen the Geodesic Voxel method. The binding algorithm is designed to work with meshes that may contain non-manifold, non-watertight, intersecting triangles or are comprised of multiple connected components, which is uncommon for a humanoid mesh [1]. Moreover, we need to add a `geoBind` function to the skin cluster previously created. Therefore, we loop around every object in the mesh and perform the following functions.

```
base.skinCluster(sel[i], "RIG_ROOT", bm = 3, sm = 1, dr = 0.1,
name = "Mesh"+str(i))
```

```
base.geomBind('Mesh'+str(i), bm = 3, gvp = [256, 1])
```

Where `Binding Method (BM)` value is set to 3, representing the algorithm we have selected. We have opted to linear skinning in the `Skinning Method (SM)` variable. Regarding the `Drop-off Radius (DR)`, we have chosen the lowest value since it is not desirable for the mesh to expand far on the joint. Concerning the resolution, we have chosen 256. As of now, the skin is successfully bound to the rig.

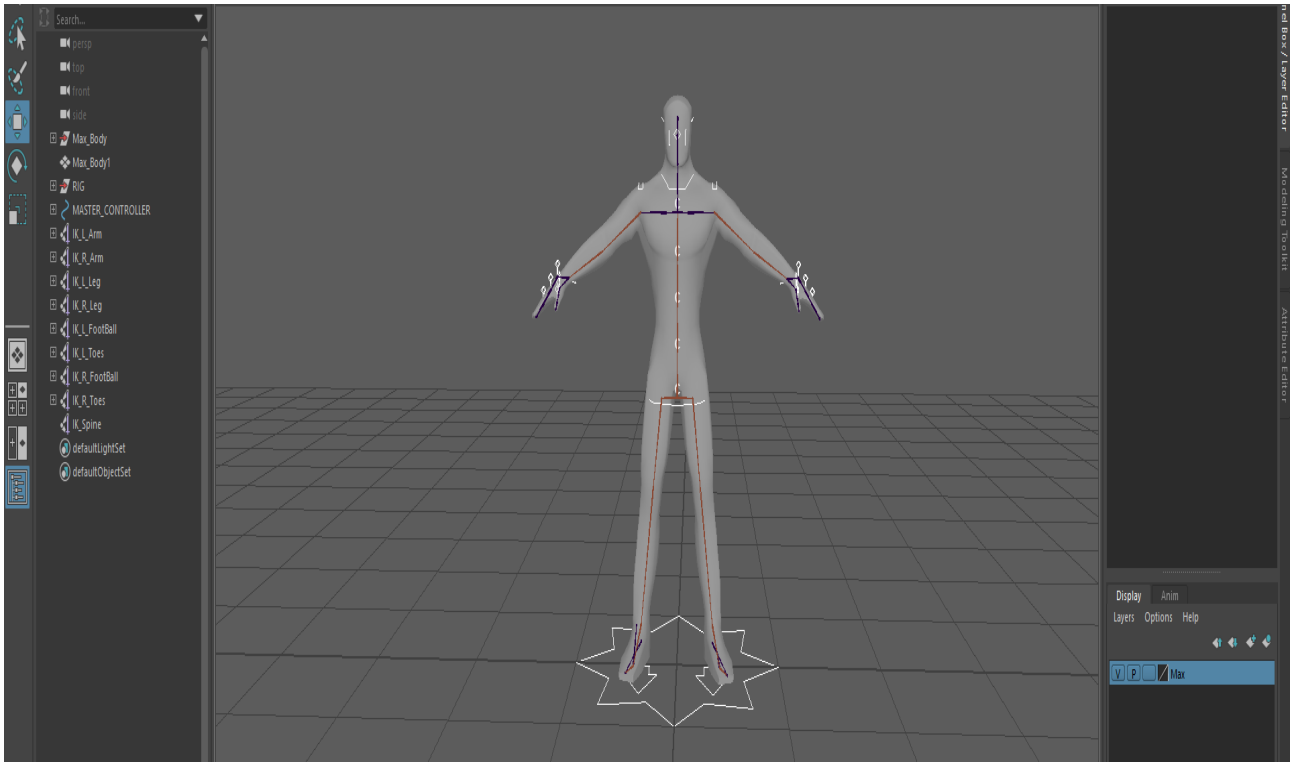

Figure 3.17: Rigged Character

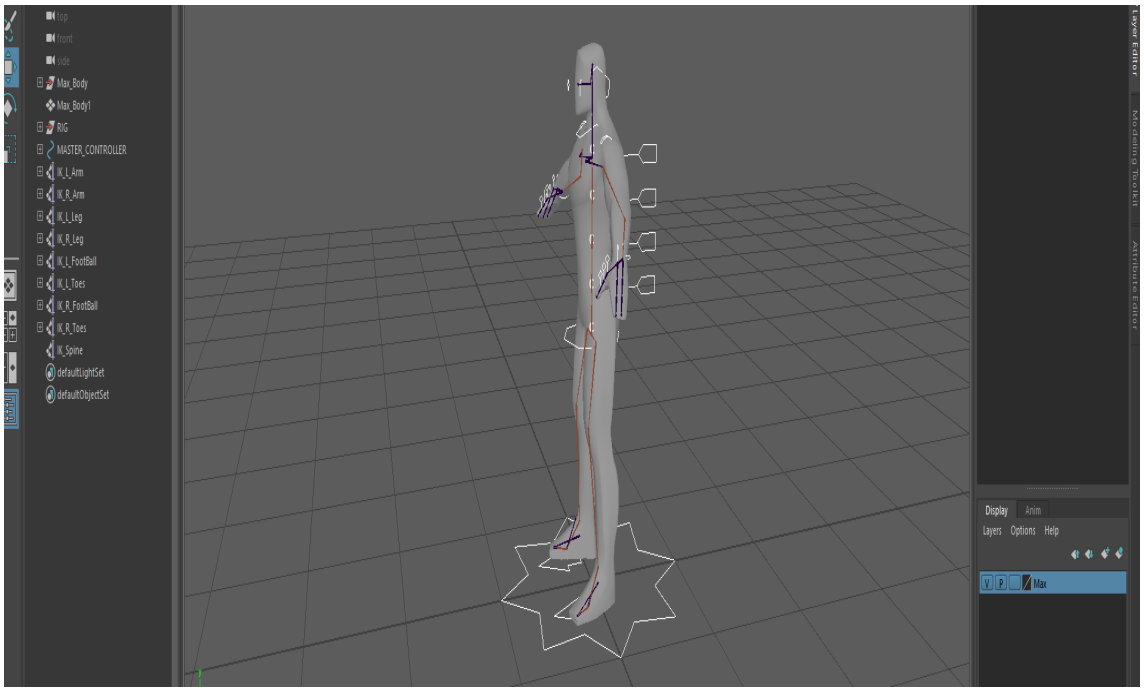

Figure 3.18: Rigged Character Position Two

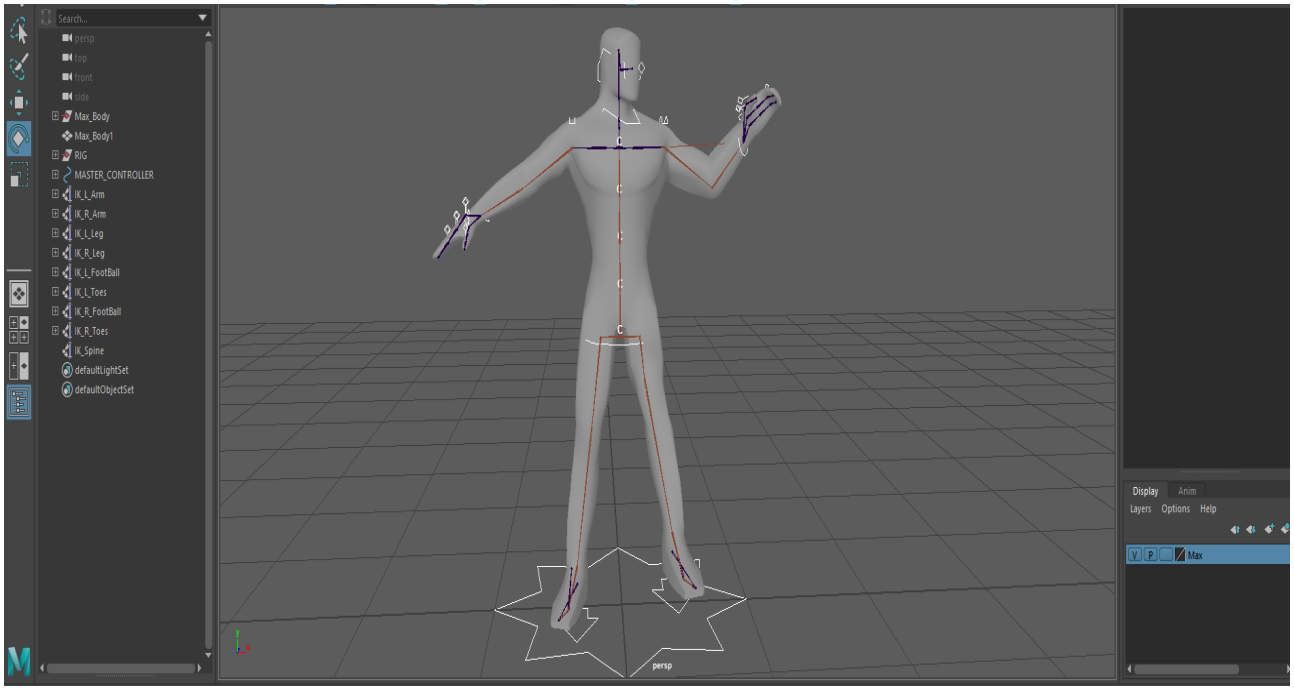

Figure 3.19: Rigged Character Position Three

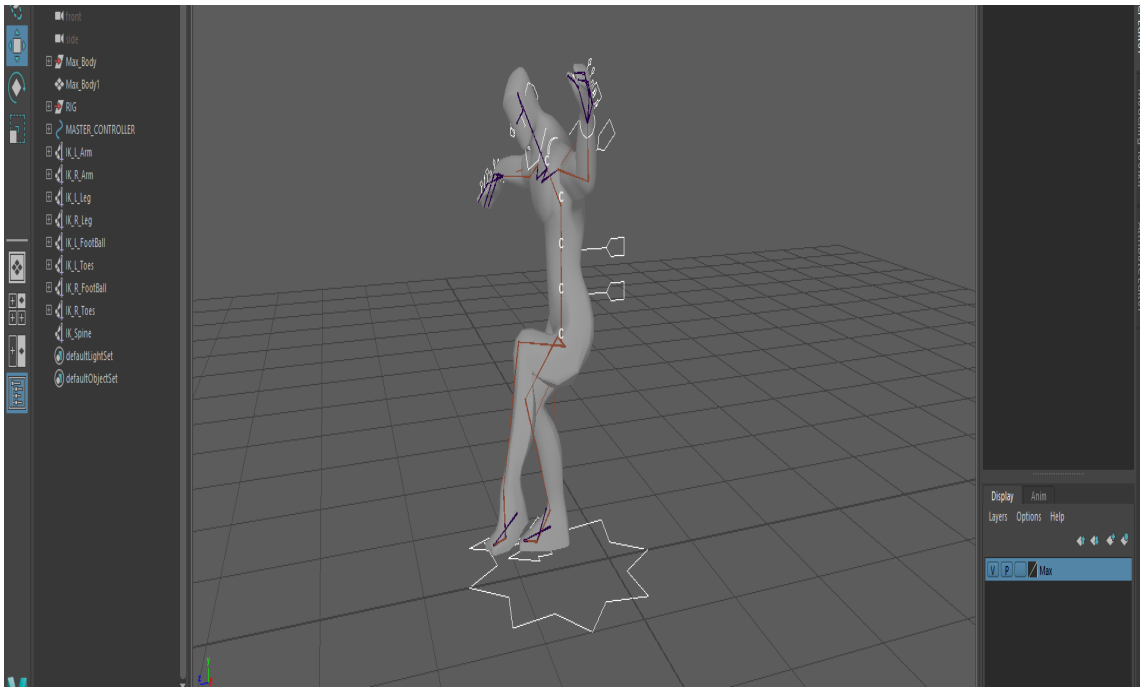

Figure 3.20: Rigged Character Position Four

Chapter 4

Conclusion and Future Work

4.1 Conclusion

This paper explores the possibility of converging character rigging to an automated process. Maximizing efficiency, and decreasing the amount of time spent rigging, were of utmost importance during figuring this out. This would be implemented using AutoDesk Maya 2019 and scripting using Python 2.7 syntax.

Primarily, the visual effects software applications were briefly discussed and the cause for picking our coding language. Insight was given as to how rigging artists rig their character and the tools they use. The human skeleton anatomy was mentioned in order to associate digital work and real life.

Last and foremost, a step by step implementation and overview of the rigging process was discussed. Starting from importing the Python commands to our library, to assembling a fully working automatic rig, bound to the character.

4.2 Future Work

During the courser of this research, several problems arose, due to the fact that Maya is not fully compatible with Python. There are some methods and internal files written in Python code that will not integrate with the visual software. Hopefully, in the near future, all the bugs would be completely gone.

Since an automatic rigger was used, it is more unlikely that a user incorporates their own design to the rig. Some artists prefer to add extra joints to the arm; ergo, creating a forearm twist, which in turn simulate sleeves flapping or fat jiggling. Moreover, the more the joints are, the faster the kinematics calculations.

Not all humanoid characters are the same, some are bigger in volume than others. Therefore, some meshes are too big in volume and can cause some glitches. This can be

solved by adding extra joints in the chest area, mimicking a large ribcage. Nonetheless, this would affect smaller characters, increasing their stiffness.

Ideally, the system should have been tested by many users. This is to pinpoint missing features and needs to be added. However, due to the time and resource limitation on this research, not much more could be done.

Appendix

Appendix A

Lists

UI	User Interface
Maya	AutoDesk Maya [5]
MEL	Maya Embedded Language
IK	Inverse Kinematics
OJ	Joint Orientation
MO	Maintain Offset
SJ	Starting Joint
EE	End Effector
SOL	Solver
FL	Flatten
CCV	Create Curve
S	Sections
BM	Binding Method
SM	Skinning Method
DR	Drop-off Radius

List of Figures

Bibliography

[1] Geodesic voxel binding.

[2] importlib — the implementation of import.

[3] Introduction to rigging in maya - part 7 - rigging the feet.

[4] Inverse kinematics (ik) algorithm design with matlab and simulink.

[5] Maya: Create expansive worlds, complex characters, and dazzling effects.

[6] Holly Landis. Here's how: Learn character rigging.