An Ephemeral Approach to 3D Character Rigging

A Thesis

Presented to the Faculty of

the Department of Computer Science and Information Technology

Kutztown University of Pennsylvania

Kutztown, Pennsylvania

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Patrick E Stelmach

May 2022

Abstract

All 3D animated characters share one thing in common: a rig. The rig is what controls a character's movements. Nearly all rigs are manipulated through two basic methods: forward and inverse kinematics. The two kinematics need to blend together to achieve the full range of behavior desired by the animator. For the past 20 years, this blend has been achieved by swapping the character's rig between two separate kinematic rigs. This blending is clunky, unintuitive, and prone to mismatches and instability.

This thesis focuses on a new approach to rigging with a design called ephemeral rigging. Instead of two separate rigs preset in either forward or inverse kinematics and constantly swapped between, each joint in the character's rig is treated as a node with possible incoming and outgoing connections. When the animator interacts with a joint, the connections are created at that moment in the specified kinematics mode chosen by the animator. This allows one rig to act as both forward and inverse, reducing complexity and increasing ease of use.

Acknowledgements

I wish to express my appreciation to all of those who have made this thesis possible.

My sincere thanks go to Dr. Spiegel for the direct and intense encouragement to push myself and learn, understand, and appreciate far more than I could have hoped for.

Also, I would like to thank my advisor, Prof. Pham, for constantly challenging me and giving me motivation to overcome obstacles.

In addition, I would like to thank Dr. Fry for her encouragement to pursue a Master's degree and Dr. Carelli for taking the time to remediate my thesis paper.

To my family, I would like to thank you all for your support and encouragement throughout the past ten years of me finding my path.

To my wonderful wife, I love you for the support you've given me and I thank you for your patience through the many nights and weekends I have spent tied to my computer.

Table of Contents

Abstract 2
Acknowledgements 3
List of Figures 6
Glossary 7
1. Introduction 11
2. Background 14
  2.1 - Overall Rigging Concepts 14
  2.2 - Forward Kinematics 16
  2.3 - Inverse Kinematics 17
  2.4 - Benefits and Drawbacks of Both Kinematic Systems 18
3. Ephemeral Rig Concepts 21
4. Tools and Technologies 25
  4.1 - Autodesk Maya 25
  4.2 - Visual Studio 26
  4.3 – C++, Python or MEL 27
  4.4 - Maya API Objects 28
  4.5 - Dirty Nodes 30
5. Design and Implementation 31
  5.1 – Design Overview 31
  5.2 - Scene at Rest 35
  5.3 - Four Interaction Modes 35
  5.4 - Walk Nodes 37
  5.5 - Update Locators 40
  5.6 - Forward Mode Details 41
  5.7 - Pseudo Inverse Mode Details 44
  5.8 - Inverse Mode Details 47
  5.9 – Suspended Mode Details 50
  5.10 - Ephemeral Mode Destruction 51
6. Conclusion and Future Work 53
  6.1 – Summary 53
  6.2 – Shortcomings and Future Work 53

6.3 – Conclusion 56
Bibliography 57
Appendices 59
Appendix A - Design Diagrams 59
Appendix B - Illustrative Code Snippets 64

List of Figures

Figure 1: Components of a 3D Character 11
Figure 2: Control Vertex Animation in Terminator 2 12
Figure 3: Skeleton Inside 3D Model 15
Figure 4: Joint Influence Map on Mesh 16
Figure 5: Using Forward Kinematics to Grab a Cup 17
Figure 6: Using Inverse Kinematics to Grab a Cup 18
Figure 7: Switching from FK to IK 20
Figure 8: Ephemeral Connection Overview 22
Figure 9: Illustrating the DG and DAG Nodes in Maya 26
Figure 10: Generic Maya Node Structure 29
Figure 11: Class Diagram of Ephemeral System 32
Figure 12: Ephemeral System Basics in Scene 33
Figure 13: Ephemeral System Connections in Connection Graph 33
Figure 14: Overview of the Setup for the Four Interaction Modes 36
Figure 15: Walk Nodes Example. 38
Figure 16: Walk Nodes Flowchart 39
Figure 17: Update Locators Flowchart 40
Figure 18: Forward Mode Example 42
Figure 19: Forward Mode Creation 43
Figure 20: Pseudo-Inverse Example 44
Figure 21: Pseudo Inverse Creation 45
Figure 22: Inverse Mode Example 47
Figure 23: Inverse System Creation 49
Figure 24: Ephemeral Graph Destruction 51

Figure A1: Walk Nodes Details 59
Figure A2: Update Locators Details 60
Figure A3: Forward Mode Details 61
Figure A4: Pseudo Inverse Details 62
Figure A5: Inverse Mode Details 63

Figure B1: Walk Nodes Code 64
Figure B2: Ephemeral Node Compute 65
Figure B3: Master Node System Constructor 68
Figure B4: Master Node Graph Destructor 70

Glossary

- **Application Programming Interface (API)** – a pre-built connection that acts as an intermediary between two programs. In this case, the connection is between Maya and Visual Studio.
- **Callbacks –** a function in Maya that upon hearing a requested event calls a separate pre-defined function. This function can be passed data related to the requested event.
- **Child –** An object placed below another (a parent) in a hierarchy. The child follows and mimics the parent's movements.
- **Directed Acyclic Graph (DAG) Nodes** – A Dependency Graph (DG) node that also exists in 3D space.
- **Dependency Graph (DG) Nodes –** Maya has a wide variety of nodes that all serve different functions. All of the nodes in Maya are Dependency Graph nodes.
- **Dirty** – a state of an MPlug. If the MPlug' state is dirty, Maya will recalculate the value of this plug.
- **Dynamic typing** – When the majority of data type checking is performed at run time.
- **Edge** – The point where any two polygon faces meet.
- **Ephemeral** – An item that is transitory and exists very briefly.
- **Ephemeral rig** – A rig that dynamically constructs and destroys the connections between nodes based on user settings.
- **Face** – The area between vertices and edges on a 3D model. Faces make up the faceted visible form of the model, seen in Figure 1.
- **Forward Kinematics (FK)** – A rigging system in which the rotation propagates forwards down a chain of joints.

- **Integrated Development Environment (IDE)** – Software environment and supporting tools for building software applications. Combines common developer's tools into a single interface. In this paper, the IDE used is Visual Studio 2019.
- **Interpreter** – Translates written code into machine code.
- **Inverse Kinematics (IK)** – A rigging system where the animation propagates up the joint tree in a reverse direction. The child joint is moved in the scene, and all its parents are placed to maintain their relationship to that child.
- **IK Handle** – An object inside Maya that calculates IK transformations on a joint chain.
- **Keyframe** – A marker that specifies an object's position and attributes at a given point in time inside Maya.
- **Local Space** – the position of an object in respect to its parent objects.
- **Locator** – A DAG Node object in Maya that only has transformation values.
- **Matrix** – A single object that contains all nine transformation channels. These are translation, rotation, and scale in the X, Y, and Z axis. The matrix can be read and modified either through the matrix MPlug, or can be decomposed and used with all nine transformation MPlugs.
- **Maya** – The 3D software package that contains the ephemeral system.
- **MFnDependancyNode** – A dependency node function set which allows the creation and manipulation of DG nodes. This is used to access attributes inside an MObject.
- **MObject** – A generic class for internal Maya objects. These MObjects act as containers for data. An MObject can contain simple data, such as an integer, or complex data, such as other MObjects. MObjects are all Directed Graph (DG) nodes.

- **MPlug** – A connection point into an object's attributes. All attributes have a related plug, which can fetch and set the internal data.
- **MPxNode** – The base class of all user defined nodes in Maya. An MPxNode is also referred to as a Directed Graph (DG) node, which was discussed earlier. Included in this object is a constructor, destructor, and a compute function, which controls the behavior of the node.
- **MString** – A Maya string. Required for use in Maya functions instead of std::string. It is a type of character string and can be treated as such.
- **Node** – A single data structure. A node contains data and is usually connected to other nodes.
- **Object Oriented Design** – A system of interacting objects used to solve a software problem.
- **Parent** – An object above another (a child) in a hierarchy.
- **Parent Constraint** – A constraint controlled by Maya between a child and parent object. The child object follows all the parent object's translations and rotations.
- **Plugin** – A program that can be added to Maya to add new functions or improve existing behavior.
- **Recursive** – When a program calls itself one or more times until a specific condition is met to exit the program.
- **Rig –** A series of connected digital bones that are used to animate a 3d character model.
- **Rotate Pivot** – the pivot of an object in the scene. Is usually read and set as world space instead of local space.
- **Static typing** – When the majority of data type checking is done at compile time.

- **Threading** – a set of instructions designed and run independently of the parent process.
- **Vertex** – A point of intersection between three or more edges in a 3D model.
- **World Space** – the position of an object in the scene, regardless of parent transformation values.

1. Introduction

Rigging is not a new concept. The term rigging originally referred to the ropes and chains used to move a ship's masts and sails. Using ropes to give an object movement was adapted into rigging puppets with string so they could dance around a stage. As animation made its way off the stage and into film, new rigging systems were needed, as a marionette's hanging strings were difficult to hide and too imprecise. To resolve this, wire skeletons that mirrored standard human anatomy were placed inside miniature posable figures. The animators would pose these skeletons into precise stances to create and film stop motion animation.

The transition to the digital 3D space brought about a new set of challenges. Digital characters are made of polygons. These polygons have several features, such as faces, edges, and vertices. To create motion, the vertex positions of these polygons need to be updated over time. Several different methods were designed to rig and animate these vertices as the understanding of computers and their capabilities evolved.

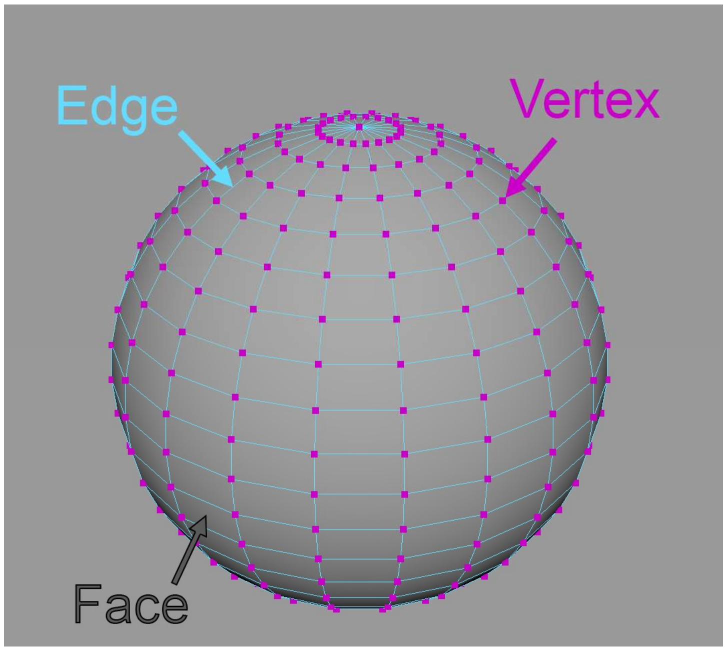


Figure 1: Components of a 3D Character

When the first digital characters were created, there was not enough computing power for an internal skeleton to control a model. A brute force approach to control the vertices was developed. The original character models were animated with control vertex animation, which is "the keyframed animation of the individual vertices of a mesh" (Vertex Animation, 2021). This forced the artists to work as mathematicians by calculating the position change for each vertex on the 3D model and manually updating each vertex for every frame. In addition to the tedium of manipulating every vertex, there was growing demand for fidelity. To increase the realism, more polygons and vertices were added to models, which increased the workload for control vertex animation. "While preparing explicit data for each key-frame was satisfactory in 1972, today's demand for realism is much higher, resulting in the number of vertices growing by several orders of magnitude. Considering such models, manufacturing each key-frame is no longer feasible" (Radovan & Pretorius, 2006). Control vertex animation was used in films such as Terminator 2, but was incredibly time-consuming and prone to failure.

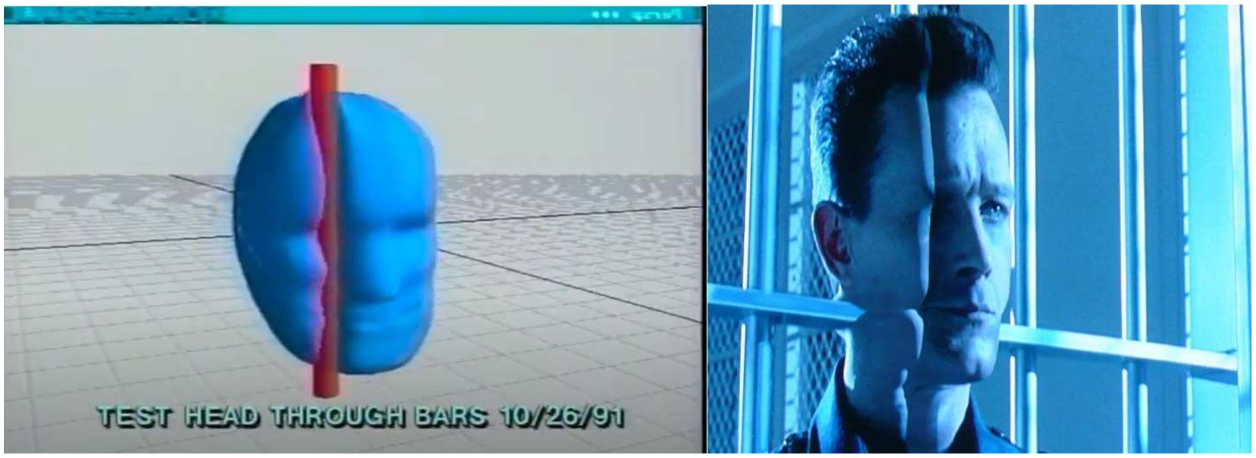


Figure 2: Control Vertex Animation in Terminator 2 (Industrial Light and Magic, 1993)

Once computer power increased, armatures similar to those used in stop motion became the most common method used to animate virtual characters due to the versatility of the system and the ease of understanding for the animators. These rigs use internal joints that are bound to the mesh and drive the vertex movements. The mathematical methodology used in driving these joints breaks down into two different processes: forward kinematics and inverse kinematics. Forward kinematics consists of rotating each joint down a limb to reach the desired pose. Inverse kinematics starts with the end of the limb placed in its final position and the leading joints are then posed automatically. Nearly all modern rigs use a combination of these two processes, which will be explained later in further detail. These processes work in opposition to each other, and need multiple rigs and animator intervention to operate on the same character. This thesis postulates a system that merges the functionality of both processes by creating connections dynamically. In this system, there is a central node that stores data and global functions, and each joint of the rig has an associated ephemeral node which stores data and procedures for that joint.

The 3D environment used for development of this tool is Autodesk Maya, which is an industry standard. The code for the plugin was written in C++ and developed inside Visual Studio. These software packages will be explained in greater detail, along with the reasoning behind their selection. The focus of this thesis is on a humanoid rig, but this system could be adapted to other non-humanoid characters as well.

2. Background

2.1 - Overall Rigging Concepts

This thesis focuses on modifying a traditional humanoid rig into an ephemeral system. To understand the importance of this change, one must also understand the universal rigging standard used today.

A modern character rig uses an armature inside the character. An armature is made of bones and joints. These terms are used interchangeably and this thesis will refer to them as joints. Some examples are upper and lower leg joints, multiple spine joints, along with joints for each segment of the hand and fingers. Facial rigs will have some joints as well, primarily for jaw and neck movement. There are additional rigging methods used for facial animation, which are outside the scope of this system. Joints are placed in a similar location as the anatomical equivalents are in humans. The joints are placed into a hierarchal system of parents and children. An example relationship is the shoulder joint, which is the parent of the elbow joint, which, in turn, is the parent of the wrist joint.

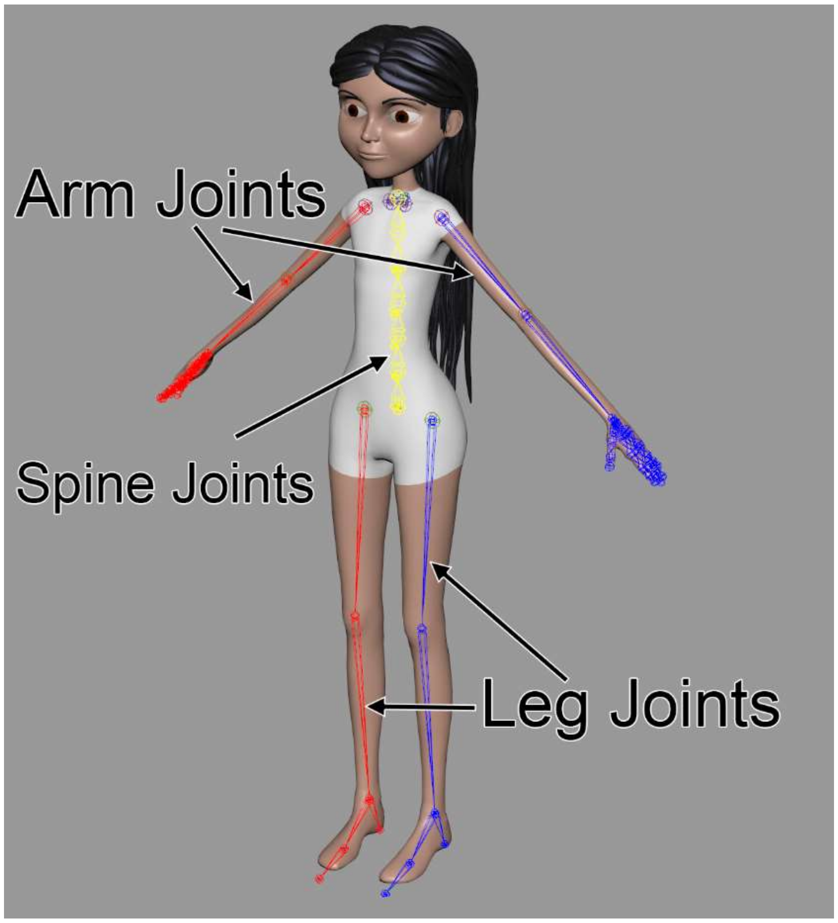


Figure 3: Skeleton Inside 3D Model

Every joint in the scene has a set influence over each vertex of the 3D character. The influence controls how much that joint moves the model. This influence, which is illustrated in Figure 4, informs the system how much to move each vertex when a specific joint is moved or rotated. The colors in the figure represent how much the shoulder joint is affecting each vertex. The closer the color is to white on the gradient, the more influence the joint has on that vertex. This places the computational load on the computer, as the animator only needs to focus on the joint movement and not the model's deformation.

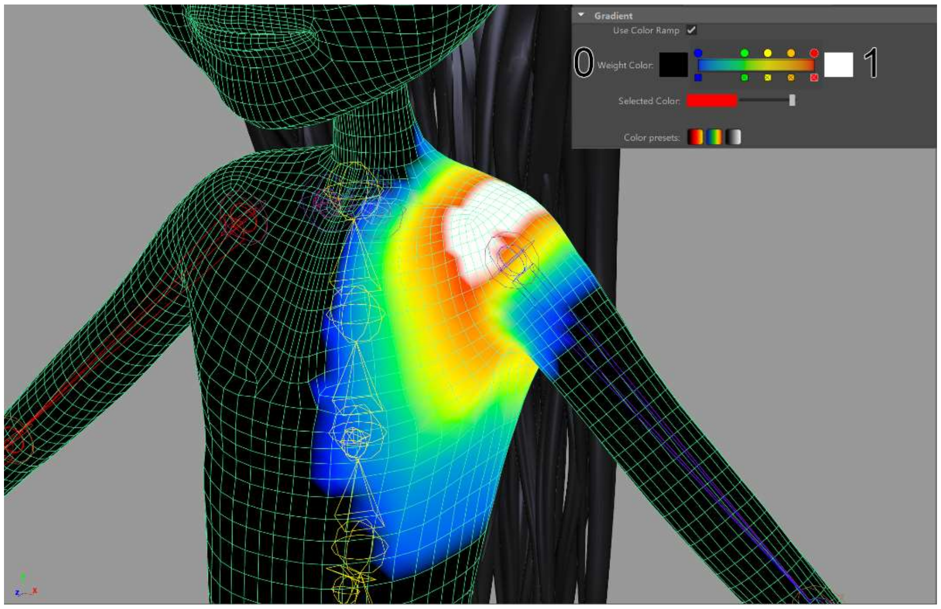


Figure 4: Joint Influence Map on Mesh

Once the joints are connected to the character, additional mechanisms are added on top of the joints to propagate animation through the system. The two most commonly used systems are Forward Kinematics (FK) and Inverse Kinematics (IK).

2.2 - Forward Kinematics

Forward Kinematics is a system in which the rotation of a parent joint propagates forward down a chain of joints. If the shoulder joint is rotated, all the children of this joint, such as the elbow and wrist, will follow that exact rotation from the shoulder's pivot point. "In an FK system the animator must specify all the parameters for degree of rotation and their order for each joint in the hierarchy, to move the limb from point A to point B in 3D workspace" (Bhatti, Shah, Shahidi, & Karbasi, 2013). FK requires the animator to move the parent joint, then work their way down the chain to the lowest child. This system can be tedious to use as each joint needs to be manually set, but is easy to understand and maintains volume well. FK is primarily used when a limb needs to make an arcing motion, such as throwing a ball or swinging an arm while walking. Also, fingers are almost always FK as they curl uniformly and unidirectionally, which leverages FK's strengths.

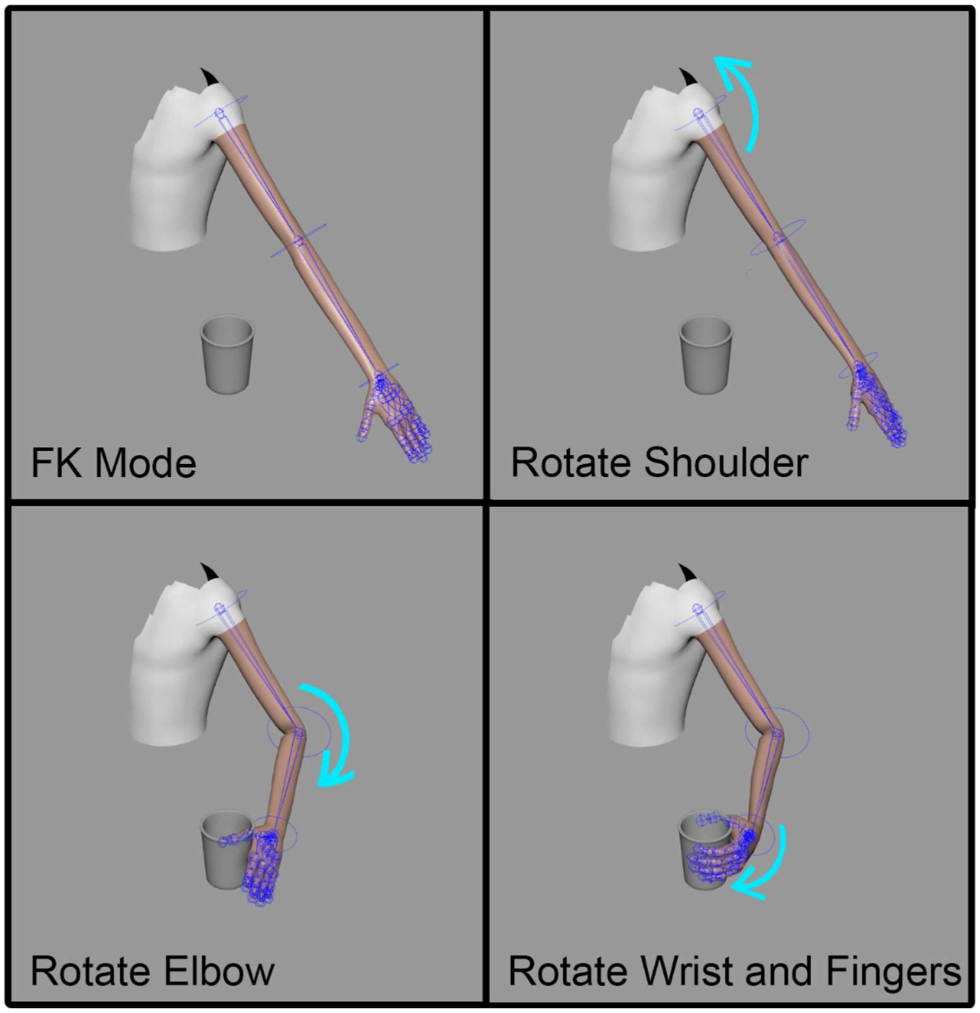


Figure 5: Using Forward Kinematics to Grab a Cup

2.3 - Inverse Kinematics

Inverse Kinematics is a system where the animation propagates up the joint tree in a reverse direction, with the child as the lead. The transformation and rotation of the parent joints are calculated by the system to maintain the relative distance between all the affected joints. "With IK, move the last child in the hierarchy and all its parent joints will rotate, in Inverse motion or kinematics" (Bhatti et al., 2013). IK chains are usually limited to three joints because the computation becomes less predictable with several middle joints. IK systems are primarily used in the arms and legs, while FK is used universally. IK is best for end-focused action, such as picking up an object or planting a foot on the ground while walking.

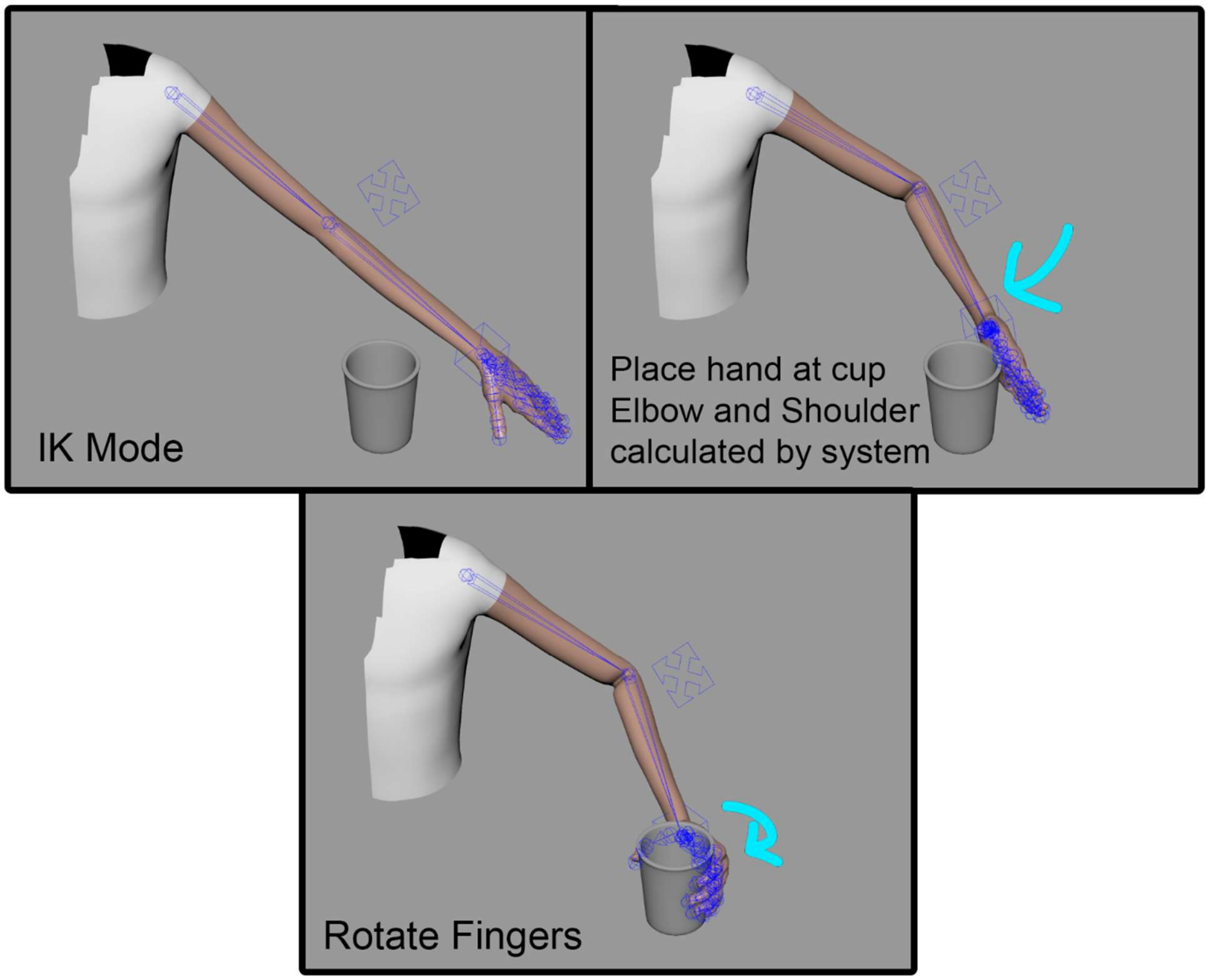


Figure 6: Using Inverse Kinematics to Grab a Cup

2.4 - Benefits and Drawbacks of Both Kinematic Systems

Both FK and IK have fundamental uses in animation, so nearly all modern rigs implement both systems onto every character. The primary issue is that Forward Kinematics is the functional opposite of Inverse Kinematics. The joint relationships cannot function in a system where the parent is driving the child while the child is driving the parent. A common solution to this is having three individual joint chains created for each character. One is specifically used for FK, one is used for IK, and one is bound to the 3D character. The animator can then decide as needed which of the two kinematic joint chains are driving the bound joint chain.

Issues arise when the animator tries to switch between the two systems during the scene. Since only one system can be active at a time, the FK and IK systems are animated separately. When switching the bound joint chain from one mode to another, the placement of the bound chain will update to the other's location, overriding the previous animation. "Riggers are therefore required to invert the rig and find the controller parameters of the target space that match the character pose in the original space. A set of switches is usually provided, such as the legs/arms IK/FK switch, many of which are complicated to write" (Corvazier & Robert, 2021). Consider a character going to shake another character's hand. The animator uses FK to swing the arm in an arcing motion to the point of contact, then switches to IK as now the hand should drive the action. However, the IK system was disabled and when the animator activates it, the bound rig will change location to match where the IK system is. This overrides the FK system's animation, placing the arm in an unexpected location. Figure 7 illustrates another scenario where a character is holding a cup. The arm was placed using FK mode. However, the animator now wants to use IK for the next part of the animation. Since the IK system wasn't active, the IK joints are still sitting in their starting locations. Switching to IK causes the arm to change position, which is not the desired outcome.

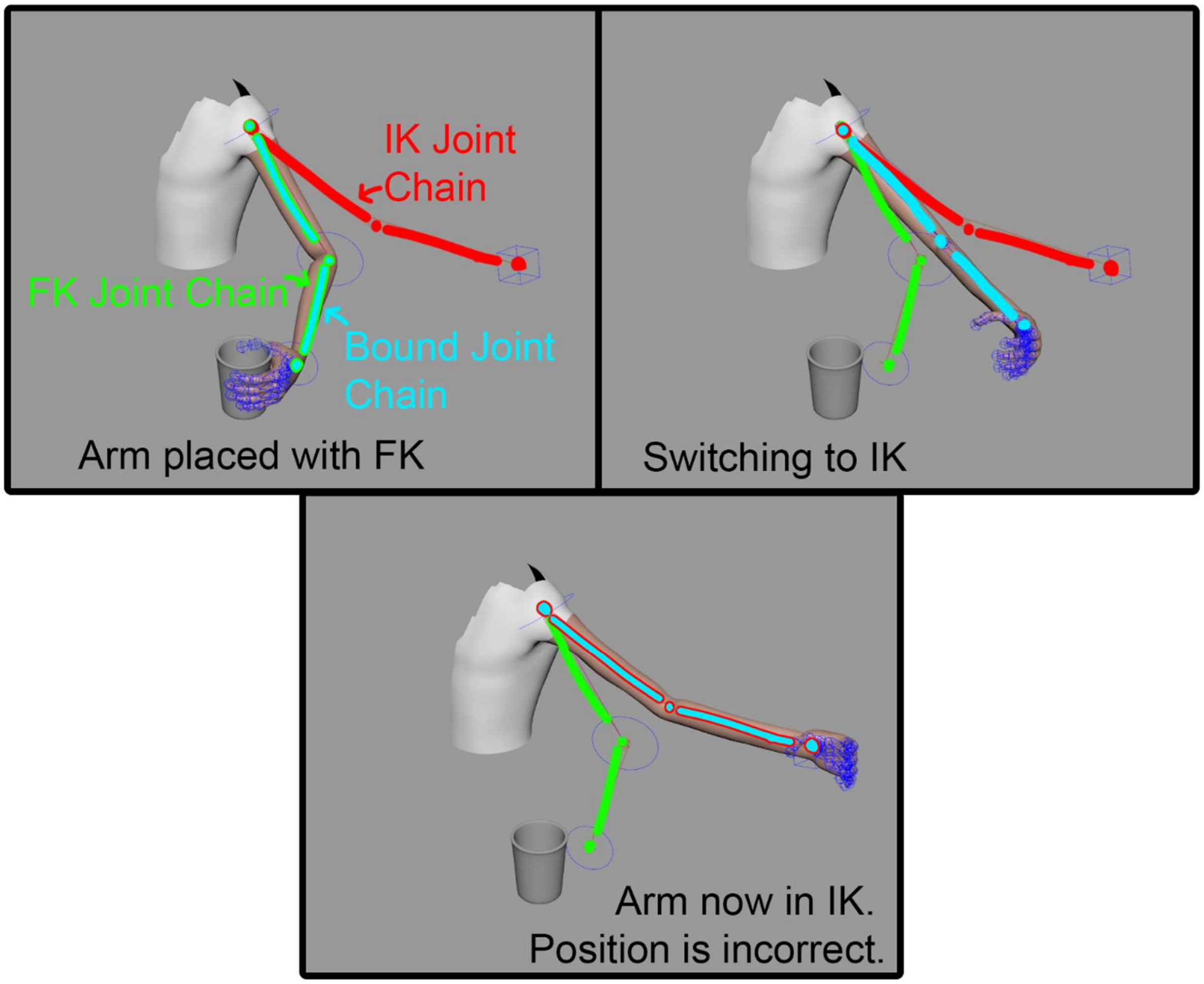


Figure 7: Switching from FK to IK

Tools do exist to mitigate this disconnect, such as the Maya plugin "Universal IK FK Switch and Match Tool" (Elbmann, 2017). This tool implements a system that updates the position of the IK rig to match that of the FK rig on switch and vice versa. These tools need to be configured on a case-by-case basis for each rig, add another layer of complexity on top of the rig, and are opaque to the animator.

3. Ephemeral Rig Concepts

The current solutions of blending between FK and IK are based around three joint chains, with the bound chain swapping between the FK and IK chains. This thesis postulates a new system that uses nodes instead of joint chains to control the relationship between FK and IK modes. These nodes are created and stored outside the bound joint hierarchy. The relationships between the nodes and the joints are created when needed, allowing the system to dynamically, or 'ephemerally', switch between different relationship modes.

The ephemeral nodes exist outside the joint chain and influence each joint directly. The relationships between the nodes are static and predefined, but when the animator switches from one mode to another, the connections between the joints are destroyed and reconfigured using the ephemeral system's static connections. This allows the animator to swap between modes at any time as the joints are no longer affected by the old system and the new system is created in place, solving any placement issues.

As illustrated in Figure 8, the forward and inverse connections between the ephemeral nodes are predefined. When the ephemeral nodes are in a neutral state, there are no connections between the joints. Once the ephemeral nodes enter forward mode, the joints are connected following a parent to child relationship. If the ephemeral nodes are set to inverse mode, the connections between the joints are from child to parent. During any change of mode, the connections between the joints are destroyed to make way for new connections.

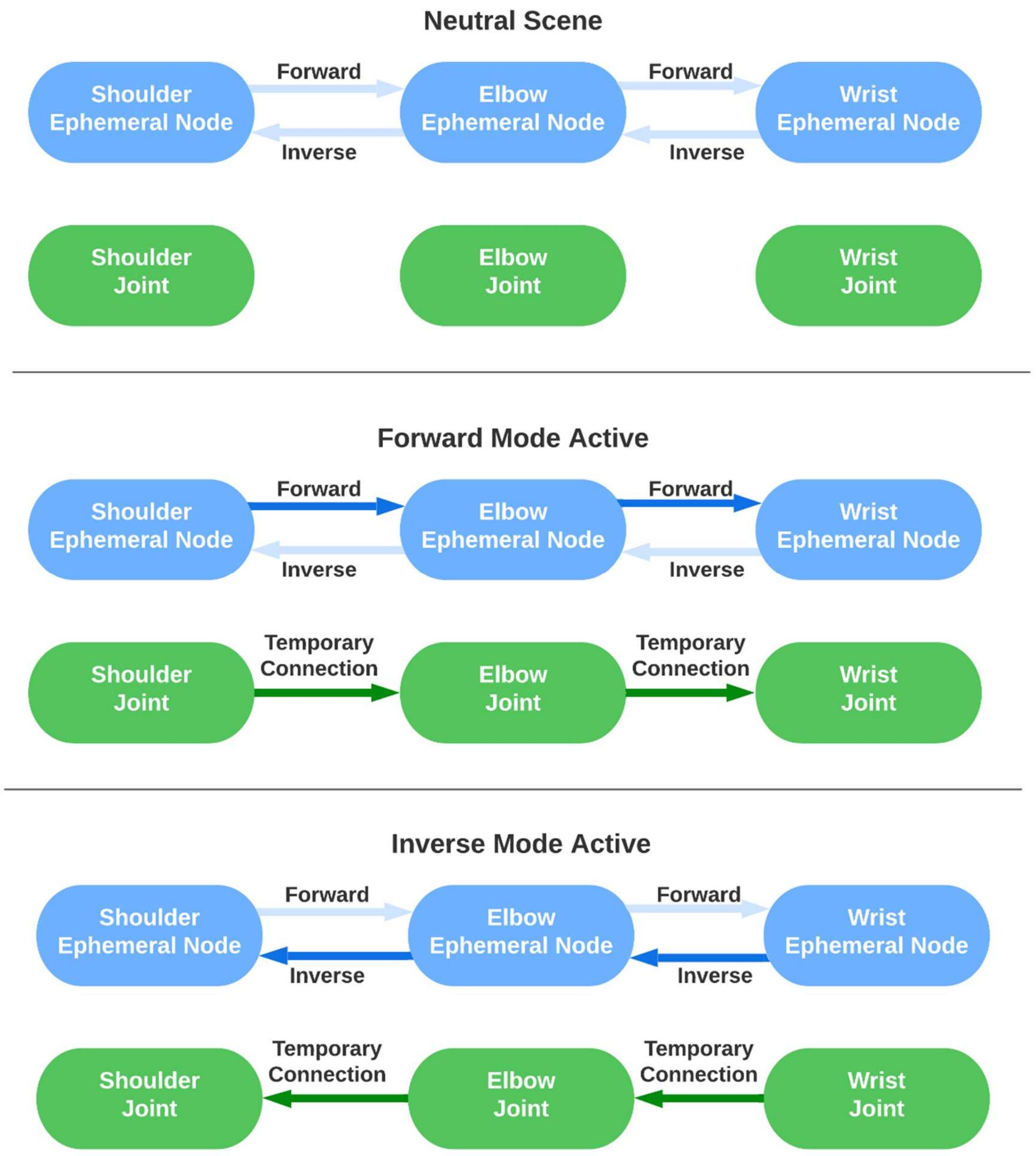


Figure 8: Ephemeral Connection Overview

There are multiple benefits of the ephemeral rigging system. Since the rig is created in place on demand, there is no need for the rigger to create separate FK and IK joint chains. Each rig is created on a small scale behind the scenes when the animator requests it, and the rig automatically controls the bound joints. The ephemeral rigging system also removes the need for blend systems and position updates, as there is no need to blend between static elements.

The ephemeral rig is also easier for an animator to understand. Since the need to blend between the systems is removed, the animator can simply choose the mode they want at that moment and drive the interaction. They do not need to compensate for any lag between FK and IK communication or learn technical relationships to interact with the rig.

Since the ephemeral rig is node-based and outside the hierarchy, it gains the benefit of encapsulating its own functionality. During production, modifications might need to be made to a rig. Because the code is contained within the ephemeral rig, a developer can modify the plugin's code, add new functionality, or repair unexpected behavior. These changes can then be placed into an active scene without destroying any existing work.

Instead of basing the rig structure in a traditional hierarchy, the ephemeral rig is node based. Using nodes rather than hierarchies to construct the rig mirrors the workflow that developers and artists have in other production suites. Connections between nodes allow easy mid-stream insertions and modifications that are prohibitive with hierarchal objects. While other software such as SideFX's Houdini (SideFX, 2022) and The Foundry's Nuke (The Foundry, 2022) are used for different stages of production, they also use nodes instead of hierarchies to allow more direct control of their data processing. "The data can be accessed or previewed at different points of the process, as a snapshot of the input/output of a given node. This is quite similar to the concepts behind Houdini's node graph but quite foreign to Maya" (Nieto, Banks, & Chan, 2018). Using nodes for the ephemeral rig gives a familiar workflow to developers transitioning from other software into Maya and reduces the potential learning curve.

The inspiration of the ephemeral system was drawn from a SIGGRAPH paper published in 2019 titled *Fast, Interpolationless Character Animation Through "Ephemeral" Rigging.* The author created a plugin using Python to create an ephemeral system in Maya, similar to the one discussed in this thesis. Anzovin's ephemeral rig used dynamic rig construction to create "interchangeable forward and "backward" kinematics" (Anzovin, 2019), which is the heart of the ephemeral system discussed in this thesis. Anzovin's paper also focused on changing the way the animator interacted with the Maya timeline, which is outside the scope of this thesis.

4. Tools and Technologies

The tools and technologies chosen for this thesis were selected primarily on their use in the professional animation industry.

4.1 - Autodesk Maya

Autodesk Maya is the most common 3D suite used in film and game production. It was initially released in 1996 and quickly became an industry standard. "Maya's been used on every [Oscar] winning film since 1997…" (Terdiman, 2015). The versatility of this platform allows its use in video games, film, advertisement, education, and other 3D industries.

The program is built on C++ code (Stroustrup, 2021) and hosts a complex and customizable user interface. Maya also provides an API for plugin creation in either C++ or Python. It also supports a custom language MEL (Maya Embedded Language), which is used for internal commands.

Maya works as a collection of nodes operating inside a scene. "A Maya scene is a system of interconnected nodes that are packets of data. The data within a node tells the software what exists within the world of a Maya scene" (Derakhshani, 2015). All nodes in Maya are classified as Dependency Graph (DG) nodes. DG nodes can store data, perform calculations, receive inputs, and generate outputs. The nodes are stored in a network and use unidirectional edges to connect to each other. There is a subclass of DG nodes called Directed Acyclic Graph (DAG) nodes. These are nodes that also exist in 3D space. DAG nodes are always stored in a hierarchy and have additional functionality specific to their relationship to other DAG nodes and the 3D space.

A humble cube in Maya sitting at the world origin has two nodes of its own and one more connected to it. The cube has a transformation node, which is a DAG node, and shows in the hierarchy on the left of Figure 9. This stores the location, rotation, and scale information of the object, along with its hierarchal relationships to other objects. The cube also has a shape node, which is a DG node. This node contains information about the vertices, edges, and faces that make up the displayed object in the scene. Both nodes have a distinct function and are tied together by Maya. The shape node is also connected to a shadingGroup node, which informs Maya which material to display on the object.

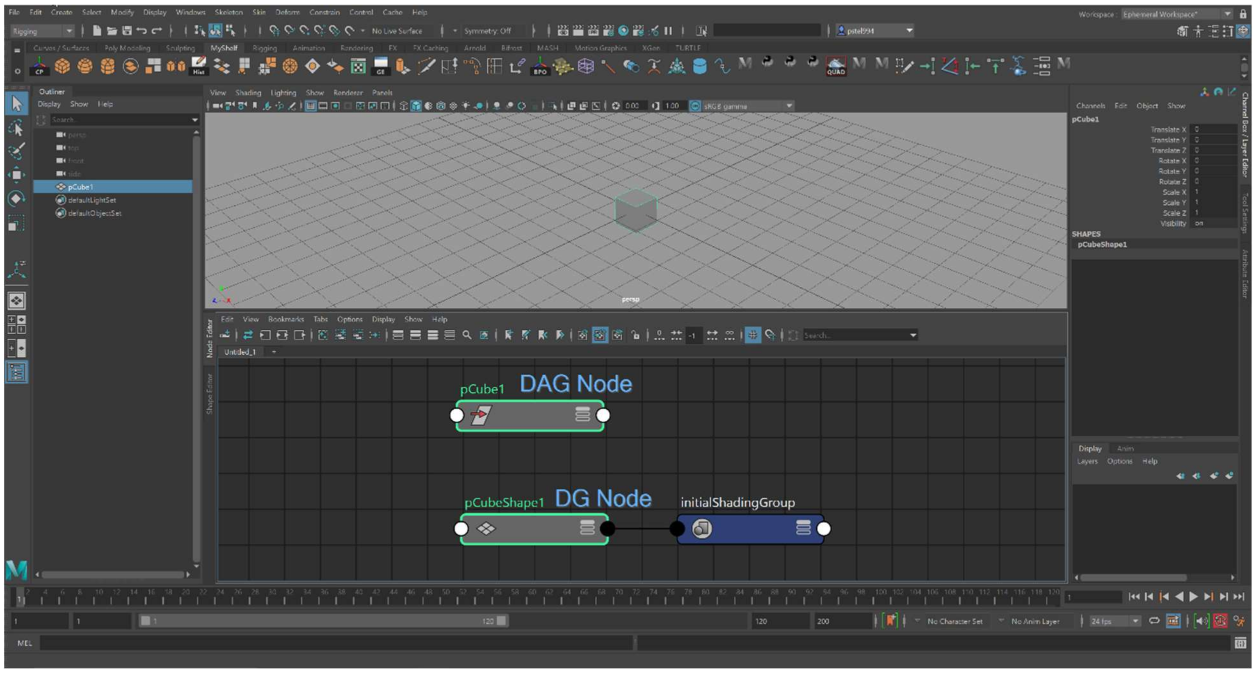


Figure 9: Illustrating the DG and DAG Nodes in Maya

4.2 - Visual Studio

The ephemeral rig plugin was written inside Visual Studio, which is an Integrated Development Environment (IDE). This environment supports the programming language C++

(Microsoft, 2021), which was the programming language chosen for this project. Inside the installed files for Maya is a developer kit with hundreds of included libraries used for the creation of Maya plugins in C++. Visual Studio allows the inclusion of these libraries which are necessary for the compilation of the coded plugin. The code is compiled into an .mll, which is a Maya version of a .dll file. This .mll file is read in by Maya through its plugin API and allows the functionality of the ephemeral rig into the Maya system.

4.3 – C++, Python or MEL

When comparing the three programming languages of C++, Python, and MEL, C++ has several measurable benefits. The first benefit is that C++ is far more efficient, especially during execution. When a program is compiled, the written source code of the plugin is converted into machine code. This machine code is what runs during the execution of the program. Python needs an interpreter to run alongside the machine code. This is due to the dynamic typing used by Python, which allows the user to not declare data types inside their written code, but forces the system to compensate during execution. C++ uses static typing, so there is no need for an interpreter. "Writing code in C++ is somehow difficult as compared to Python but when it comes to performance, C++ gives excellent results. Research proves that languages like Python, PHP, Java, and C# cannot compete with C++ in terms of speed and memory efficiency" (Zehra, Javed, Khan, & Pasha, 2020). Predefining the variables in C++ does add complexity to writing the code, but this is a tradeoff made for increased performance.

A second benefit of C++ compared to Python is threading. Due to the interpreter, Python is forced to run on a single thread. "Python's global interpreter lock (GIL) forces all python nodes to be scheduled serially. This prevents [the system] from evaluating efficiently and causes a major bottleneck in system performance" (Kahwaty, Yoder, Lin, Lee, & Suroviec, 2019). C++, however, can use multiple threads, which decreases run time. This is most applicable in the recursive situations used in the functionality of the ephemeral plugin.

A third benefit of C++ is the integrated concept of object-oriented design. Every node created by the plugin is an instantiation of an ephemeral node, which encapsulates the functionality of the system. Object-oriented design also encourages code reuse, as the objects each contain a suite of functions that can be called at specific times. "The concept of software reuse is not new and is used in many techniques, methods, and processes. These include portability in the classical sense, code-sharing in successive release, common subsystems, common routines (subroutines and functions of language constructs), and repeated exploitation of algorithms" (Li & Kiran, 1996). This encapsulation also helps integrate with existing Maya nodes, as the outputs and inputs can be controlled and obscure the internal functions. This is also why C++ was chosen instead of MEL, as MEL is a scripting language, so it does not natively support object-oriented design and classes.

4.4 - Maya API Objects

The Maya API gives access to a multitude of incredibly specific objects. Nearly all of these objects have connections or related objects that rely on each other to function. There are several types of objects referenced throughout the explanation of the ephemeral rig. These are defined in the glossary, but the most general object, an MObject, and its constituents are explained here.

An MObject is primarily a storage device for data and can execute simple commands. As an example, to read or modify the contents of a string value inside an MObject, or Maya Object, several steps must be taken. Figure 10 shows the structure of this example MObject. First, the node must be located in the Maya scene and the MObject tied to that node is stored. An MFnDependency node must then be created. This dependency node allows access to the MPlugs that are stored on the MObject. An MPlug is a connection point to an attribute inside the MObject, which in this case contains the string. The specific MPlug relating to the desired string attribute must be found next. Finally, the value contained inside that MPlug can be read and modified through the built in Maya API functions.

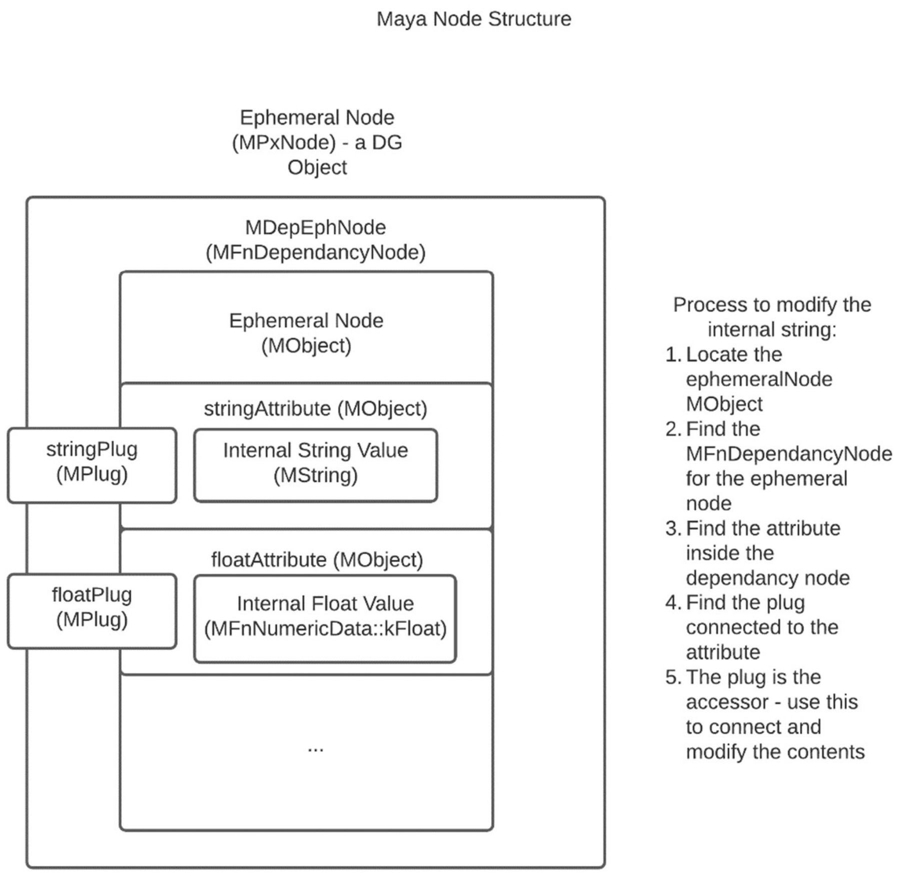

Maya Node Structure
Ephemeral Node (MPxNode) - a DG Object
MDepEphNode (MFnDependancyNode)
Ephemeral Node (MObject)
stringAttribute (MObject)
stringPlug (MPlug)
Internal String Value (MString)
floatAttribute (MObject)
floatPlug (MPlug)
Internal Float Value (MFnNumericData::kFloat)
...
Process to modify the internal string:
1. Locate the ephemeralNode MObject
2. Find the MFnDependancyNode for the ephemeral node
3. Find the attribute inside the dependancy node
4. Find the plug connected to the attribute
5. The plug is the accessor - use this to connect and modify the contents

Figure 10: Generic Maya Node Structure

4.5 - Dirty Nodes

All MPlugs in Maya can be marked as either dirty or clean, and start in a clean state. In order to preserve computation time, Maya only updates an object's values if the related MPlug is marked as dirty and then the value is requested by another node. For example, when the user moves a cube in the X Axis in the scene, the attribute plug for the X Axis of that cube is marked as dirty, along with everything connected to that plug's output. At the start of the next compute cycle, Maya checks for any dirty plugs in the scene. Finding the dirty X Axis plug on the cube, Maya then checks if the output is needed. Since the output is required by the scene to render where the cube is, Maya calls the cube's compute function, which handles updating the output. Once the output is calculated, the plug is then marked clean.

When designing Maya MObjects, the designer must designate which attributes will dirty others on the node. If an input value on the node is changed, Maya has no default way to know which output, if any, should be marked dirty and require updating. These relationships are usually declared during the creation of the MObject, but they can be declared during the run dynamically.

The ephemeral rig uses these relationships to control when to update the node's state. If one of the ephemeral node's inputs is updated, the ephemeral node is dirtied, and the compute function is called. This controls the ephemeral system's creation and destruction of the temporary rigs.

5. Design and Implementation

5.1 – Design Overview

To implement the ephemeral rig, the plugin must be loaded into Maya through the plugin interface. Once the plugin is loaded, the Maya scene needs to be set up with several interconnected objects. Each object has a distinct role and relationship to the other objects.

At the core of the rig is a central object, the Master Node, along with an Ephemeral Node for each joint in the rig. The master node acts upon all ephemeral nodes in the scene. The design of the ephemeral rig is each independent ephemeral node in the scene does not have control over the other ephemeral nodes. Each ephemeral node is only responsible for its own data management and connections. The master node contains data structures to track the scene state, including the current interaction mode the user has set (forward, inverse, etc.). One of the most important internal objects to the master node is the activeEphNodeDict – a map object that stores the name and MObject for each ephemeral node currently constrained in the active rig. The master node also contains data structures that track constraints, IK handles, and temporary joints created in the scene. Finally, the master node controls the creation, tracking, and destruction of the temporary rigs created throughout the animation process.

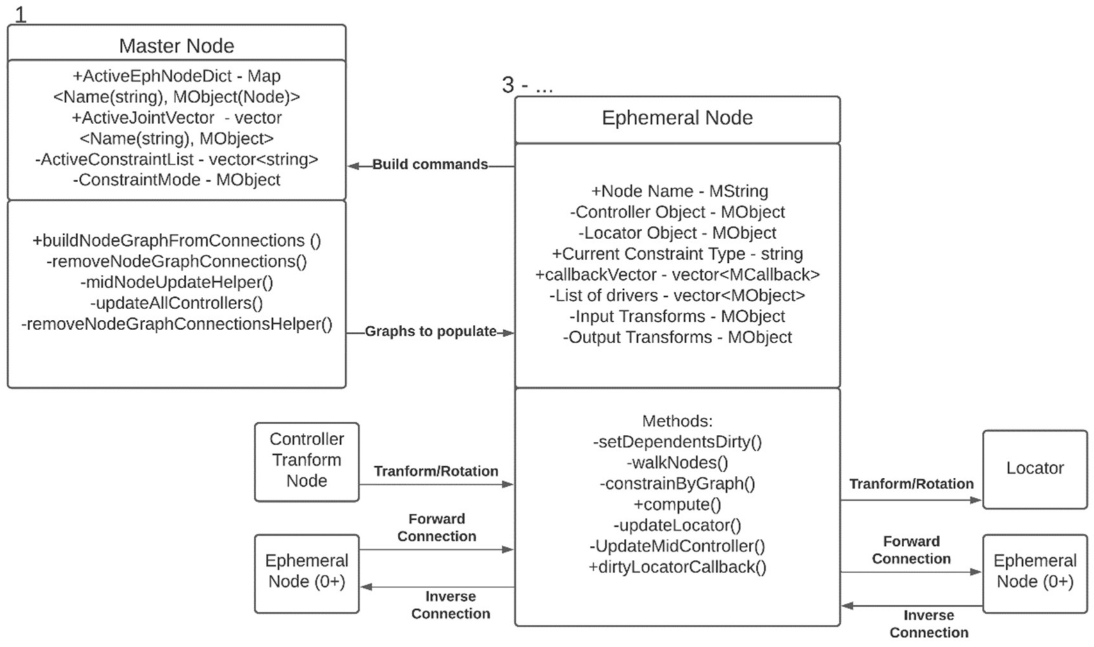

1
Master Node
+ActiveEphNodeDict - Map <Name(string), MObject(Node)>
+ActiveJointVector - vector <Name(string), MObject>
-ActiveConstraintList - vector<string>
-ConstraintMode - MObject
+buildNodeGraphFromConnections ()
-removeNodeGraphConnections()
-midNodeUpdateHelper()
-updateAllControllers()
-removeNodeGraphConnectionsHelper()
3 - ...
Ephemeral Node
+Node Name - MString
-Controller Object - MObject
-Locator Object - MObject
+Current Constraint Type - string
+callbackVector - vector<MCallback>
-List of drivers - vector<MObject>
-Input Transforms - MObject
-Output Transforms - MObject
Methods:
-setDependentsDirty()
-walkNodes()
-constrainByGraph()
+compute()
-updateLocator()
-UpdateMidController()
+dirtyLocatorCallback()
Build commands
Graphs to populate
Controller Tranform Node
Tranform/Rotation
Forward Connection
Inverse Connection
Ephemeral Node (0+)
Locator
Tranform/Rotation
Forward Connection
Inverse Connection
Ephemeral Node (0+)

Figure 11: Class Diagram of Ephemeral System

The rig requires a joint skeleton bound to a 3D character model. This skeleton is what drives the movement of the character by influencing the vertices of the object. The connection between the character and the skeleton is handled by Maya outside the ephemeral rig, and is common practice for all rigs.

Every joint in the bound skeleton needs several objects and connections for the ephemeral rig to operate. First, the bound joint itself needs to be oriented and named. A locator, which is an object with only transformation values, needs to be created at the same space and orientation as the joint, and the joint needs to be constrained to follow this locator. A controller will also share this exact space, and is what the animator will directly interact with to drive the ephemeral rig. Finally, an ephemeral node needs to be created and connected to the controller.

The controller's translation and rotation will feed directly into the ephemeral node, informing it of user updates.

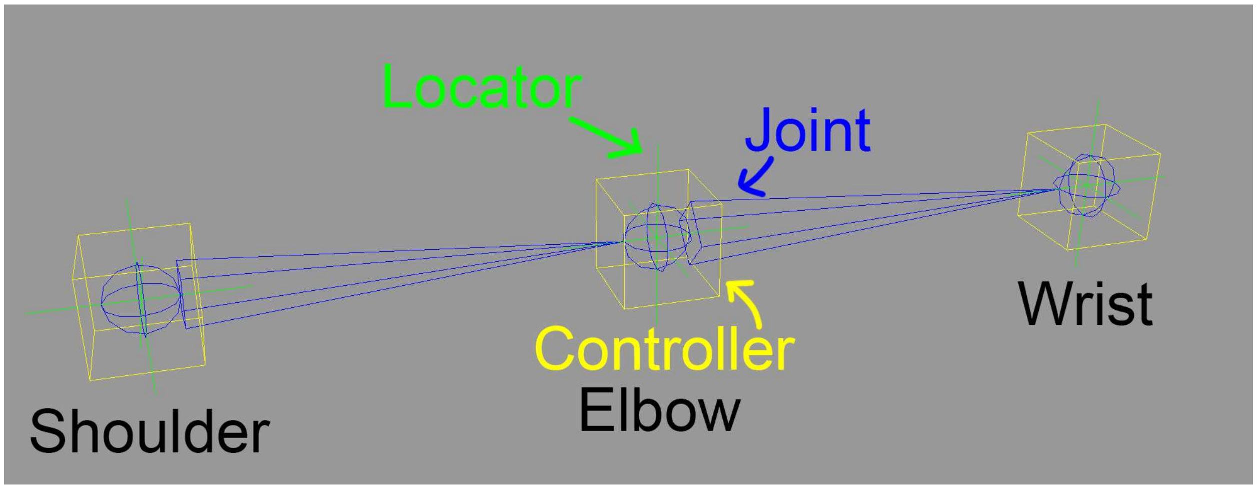


Figure 12: Ephemeral System Basics in Scene

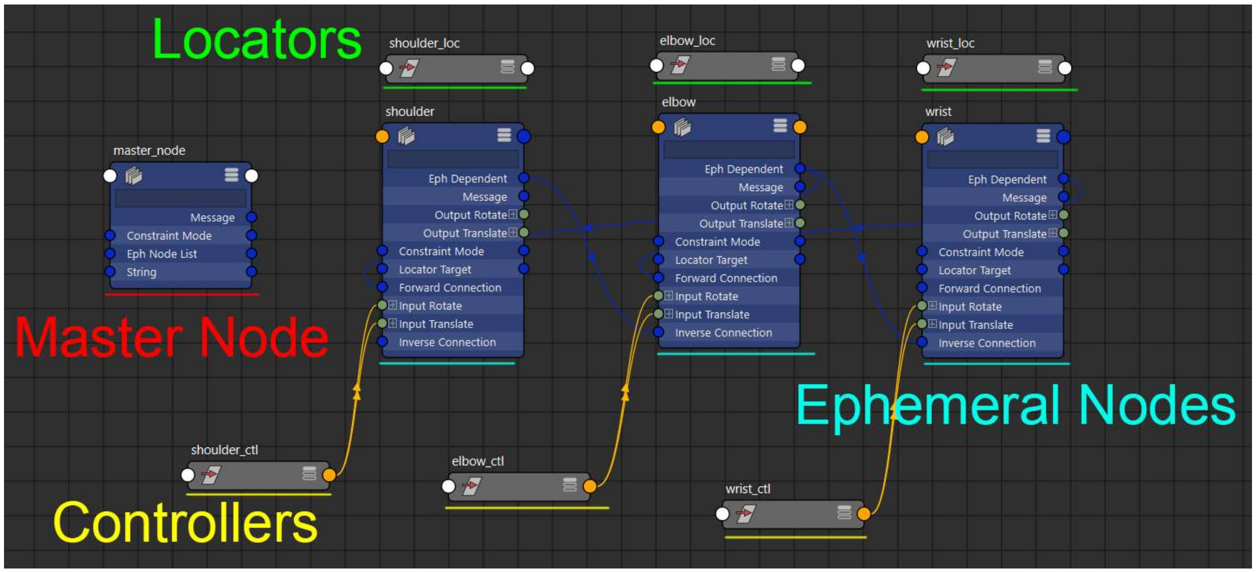


Figure 13: Ephemeral System Connections in Connection Graph

Once each joint is set up with all the objects, the ephemeral nodes need to be connected. Using built in attributes, the nodes are connected to other ephemeral nodes based on the desired relationships. Forward connections are sent down the chain, so the shoulder ephemeral node's Forward Connection connects to the elbow's Ephemeral Dependent, and the elbow's Forward

Connection connects to the wrist's Ephemeral Dependent. The inverse is set up the same way, where the wrist's Inverse Connection connects to the elbow's Ephemeral Dependent, and so on up the chain. Each ephemeral node also stores the related locator name as a string inside an internal attribute.

The locators that are connected to each joint serve a necessary purpose and are part of the core of the ephemeral rig. They act as an in-between for the ephemeral rig and the bound skeleton. Each joint's locator and controller always share the same location in the scene. During operation, the ephemeral rig creates constraints on the controllers. Additionally, keyframes, which are constraints that store animation data, need to be connected to the rig. If both the keyframes and ephemeral constraints were on the controllers, there could be possible conflicts and unexpected behavior. Instead, the locators are set up to connect to the keyframes, which separates the constraints and reduces unpredictability.

It is essential that the locator only has input connections from the keyframes. However, the locator needs to follow the controller's movement. The rig directly updates the locator whenever the controller is moved. This allows the rig to create and destroy connections to the controllers freely without worrying about damaging animation data. The relationship between the controller and locator can be reversed so the locator sets the position of the controller. For example, in animation playback, the locator has the animation data and moves in the scene. The controller needs to follow the locator's animation and the controller's position is directly updated. This ephemeral connection is part of what makes the rig so adaptable.

5.2 - Scene at Rest

Once every ephemeral node is set up and all the connections are complete, the animator can begin interacting with the rig. When the scene is at rest, there are no connections between the controllers. The only connections are between the related ephemeral nodes, explained above, which informs the rig of the scene's layout.

When an animator interacts with one of the controllers, the related ephemeral node needs to be recomputed. The rig marks this node as dirty. On the next compute step, Maya calls the compute function of the dirty ephemeral node, which then calls the master node to construct the connections in the scene.

5.3 - Four Interaction Modes

There are four modes the animator can set for the behavior of the ephemeral rig. The four modes can be swapped between at will and only exist for the required time span.

The first interaction mode is Forward mode. When this mode is active, the manipulation of the selected controller will directly affect all the child controllers. This mirrors the functionality of an FK system. If the shoulder rotates in forward mode, the elbow and wrist will follow from the shoulder's pivot point.

The second interaction mode is Pseudo Inverse mode. This mode takes the delta of the manipulated controller's updated position compared to its starting position. The delta is then divided and applied evenly to all controllers between the active controller and the uppermost parent controller. If the wrist is moved 5 cm in the X axis while in pseudo inverse mode, the elbow will move 2.5 cm in the X axis. The movement in this mode imitates IK, but does not preserve volume like a traditional IK system.

The third interaction mode is Inverse mode. Inverse mode leverages Maya's internal IK capabilities to drive the rig. To do so, a temporary joint chain is created at the rig's position and is bound to a Maya IK Handle. The IK Handle calculates the IK movement. The temporary joint chain is then set to update the ephemeral rig's controllers, which move the joints in the rig.

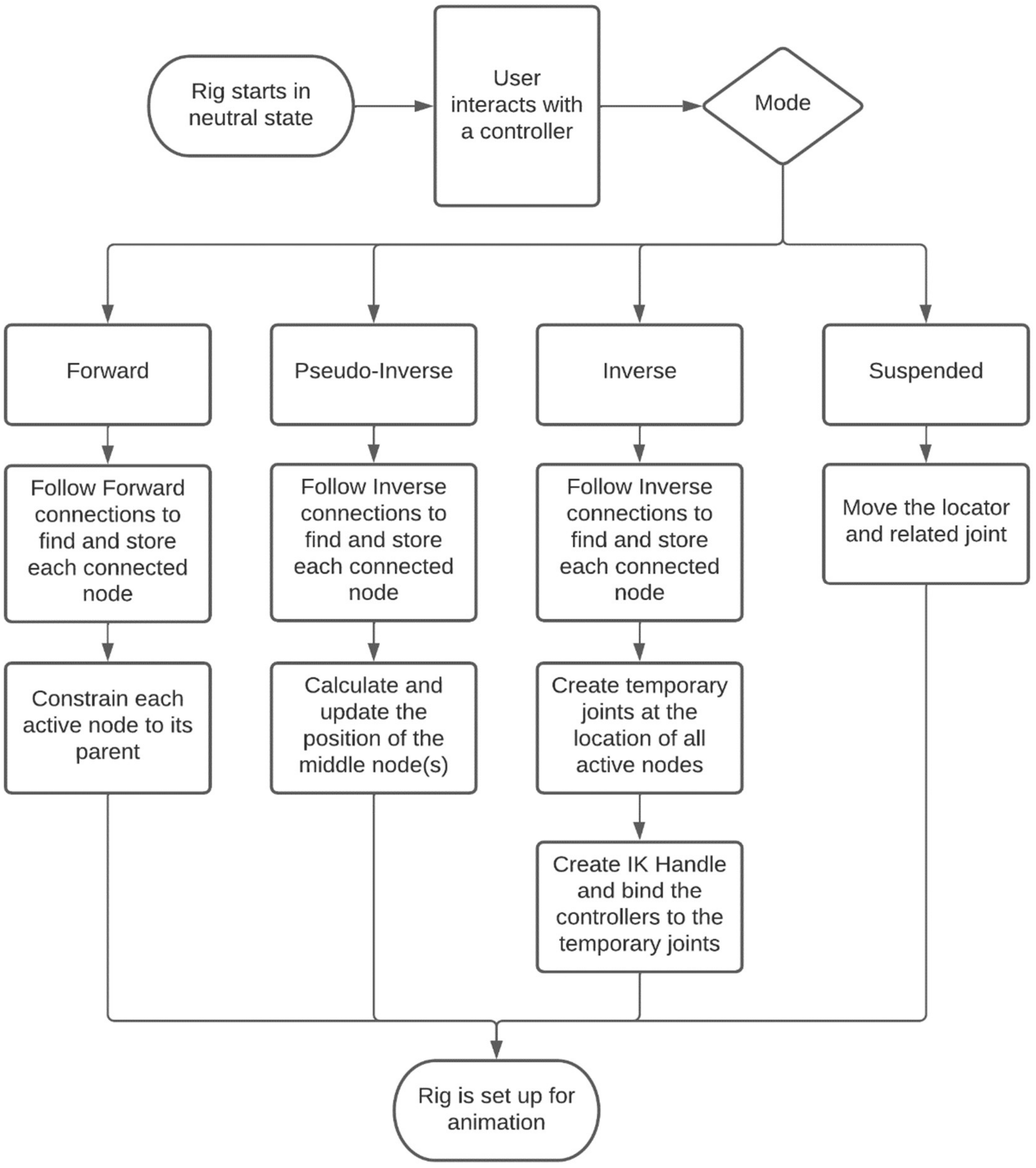

Rig starts in neutral state
User interacts with a controller
Mode
Forward
Pseudo-Inverse
Inverse
Suspended
Follow Forward connections to find and store each connected node
Follow Inverse connections to find and store each connected node
Follow Inverse connections to find and store each connected node
Move the locator and related joint
Constrain each active node to its parent
Calculate and update the position of the middle node(s)
Create temporary joints at the location of all active nodes
Create IK Handle and bind the controllers to the temporary joints
Rig is set up for animation

Figure 14: Overview of the Setup for the Four Interaction Modes

The final mode is Suspended mode. Suspended mode permits the animator to directly manipulate a single controller without affecting any others. This is primarily useful for fine detail manipulations.

There are several reasons that there are two inverse modes. The primary reason is to explore the possible solutions to an ephemeral IK rig. Without having prior experience with ephemeral rigging, putting both inverse modes into practice allows the animator and rigger to decide which mode they prefer. Comparing one to the other, the pseudo inverse mode is simpler and more adaptable. The entirety of the calculations is contained inside the ephemeral rig, so any Maya software changes would not affect the ephemeral rig's performance. By increasing the complexity of the calculation, the developer could increase the functionality of the pseudo inverse mode to act closer to true IK or potentially as something else entirely. Inverse mode depends on Maya's existing IK calculations and has the benefit of reusing existing work. However, inverse mode needs to create objects on demand and connect the necessary attributes. This creation and manipulation has a higher chance of unforeseen consequences, as the attributes and behavior of the temporary created objects are outside the ephemeral rig.

5.4 - Walk Nodes

When the ephemeral rig is instructed to build forward, inverse, or pseudo inverse mode, the first step is to search for connected nodes in either the forward or inverse direction. To reuse code, each of these modes calls the function walkNodes. The functionality of walkNodes is summarized here for reference. Figure A1 in Appendix A gives a more in-depth view of the steps and data management.

The purpose of walkNodes is to load into the master node every currently affected ephemeral node based on the interaction mode and the animator's selected controller. If the animator selects the shoulder controller and is in forward mode, the connected nodes in that direction to be found are elbow and wrist. WalkNodes discovers and stores the connected nodes into the master node's activeEphNodeDict.

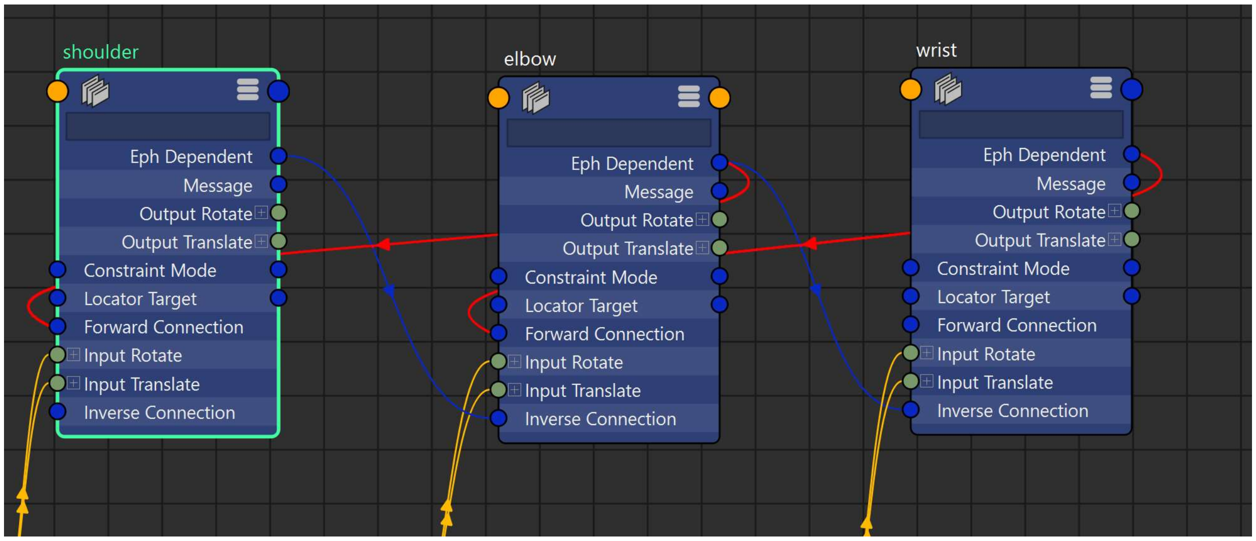


Figure 15: In this example, walkNodes is following the red connections from Forward Connection on the shoulder to the Eph Dependent on the elbow, which then follows the same connection to the wrist.

Walking through the function, walkNodes first checks if the current ephemeral node is already stored in the activeEphNodeDict. If it is already stored, there is no reason to store this node again, so the walkNodes function ends. If the active node is not stored, it is stored into the activeEphNodeDict. The next step is to find the connected ephemeral nodes in the desired direction. The function reads from the Forward Connection/Inverse Connection MPlug (depending on mode) and grabs the connected ephemeral nodes. Then walkNodes is called recursively on all the ephemeral nodes that were found from these connections, continuing the process. If there are no connected ephemeral nodes, the function simply exits.

The use of recursion has one primary purpose. If there is a branching path, for example at the top of the spine going into two arms and a head, walkNodes will be able to travel down all these paths and store each of the affected ephemeral nodes. Since multiple paths can be running in multiple threads, the rig can't rely that the MObjects were added in a linear parent to child order. This means that in the activeEphNodeDict, the ephemeral node MObjects are stored using the default C++ ordering instead of chronological ordering.

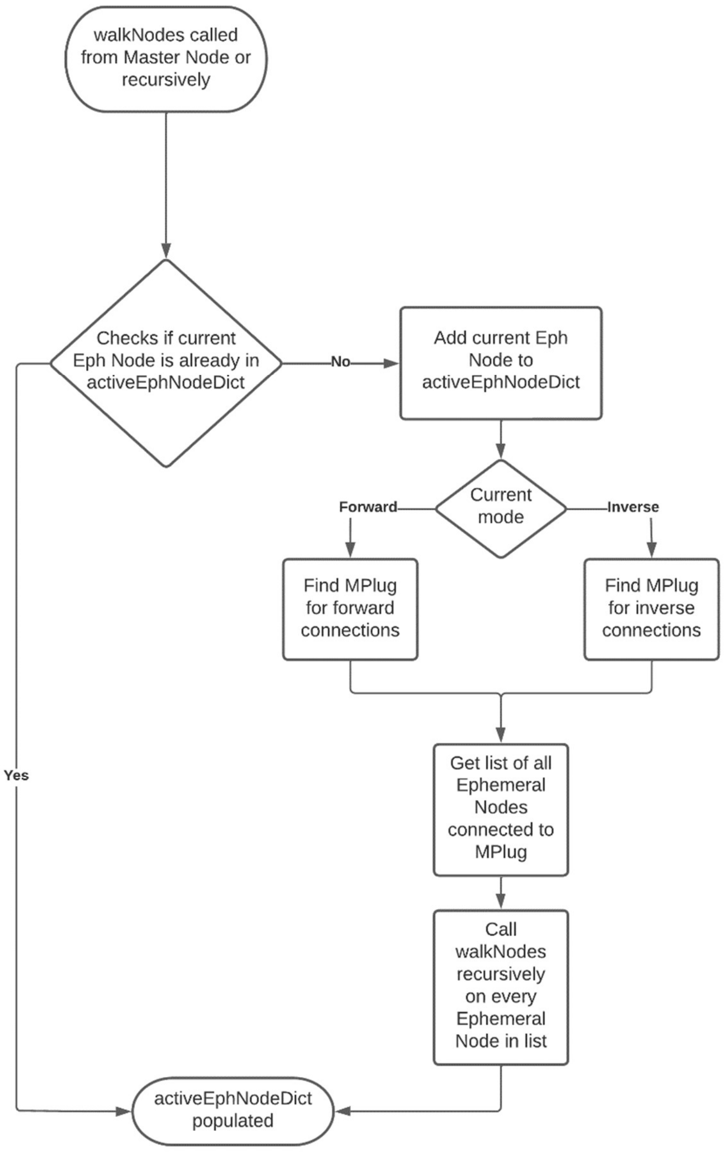

walkNodes called from Master Node or recursively
Checks if current Eph Node is already in activeEphNodeDict
No
Add current Eph Node to activeEphNodeDict
Current mode
Forward
Inverse
Find MPlug for forward connections
Find MPlug for inverse connections
Get list of all Ephemeral Nodes connected to MPlug
Call walkNodes recursively on every Ephemeral Node in list
Yes
activeEphNodeDict populated

Figure 16: Walk Nodes Flowchart

5.5 - Update Locators

The locators for each ephemeral node are not connected while at rest. The ephemeral rig is set up in this manner to keep the inputs of the locators free from connections so they can be keyframed without conflict. This setup does create an issue, as the locators need to be updated to match the position of the respective controllers. The issue is resolved by having any dirtied ephemeral node call a function named updateLocatorPosition. As updateLocatorPosition is used in all four interaction modes, the functionality is outlined below to prevent unnecessary repetition. A more detailed view can be found in Figure A2 in Appendix A.

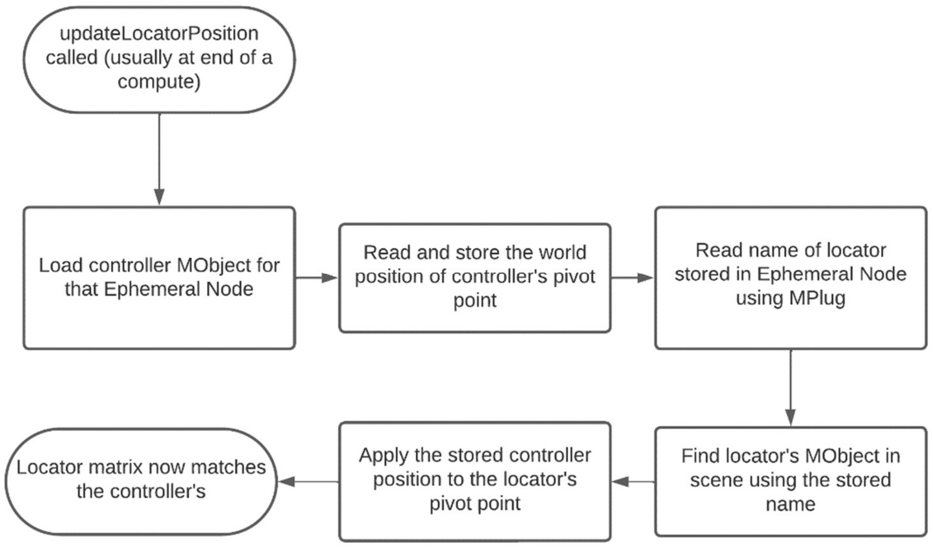

updateLocatorPosition called (usually at end of a compute)
Load controller MObject for that Ephemeral Node
Read and store the world position of controller's pivot point
Read name of locator stored in Ephemeral Node using MPlug
Find locator's MObject in scene using the stored name
Apply the stored controller position to the locator's pivot point
Locator matrix now matches the controller's

Figure 17: Update Locators Flowchart

UpdateLocatorPosition is usually called after the master node has finished setting up one of the interaction modes and returned to the active ephemeral node to finish cleanup. During updateLocatorPosition's execution, the first step is to load the matrix for that ephemeral node's controller. The ephemeral node's controller is found by accessing the MPlug connecting the ephemeral node to the controller, and the controller's MObject is stored. The world position matrix is retrieved from the controller. Next, the locator must be found. Every ephemeral node contains the name of the associated locator as a string attribute. The locator is found in the scene by name and the locator's MObject is stored. Once the locator is stored, the values of the world position matrix are used to set the matrix MPlug on the locator. When this is complete, the locator shares the position of the controller and no permanent connection between the controller and the locator was necessary.

In addition to updating the locator position, the locator must also be marked dirty. At the start of the compute for the ephemeral node, an overridden function from the Maya API is automatically called. This function loads the locator object and finds the output MPlugs, which are temporarily connected to all the input plugs on the ephemeral node. If any of the incoming ephemeral node's values are dirtied, such as when a controller moves or rotates, the locator's output plugs are dirtied as well. This is necessary as the bound joints are connected to the locators. The dirty status of the locator outputs tells Maya to update the connected joints as well, which is critical to the ephemeral rig moving the bound joint chain. Without this, Maya would not calculate the updated positions of the joints and the joints would not move.

5.6 - Forward Mode Details

Forward mode leverages parent constraints inside Maya to function. The end goal of forward mode is that each controller is parent constrained to its parent controller. When the parent controller selected by the animator is manipulated, each child controller follows its parent. This mimics the functionality of a Forward Kinematics system.

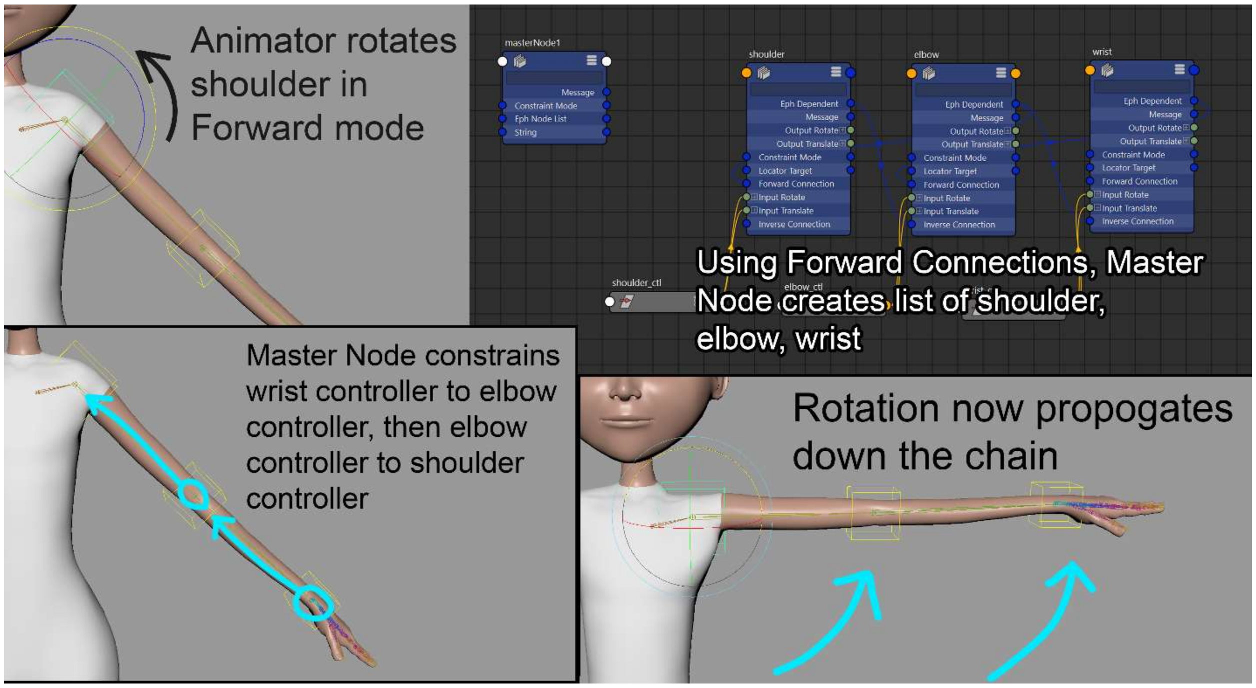

Animator rotates shoulder in Forward mode
masterNode1
Message
Constraint Mode
Eph Node List
String
shoulder
elbow
wrist
Eph Dependent
Message
Output Rotate
Output Translate
Constraint Mode
Locator Target
Forward Connection
Input Rotate
Input Translate
Inverse Connection
shoulder_ctl
elbow_ctl
Using Forward Connections, Master Node creates list of shoulder, elbow, wrist
Master Node constrains wrist controller to elbow controller, then elbow controller to shoulder controller
Rotation now propogates down the chain

Figure 18: Forward Mode Example

In this example, the shoulder controller is rotated in forward mode. When the shoulder controller is rotated, the shoulder ephemeral node is dirtied and Maya calls that node's compute function. The compute function calls the master node, passing the shoulder node's MObject as a reference. The master node first checks the isConstrained attribute of the shoulder ephemeral node. If this value is set to true, there is no need to create a new network of constraints as they already exist. The master node would then return to that ephemeral node to update the locator. If the isConstrained value is false, the constraints need to be built.

The master node calls walkNodes, discussed earlier, which populates the master node's activeEphNodeDict with the shoulder, elbow, and wrist ephemeral nodes. Once this list is populated, the master node calls constrainByGraph on each ephemeral node in the activeEphNodeDict.

In constrainByGraph, using the incoming MPlugs, the controllers connected to each ephemeral node and their parent's ephemeral node are found. If there is no parent, as is the case with the shoulder, no constraint is made. Otherwise, the child controller is then constrained to the parent controller using a Maya MEL command. The resulting constraint is added to the master node's activeConstraintList, and the ephemeral node is marked as constrained.

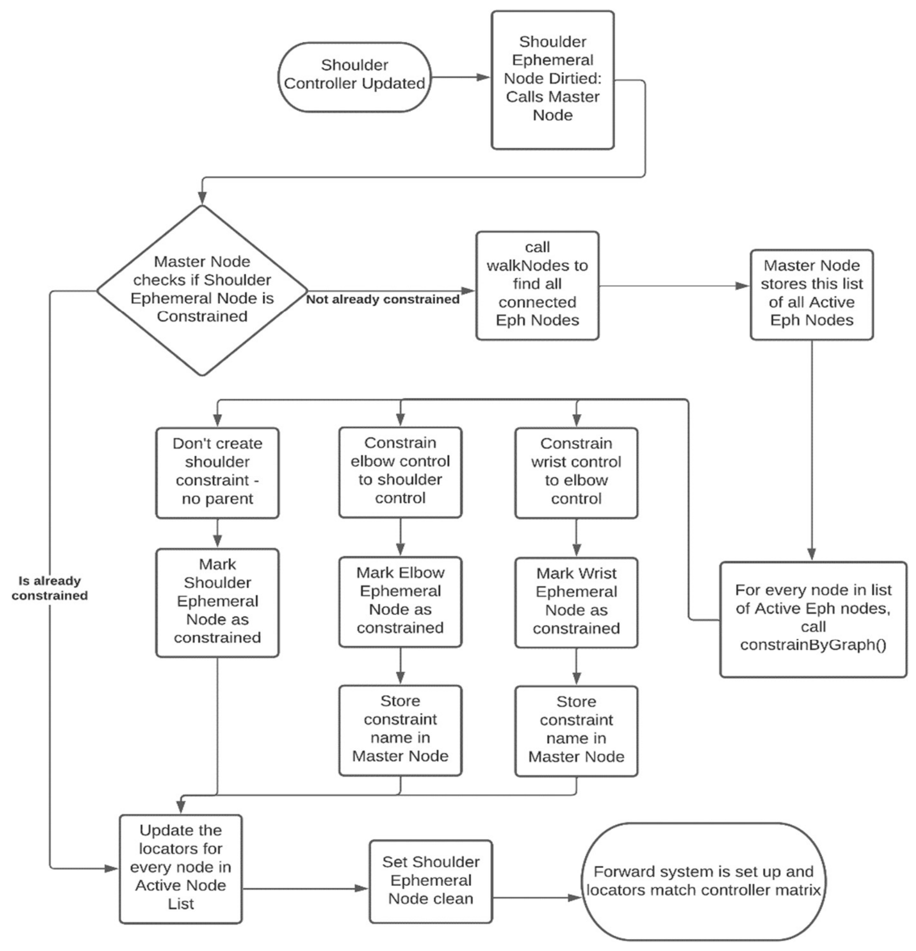

Shoulder Controller Updated
Shoulder Ephemeral Node Dirtied: Calls Master Node
Master Node checks if Shoulder Ephemeral Node is Constrained
Not already constrained
call walkNodes to find all connected Eph Nodes
Master Node stores this list of all Active Eph Nodes
Is already constrained
Don't create shoulder constraint - no parent
Constrain elbow control to shoulder control
Constrain wrist control to elbow control
Mark Shoulder Ephemeral Node as constrained
Mark Elbow Ephemeral Node as constrained
Mark Wrist Ephemeral Node as constrained
For every node in list of Active Eph nodes, call constrainByGraph()
Store constraint name in Master Node
Store constraint name in Master Node
Update the locators for every node in Active Node List
Set Shoulder Ephemeral Node clean
Forward system is set up and locators match controller matrix

Figure 19: Forward Mode Creation

After all the constraints are made, or if they already existed, the master node returns to the active ephemeral node, which in this case is the shoulder. The active ephemeral node calls updateLocatorPosition, which updates all the active locators to match the current position of the controllers. Finally, the dirty plug on the ephemeral node that called this compute function is marked as clean to inform Maya the calculation is complete. Figure 19 is a flow chart showing the progression through the process. For a more detailed diagram, see Figure A3 in Appendix A.

5.7 - Pseudo Inverse Mode Details

Pseudo Inverse Mode uses a mathematical formula to calculate the distance to move the middle objects that are dependent upon the selected object's position change. Pseudo inverse mode mimics IK functionality, but does not maintain volume. The end goal of pseudo inverse mode is to update the position of all the middle controllers to match a percentage of the active controller's movement.

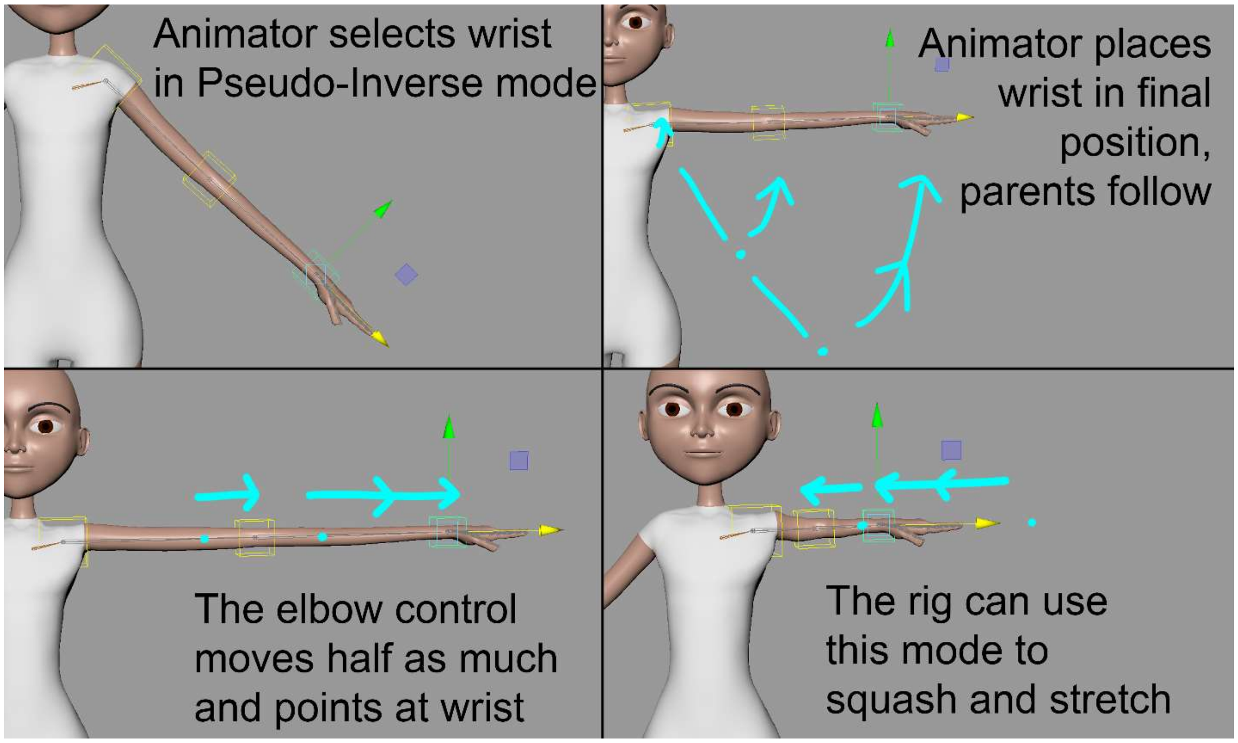

Animator selects wrist in Pseudo-Inverse mode
Animator places wrist in final position, parents follow
The elbow control moves half as much and points at wrist
The rig can use this mode to squash and stretch

Figure 20: Pseudo-Inverse Example

In this example, the wrist is moved in the X axis. The wrist ephemeral node is dirtied and the compute function is called. Once inside the compute function, the wrist ephemeral node calls the master node. The same first few steps are identical to the forward mode, where the master node checks to see if the wrist is already constrained and does not build any constraints if the wrist's isConstrained attribute is set to true.

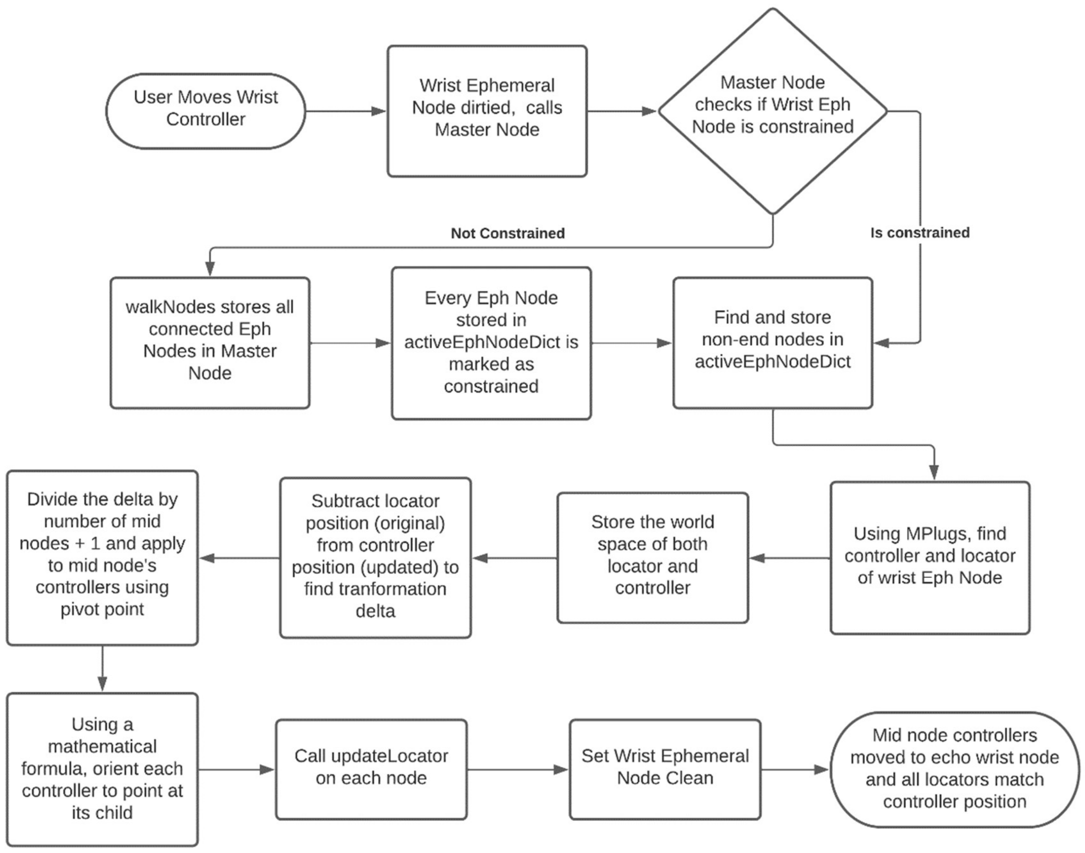

User Moves Wrist Controller
Wrist Ephemeral Node dirtied, calls Master Node
Master Node checks if Wrist Eph Node is constrained
Not Constrained
Is constrained
walkNodes stores all connected Eph Nodes in Master Node
Every Eph Node stored in activeEphNodeDict is marked as constrained
Find and store non-end nodes in activeEphNodeDict
Using MPlugs, find controller and locator of wrist Eph Node
Store the world space of both locator and controller
Subtract locator position (original) from controller position (updated) to find tranformation delta
Divide the delta by number of mid nodes + 1 and apply to mid node's controllers using pivot point
Using a mathematical formula, orient each controller to point at its child
Call updateLocator on each node
Set Wrist Ephemeral Node Clean
Mid node controllers moved to echo wrist node and all locators match controller position

Figure 21: Pseudo Inverse Creation

If the ephemeral node is not constrained, the master node goes into the pseudo inverse mode section and calls walkNodes to populate the activeEphNodeDict with the wrist, elbow, and shoulder MObjects. Once this is complete, the master node goes through each node in that dictionary and sets the node's attribute isConstrained as true using the MPlug. The master node also adds a dummy constraint to the activeConstraintList, as no Maya constraints are created for pseudo inverse mode.

At this point, the wrist is in the correct location and the related nodes are all marked as constrained. Next, the rig needs to update the position and orientation of the middle node. To do this, the master node returns to the wrist node, which checks if the interaction mode is in pseudo inverse. Since it is, the updateMidControllers function is called. updateMidControllers takes the activeEphNodeDict as a parameter, along with the active node, which in this example is the wrist ephemeral node. The first step is to find and store a temporary list of all the middle nodes, which in this case is only the elbow. Next, the delta of the wrist movement must be calculated. Using the wrist controller MObject, which has the updated position, and the wrist locator, which is still at the original position, the matrices of the two positions in world space are read and stored. The locator is still in the old position since the final step of each compute updates the locator position, and that has not been called yet.

The original (locator) values are subtracted from the new (controller) values, which generates the delta of the move. This delta is divided by number of middle nodes plus 1, to spread the values evenly over a chain. In this example, the only middle node is the elbow, so the delta is divided by 2. In other examples, such as a spine, there could be multiple joints and the effects would be spread out evenly.

The next step in pseudo inverse mode is to apply the divided delta to the middle node's controllers. The rotate pivot of the controller is set by adding the calculated delta to the controller's current world position. A formula is then applied to each active node where the orientation of the controller is updated to point at its child. These two updates in tandem create the IK behavior, where the parent joint positions and rotations are calculated in respect to the child's position.

Once all the middle controllers are updated, updateLocatorPosition is called on each ephemeral node and the locator positions are matched up to the controller positions. Finally, the function returns to the wrist ephemeral node, which updates its own locator position, marks itself as clean, and exits the compute. For a more detailed diagram, see Figure A4 in Appendix A.

5.8 - Inverse Mode Details

Inverse mode takes advantage of Maya's built in IK system to calculate the inverse movement of the parent joints. Inverse mode creates a temporary joint chain that mirrors the position of the controllers, creates an IK handle to calculate these joints, then constrains the IK handle to the active controller. Inverse mode also updates the middle controller of the ephemeral rig to match the middle joint of the temporary joint chain.

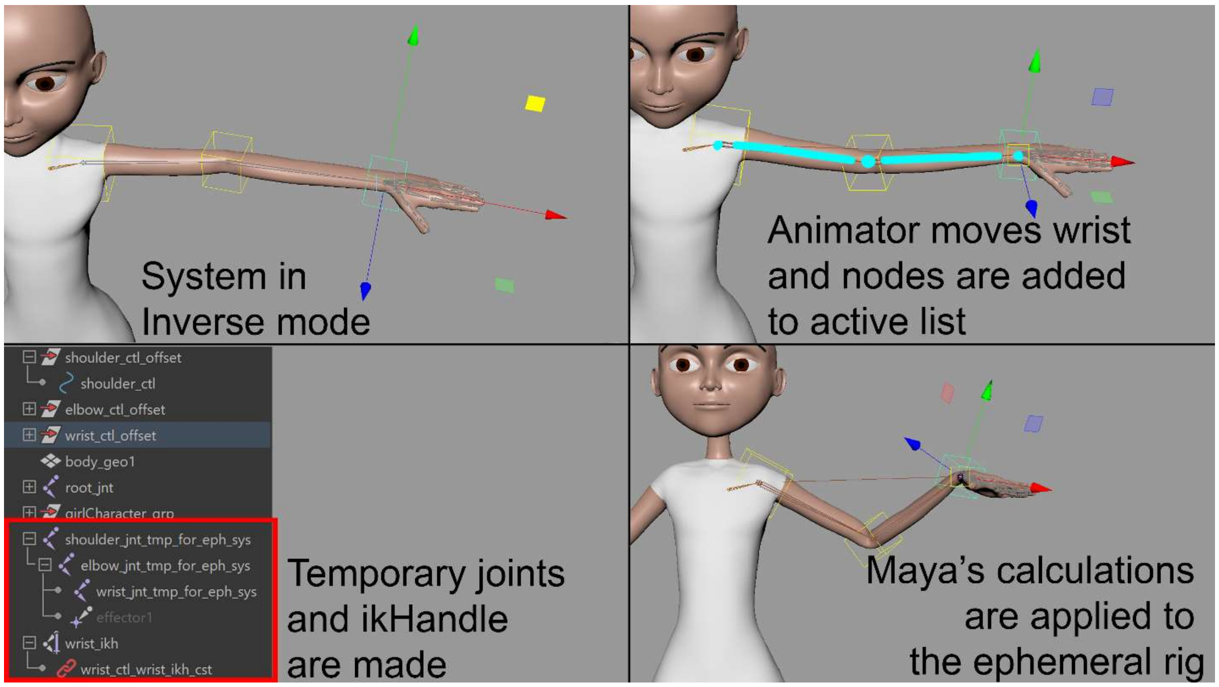

System in Inverse mode
Animator moves wrist and nodes are added to active list
shoulder_ctl_offset
shoulder_ctl
elbow_ctl_offset
wrist_ctl_offset
body_geo1
root_jnt
girlCharacter_grp
shoulder_jnt_tmp_for_eph_sys
elbow_jnt_tmp_for_eph_sys
wrist_jnt_tmp_for_eph_sys
effector1
wrist_ikh
wrist_ctl_wrist_ikh_cst
Temporary joints and ikHandle are made
Maya's calculations are applied to the ephemeral rig

Figure 22: Inverse Mode Example

In this example the wrist is moved in the scene. The wrist ephemeral node is dirtied and the compute function is called. Once inside the compute function, the wrist ephemeral node calls the master node. Just like in the previous modes, the master node checks to see if the wrist is already constrained and exits back to the wrist ephemeral node if the isConstrained attribute is set to true.

If isConstrained is set to false, the master node enters the inverse mode section and calls walkNodes to populate the activeEphNodeDict with the wrist, elbow, and shoulder ephemeral nodes. The next step calls createJoints on the wrist ephemeral node to begin the process of building and placing the temporary joints.

The first step in creating the temporary joints is finding the wrist controller's MObject through the MPlug. Then, a temporary joint is created and named based off the ephemeral node's name. This joint's name and MObject are stored in the master node in a vector named activeJointVector. A vector is a C++ object that maintains the creation order, which matters for the parenting and the creation of the IK handle, which happens later. The matrix of the active controller is loaded and applied to the matrix of the new joint. The current scene now has a temporary joint for the wrist ephemeral node, which shares position with the wrist controller.

To create the rest of the temporary joints, the incoming connection to the wrist ephemeral node, in this case the elbow, is found through the Inverse Connection MPlug. As long as there is an incoming connection, createJoints is called recursively up the chain to create, name, and place all the remaining joints. Once the joints are created, the joints are then set in the scene hierarchy in a parent to child chain. In this example, the shoulder joint is set as the parent of the elbow joint, which is the parent of the wrist joint. Finally, the wrist ephemeral node is marked as constrained and control goes back to the master node. The result is a temporary joint chain parented in the scene hierarchy and sharing the position of the current ephemeral rig's joint chain.

Next, the master node creates and names the IK handle. The master node uses a MEL command to create the handle, based on the front and back nodes in the ordered activeJointVector. Then, the handle is constrained to the wrist ephemeral node's controller so that the handle follows the controller's movements.

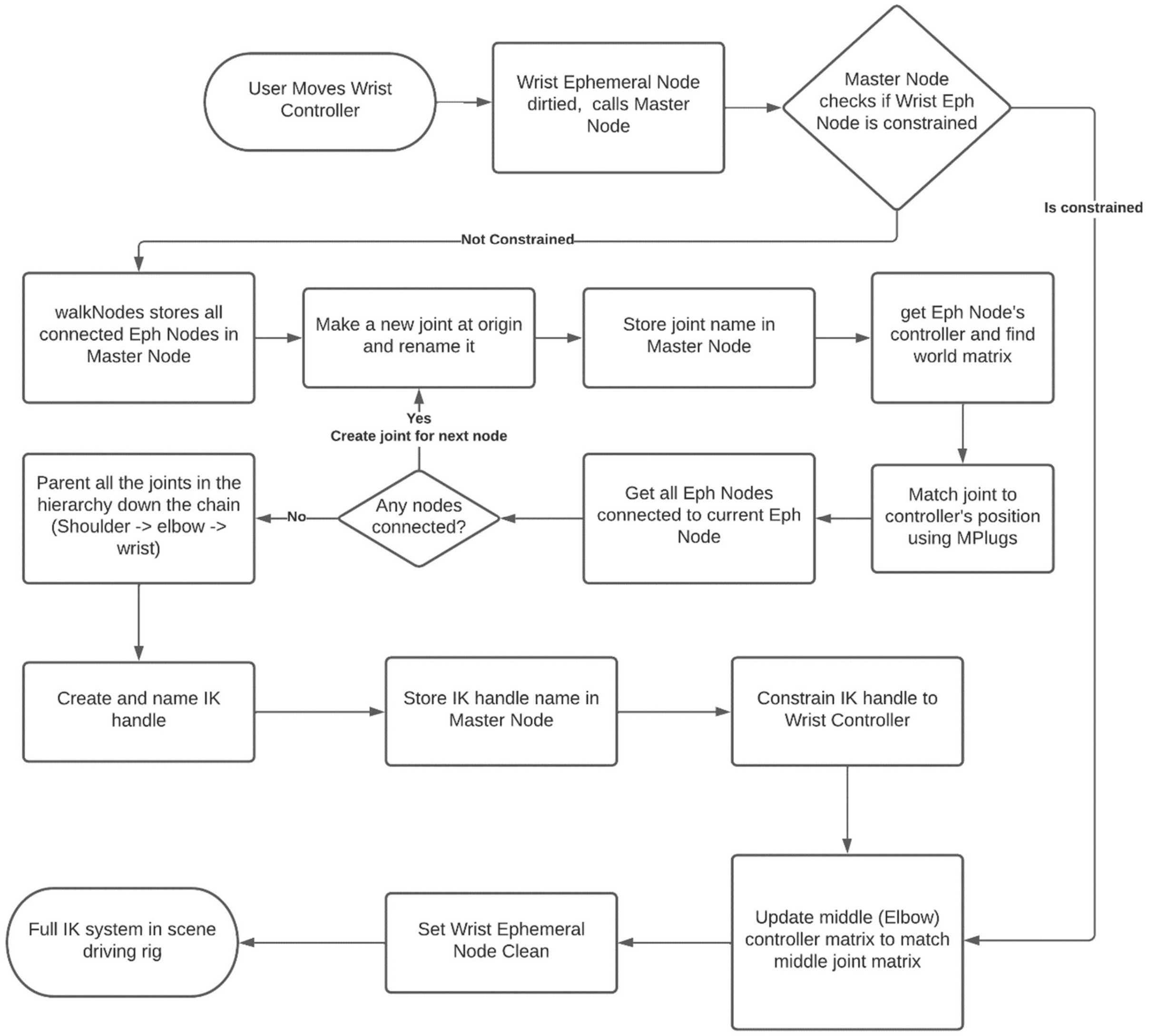

User Moves Wrist Controller
Wrist Ephemeral Node dirtied, calls Master Node
Master Node checks if Wrist Eph Node is constrained
Is constrained
Not Constrained
walkNodes stores all connected Eph Nodes in Master Node
Make a new joint at origin and rename it
Store joint name in Master Node
get Eph Node's controller and find world matrix
Match joint to controller's position using MPlugs
Get all Eph Nodes connected to current Eph Node
Any nodes connected?
Yes
Create joint for next node
No
Parent all the joints in the hierarchy down the chain (Shoulder -> elbow -> wrist)
Create and name IK handle
Store IK handle name in Master Node
Constrain IK handle to Wrist Controller
Update middle (Elbow) controller matrix to match middle joint matrix
Set Wrist Ephemeral Node Clean
Full IK system in scene driving rig

Figure 23: Inverse System Creation

Finally, when control is passed back to the wrist ephemeral node, the parent controllers need to be updated. Using a similar function to the updateLocatorPosition, the parent controllers are matched to the matrix of their respective temporary joints. This is so the shoulder and elbow controllers now follow the calculated temporary joints in Maya. Then, the updateLocatorPosition function is called on the wrist's locator, the wrist ephemeral node is marked clean, and the compute exits. The result of this function is a temporary joint chain calculated by Maya's inbuilt IK system and applied to the ephemeral rig. For a more detailed diagram, see Figure A5 in Appendix A.

5.9 – Suspended Mode Details

Suspended mode is the simplest of the four ephemeral modes. In suspended mode, the animator can move one controller and neither the parents nor the children will move or react. The purpose is primarily for fine tuning. If the animator wants to modify the rig beyond standard behavior, such as stretching the elbow out of place for an exaggerated effect, suspended mode will allow that.

Upon entering suspended mode, like all the others, the compute of the active node calls the master node. Then, the master node checks if the active node is already constrained. If it is not, the master node marks the active node as constrained and stores the ephemeral node's locator name and MObject in the master node. This allows the active ephemeral node to quickly find and update its locator. Once this is done, the ephemeral node updates the locator's position, marks itself as clean, and exits.

5.10 - Ephemeral Mode Destruction

Creating the ephemeral modes is the primary function of the rig, but to make the rig dynamic these modes need to be destroyed. After the mode is used and the animator wants to either change the active controller or the active mode, the previous mode needs to be automatically cleaned up and the node values reset.

Using a built-in Maya feature called callbacks, a listener is set up on the master node that activates every time the selection changes in the scene or the ephemeral mode changes. This callback calls the cleanup function, which methodically goes through all the temporary attributes stored in the master node and does the necessary cleanup.

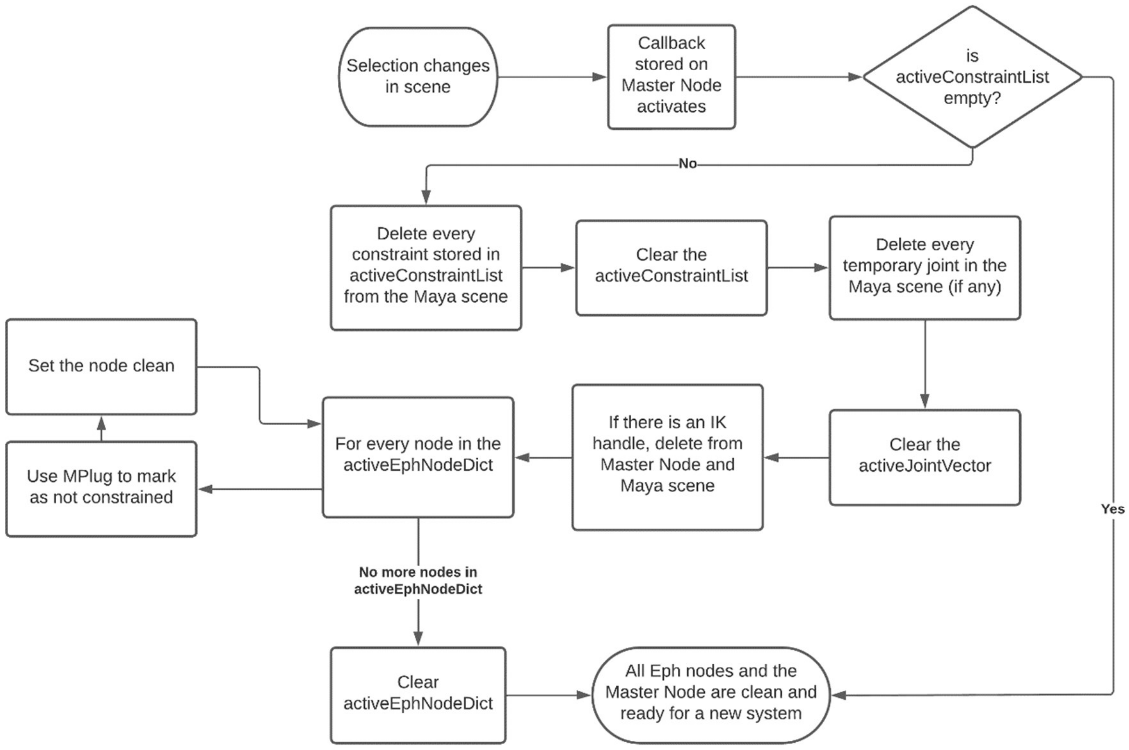

Selection changes in scene
Callback stored on Master Node activates
is activeConstraintList empty?
No
Yes
Delete every constraint stored in activeConstraintList from the Maya scene
Clear the activeConstraintList
Delete every temporary joint in the Maya scene (if any)
Clear the activeJointVector
If there is an IK handle, delete from Master Node and Maya scene
For every node in the activeEphNodeDict
Use MPlug to mark as not constrained
Set the node clean
No more nodes in activeEphNodeDict
Clear activeEphNodeDict
All Eph nodes and the Master Node are clean and ready for a new system

Figure 24: Ephemeral Graph Destruction

Upon first entering the cleanup function, the first check is on the master node's activeConstraintList, which is populated during the creation of any of the ephemeral modes. If the activeConstraintList is empty, it is assumed there are no active ephemeral systems in the scene, and there is nothing to clean up. This check is necessary as every selection change in the scene starts the callback. If there is anything in the activeConstraintList, all cleanup steps are completed in sequence. The first step is going through the activeConstraintList, deleting all the constraints in the Maya scene stored in this list, then emptying the list. The next step is going through and deleting, if applicable, all temporary joints and IK handles stored in the activeJointVector and IKHandle from the Maya scene, which would have been created in inverse mode, then emptying those lists. The final step is going through every node stored in the activeEphNodeDict, setting each node as not constrained using that node's isConstrained MPlug, then clearing that list.

The end result is all temporary objects are removed, all affected nodes are marked clean and not constrained, and the master node is back at the initial state. The Maya scene is at rest, the controllers have no connections, and the ephemeral rig is ready for the next mode.

6. Conclusion and Future Work

6.1 – Summary

With the ephemeral rig fully set up and operational, the animator starts with the scene in a neutral state. On demand, the animator can change between ephemeral modes which creates and destroys the connections explained in Chapter 5. The forward mode mimics the FK functionality and allows the animator to rotate down the chain. The pseudo inverse mode allows the animator to move the child in the scene, and the middle joints follow the movement by half. Each parent joint points to the child, as the shoulder always points toward the elbow, and so on down the chain. The inverse mode builds a temporary IK system and connects to it, and the suspend mode allows single joint manipulation. The animator can also add animation keys to the locators, which store positions over time for animation. When the animation plays, a callback fires in the scene that reverses the standard locator controller relationship. Usually, the controller position is pushed onto the locator, as the animator only interacts with the controller. In playback mode, the locators are moving to their animated positions, and the controllers are updated to match. This demonstrates the inherent flexibility of the ephemeral rig, as the directionality can be reversed when needed.

6.2 – Shortcomings and Future Work

With any new system, there are complications that reveal themselves once everything is integrated. The most noteworthy complication is a consistent issue with inverse mode. During creation of the temporary IK rig, the middle joint position sometimes changes when attaching the temporary joints to the IK handle. This is due to Maya expecting a clean neutral state for the joints when the IK handle is created. The joints are expected to have minimal rotation values and be oriented on the same rotation axis. This is not the case, as the animator is moving the joints around during the animation. Since the joints are not oriented cleanly, Maya moves the middle joint to compensate. This reduces the viability of the full inverse mode and illustrates the importance of encapsulating the IK functionality fully, which is the purpose of the pseudo inverse mode. The animator can mitigate this movement by using the suspend mode to place the middle joint back in the expected location, but the issue is still undesirable.

In addition, there are several lessons learned along the way that could be applied to either a new or updated version of the ephemeral rig. The first of these is the idea of storing the controllers in a hierarchy, similar to a more traditional FK system, instead of the controllers being unrelated to each other by default. This would essentially set the default state of the rig as forward and remove the need for creating the parent constraints on compute. There would be new challenges, such as preventing double transformations, as the parent controller might move the child controllers inadvertently. The ephemeral rig should be flexible enough to overcome this and would most likely be more efficient.

Another change worth implementing is the application of callbacks and dirty node propagation in the rig. The original design of the ephemeral rig was created with a simplified knowledge of the capabilities of the Maya API, so more advanced concepts were not considered. As development progressed, some callbacks were integrated. Using callbacks more consistently throughout the design would have given more direct access to functions and cleaned up node dirtying.

Dynamically linked attributes were added late in the process as well. Normally, attributes only dirty the other attributes in the same node. In Maya, it is possible to have attributes from one node dirty another node’s attributes. Not only was this dynamic link used so the controllers could dirty the disconnected locators, but also it could have been implemented more consistently throughout.

Another design change involves the handling of keyframes. In the ephemeral rig, keyframes are stored on the locators, as the locators' connections are always clean. However, the keyframes are partially hidden from the animator, making it difficult to know which frames have animations. It is worth considering reversing the controller locator relationship. Instead of constraining the controllers and having them ephemerally update the locators, the rig could constrain the locators and update the controllers. This would allow the animator to key directly on the controllers, which mimics how most rigs are controlled now, but maintains the clean connections the ephemeral rig currently has.

Finally, it would have been interesting to add more complex calculations to the pseudo inverse mode. Currently pseudo inverse mode does not fully conform to a true IK system, as it does not maintain volume. Due to encapsulation, the function that calculates the movement could be made more robust without affecting the rest of the rig. More thorough research is required to implement the formulas needed to run a fully accurate IK system.

Overall, an arm, as highlighted in this thesis, is a good proof of concept, but the ephemeral rig is designed to be applied to a full humanoid character. The scope of this thesis was limited to ensure the core systems are functional and reliable. Ephemeral rigging has the potential to be a solution to common rigging problems, such as those found in an inverse foot rig. Foot rigs need extra controls and blending to allow rolling, toe tapping, and heel motion, which tend to be in layered systems. An ephemeral foot rig would allow all these motions by switching between states as needed.

6.3 – Conclusion

The ephemeral rig, while still in the proof-of-concept stage, shows real promise with the versatility offered. It both demonstrates the power of the Maya API and runs incredibly quickly considering the number of calculations that need to be executed.

Overall, the concept of node based ephemeral rigging is a success and deserves further research. Merging forward and inverse systems into one gives the animator more control over their characters. The possibility of adding more modes beyond the ones discussed in this thesis is also an exciting consideration. Other complex rigs, such as reverse foot IK, hand rigs, and tail rigs, could all benefit from the ability to switch between modes dynamically. Node based dynamic rigs could one day become the industry norm, separating the technical knowledge requirement of the rig from artistic expression, allowing animators to quickly, easily, and intuitively control their characters and interact with them in ways they never thought possible.

Bibliography

Anzovin, R. (2019). Fast, interpolationless character animation through "ephemeral" rigging. *ACM SIGGRAPH 2019 Talks*. doi:https://doi.org/10.1145/3306307.3328165

Bhatti, Z., Shah, A., Shahidi, F., & Karbasi, M. (2013). Forward and Inverse Kinematics Seamless Matching Using Jacobian. *Sindh University Research Journal, 45*(2), 1-3.

Corvazier, C., & Robert, T. (2021). How the rig design impacts the animation process. *The Digital Production Symposium*. doi:https://doi.org/10.1145/3469095.3469278

Derakhshani, D. (2015). *Introducing Autodesk Maya 2016.* John Wiley & Sons, Incorporated. .

Elbmann, M. (2017). Universal IK FK Switch and Match Tool (PRO) 2.0.0 for Maya. *[Computer Software]*. Retrieved from https://www.highend3d.com/maya/script/universal-ik-fk-switch-and-match-tool-pro-for-maya

Industrial Light and Magic. (1993). Hollywood FX Masters - Terminator 2 CGI Special Effects.

Kahwaty, J., Yoder, W., Lin, A., Lee, G., & Suroviec, D. (2019). Optimizing rig manipulation with GPU and parallel evaluation. *ACM SIGGRAPH 2019 Talks*. doi:https://doi.org/10.1145/3306307.3328181

Li, W., & Kiran, R. (1996). An object-oriented design and implementation of reusable graph objects with C++. *Proceedings of the 1996 ACM Symposium on Applied Computing - SAC '96*. doi:https://doi.org/10.1145/331119.331433

Microsoft. (2021, September 9). *C and C++ in Visual Studio*. Retrieved from Microsoft Documentation: https://docs.microsoft.com/en-us/cpp/overview/visual-cpp-in-visual-studio?view=msvc-170

Nieto, J., Banks, C., & Chan, R. (2018). Abstracting rigging concepts for a future proof framework design. *Proceedings of the 8th Annual Digital Production Symposium*. doi:https://doi.org/10.1145/3233085.3233088

Radovan, M., & Pretorius, L. (2006, October). Facial animation in a nutshell: past, present and future. *Proceedings of the 2006 annual research conference of the South African institute* *of computer scientists and information technologists on IT research in developing countries*, 71-.

SideFX. (2022). Houdini. *[Computer Software]*. Retrieved from https://www.sidefx.com/products/houdini/

Stroustrup, B. (2021, May 13). *C++ Applications*. Retrieved from Stroustrup: https://www.stroustrup.com/applications.html

Terdiman, D. (2015, January 15). *And the Oscar for Best Visual Effects goes to...Autodesk's Maya*. Retrieved from VentureBeat: https://venturebeat.com/2015/01/15/hollywood-fx-pros-i-want-to-be-an-oscars-maya-winner/

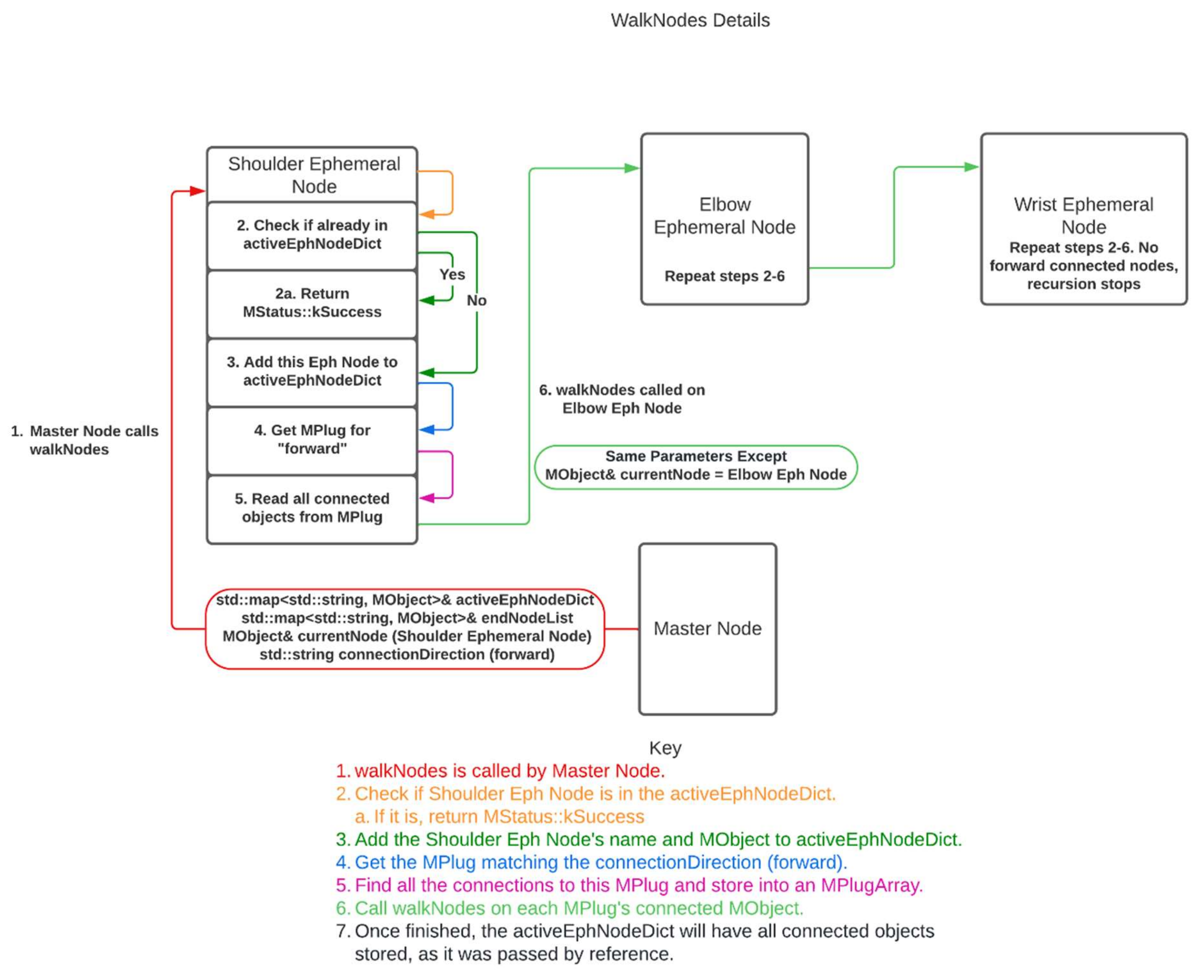
The Foundry. (2022). Nuke. *[Computer Software]*. Retrieved from https://www.foundry.com/products/nuke-family/nuke

Vertex Animation. (2021, September 11). Retrieved from Valve Developer Community: https://developer.valvesoftware.com/wiki/Vertex_animaion

Zehra, F., Javed, M., Khan, D., & Pasha, M. (2020). Comparative analysis of C++ and python in terms of memory and Time. *NED University of Engineering and Technology*. doi:https://doi.org/10.20944/preprints202012.0516.v1

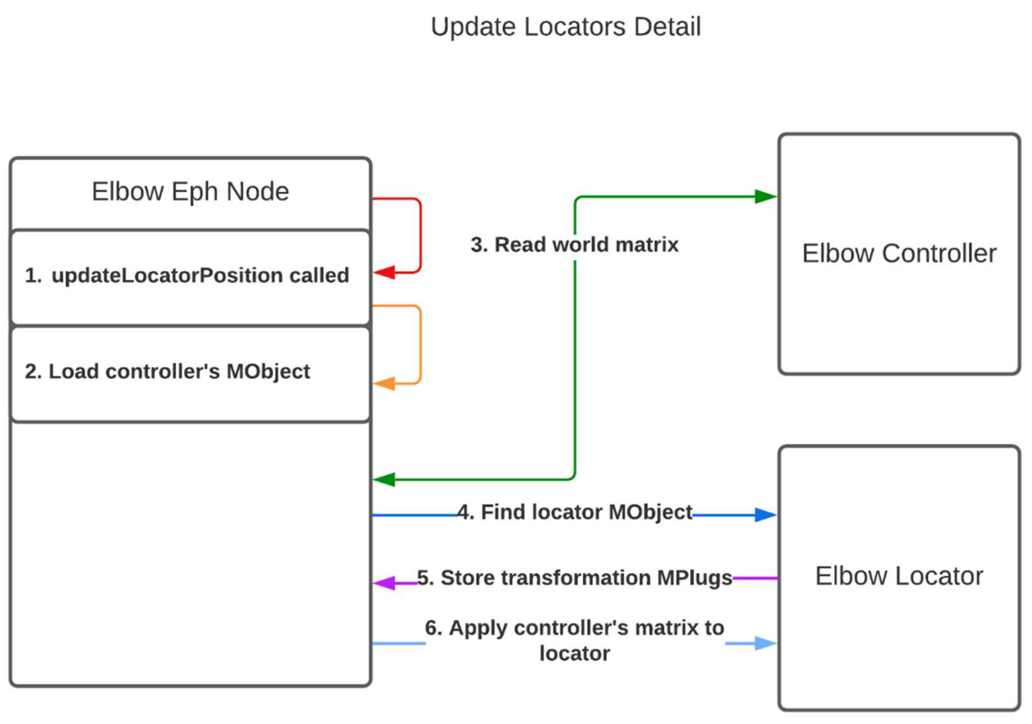
Appendices

Appendix A - Design Diagrams


WalkNodes Details
Shoulder Ephemeral Node
2. Check if already in activeEphNodeDict
Yes
No
2a. Return MStatus::kSuccess
3. Add this Eph Node to activeEphNodeDict
4. Get MPlug for "forward"
5. Read all connected objects from MPlug
1. Master Node calls walkNodes
Elbow Ephemeral Node
Repeat steps 2-6
Wrist Ephemeral Node
Repeat steps 2-6. No forward connected nodes, recursion stops
6. walkNodes called on Elbow Eph Node
Same Parameters Except MObject& currentNode = Elbow Eph Node
std::map<std::string, MObject>& activeEphNodeDict
std::map<std::string, MObject>& endNodeList
MObject& currentNode (Shoulder Ephemeral Node)
std::string connectionDirection (forward)
Master Node
Key

1. walkNodes is called by Master Node.
2. Check if Shoulder Eph Node is in the activeEphNodeDict.
   a. If it is, return MStatus::kSuccess
3. Add the Shoulder Eph Node's name and MObject to activeEphNodeDict.
4. Get the MPlug matching the connectionDirection (forward).
5. Find all the connections to this MPlug and store into an MPlugArray.
6. Call walkNodes on each MPlug's connected MObject.
7. Once finished, the activeEphNodeDict will have all connected objects stored, as it was passed by reference.

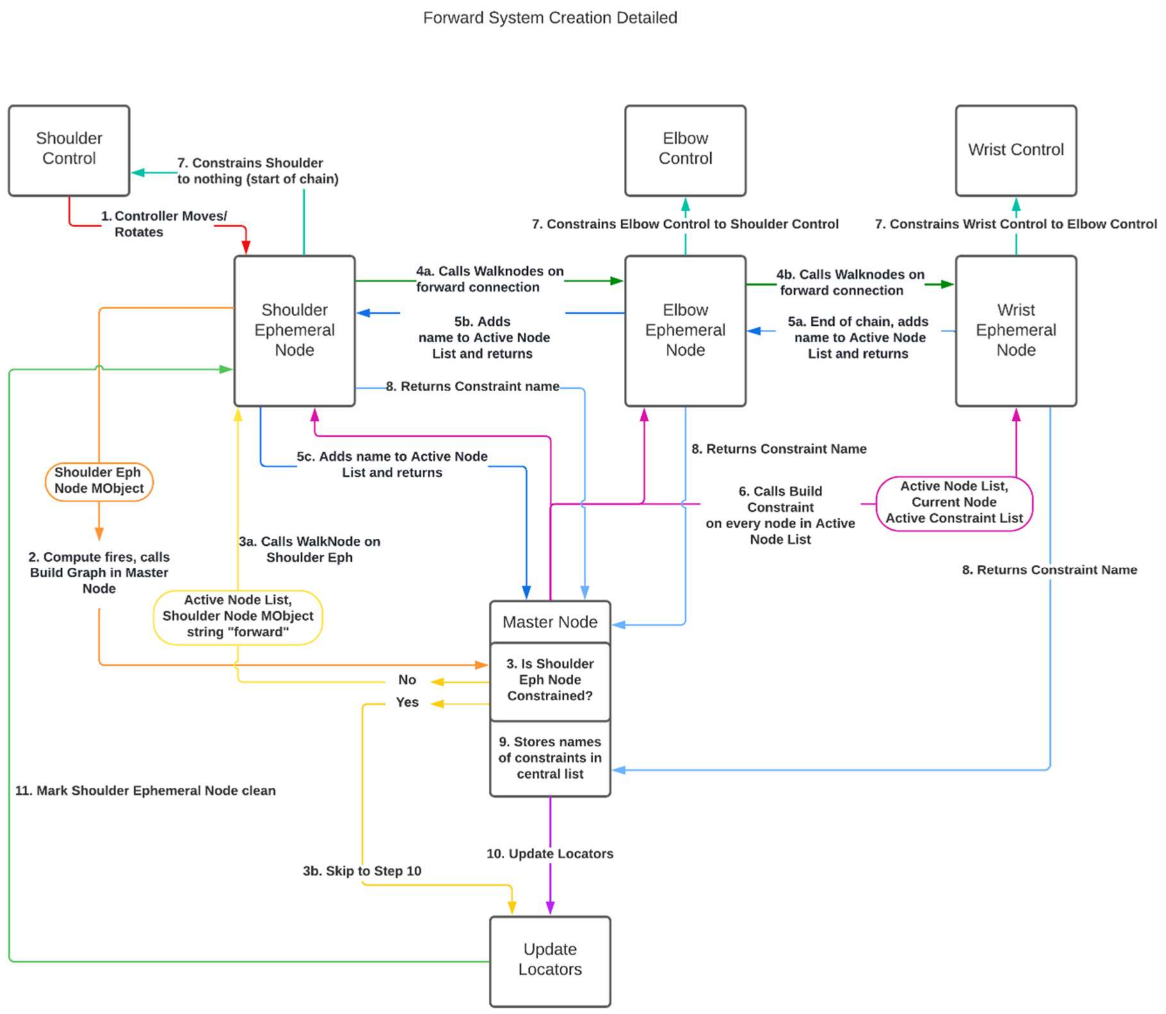
Figure A1: Walk Nodes Details


Update Locators Detail
Elbow Eph Node
1. updateLocatorPosition called
2. Load controller's MObject
3. Read world matrix
Elbow Controller
4. Find locator MObject
5. Store transformation MPlugs
6. Apply controller's matrix to locator
Elbow Locator

Key

1. updateLocatorPosition called.
2. Using incoming transform MPlug, load elbow controller's MObject.
3. Store the world matrix of the controller.
4. Get MPlug to locatorName attribute stored in Elbow Eph Node and use it to find the elbow locator MObject.
5. Store the 9 transformation MPlugs in a std::list<MPlug>.
6. Decompose the controller's matrix and apply the values to the locator through the MPlugs.
7. The locator's postion now matches the controller's.

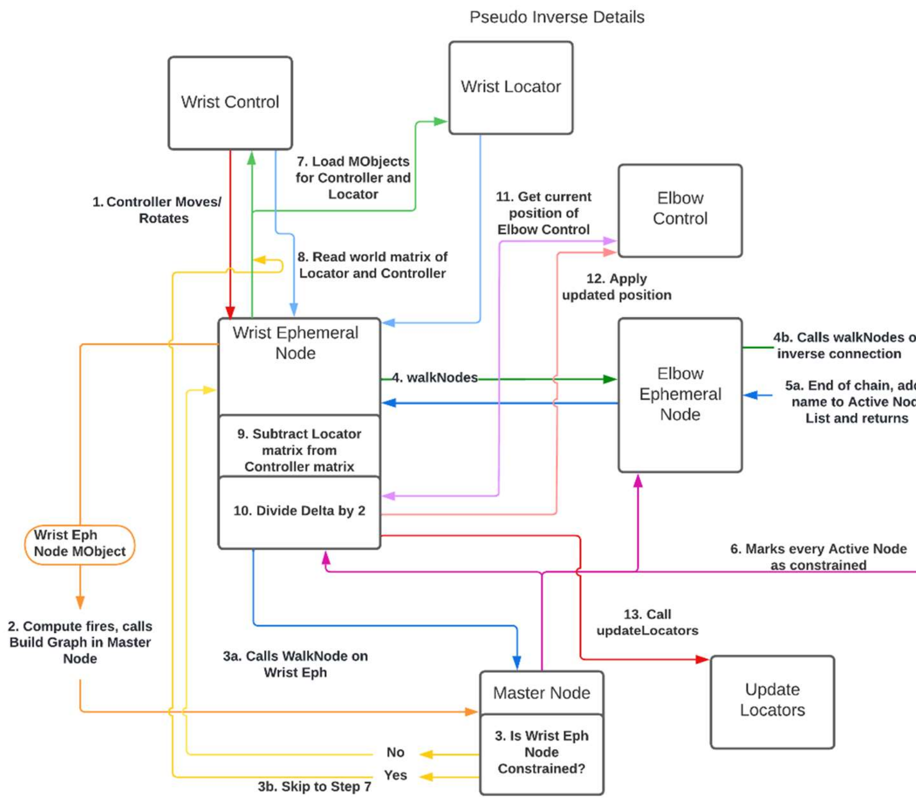
Figure A2: Update Locators Details


Forward System Creation Detailed
Shoulder Control
Elbow Control
Wrist Control
7. Constrains Shoulder to nothing (start of chain)
1. Controller Moves/ Rotates
7. Constrains Elbow Control to Shoulder Control
7. Constrains Wrist Control to Elbow Control
Shoulder Ephemeral Node
Elbow Ephemeral Node
Wrist Ephemeral Node
4a. Calls Walknodes on forward connection
4b. Calls Walknodes on forward connection
5b. Adds name to Active Node List and returns
5a. End of chain, adds name to Active Node List and returns
8. Returns Constraint name
8. Returns Constraint Name
5c. Adds name to Active Node List and returns
Shoulder Eph Node MObject
6. Calls Build Constraint on every node in Active Node List
Active Node List, Current Node Active Constraint List
2. Compute fires, calls Build Graph in Master Node
3a. Calls WalkNode on Shoulder Eph
8. Returns Constraint Name
Active Node List, Shoulder Node MObject string "forward"
Master Node
3. Is Shoulder Eph Node Constrained?
No
Yes
9. Stores names of constraints in central list
11. Mark Shoulder Ephemeral Node clean
10. Update Locators
3b. Skip to Step 10
Update Locators

Key:

1. User moves/rotates the shoulder controller.
2. Shoulder Ephemeral (Eph) Node fires the compute function, which calls Build Graph in Master Node.
3. Check if Shoulder Eph node is already constrained.
   a. If no, Master Node calls walkNodes on Shoulder Eph Node to get the names of all downstream nodes from the Shoulder node.
   b. If yes, skip to step 10.
4. Shoulder Eph recursively calls walkNodes down the chain.
   a. Shoulder calls Elbow.
   b. Elbow calls Wrist.
5. Once at the end (Wrist) Eph node, function walkNodes adds the node's name to the Active Node List and returns. Returns up the chain, adding the node names to the list.
   a. Adds "wrist" to Active Node List.
   b. Adds "elbow" to Active Node List.
   c. Adds "shoulder" to Active Node List.
6. Master Node calls to Build Constraint on every node in the Active Node List.
7. Each Eph Node creates a constraint from its controller to its parent controller and marks self as constrained.
8. Each Eph Node returns the new constraint name to the Master Node.
9. The Master Node stores each constraint name in a list.
10. Update each Eph Node's locators to match the controller's positions.
11. Mark Shoulder Ephemeral Node clean

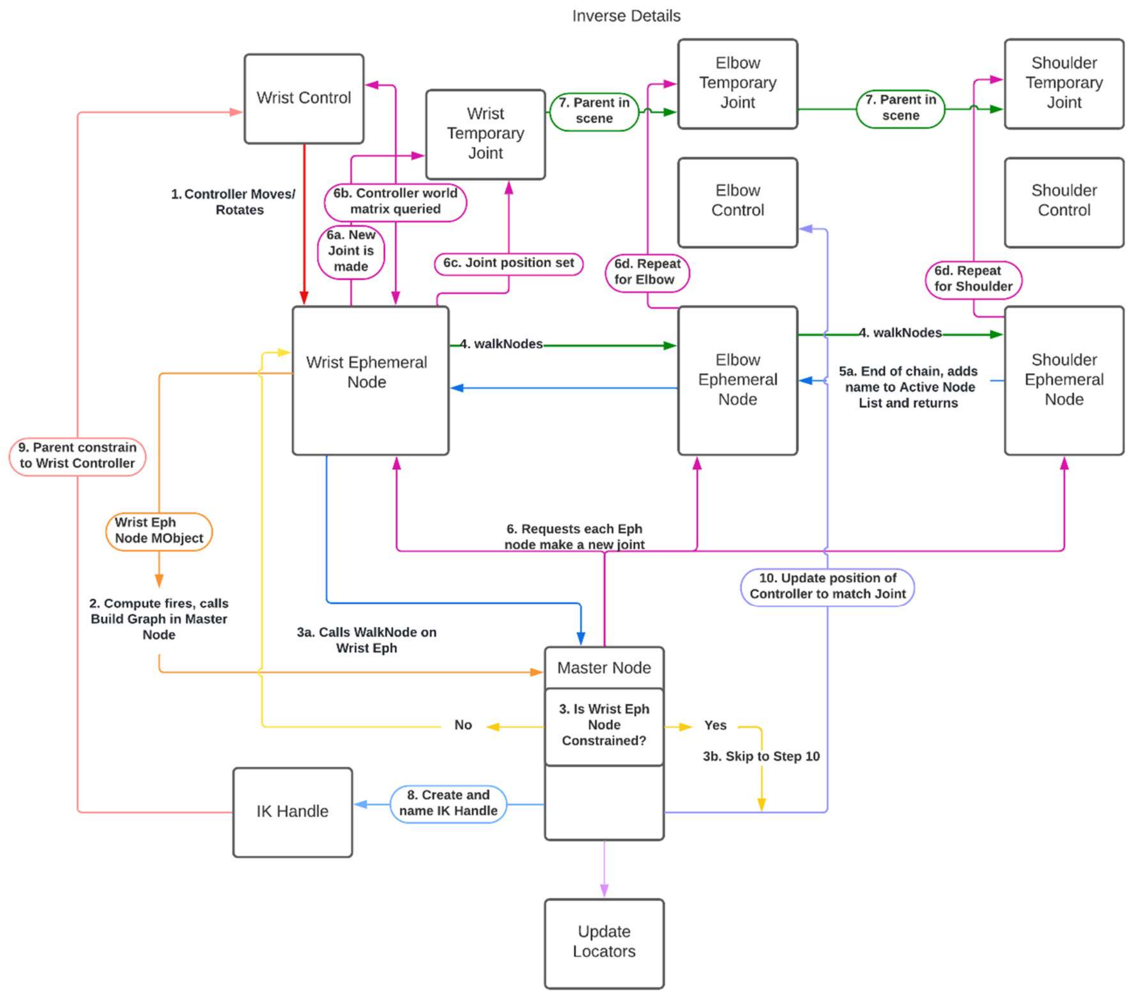
Figure A3: Forward Mode Details


Pseudo Inverse Details
Wrist Control
Wrist Locator
Elbow Control
Shoulder Control
1. Controller Moves/ Rotates
7. Load MObjects for Controller and Locator
8. Read world matrix of Locator and Controller
11. Get current position of Elbow Control
12. Apply updated position
Wrist Ephemeral Node
4. walkNodes
9. Subtract Locator matrix from Controller matrix
10. Divide Delta by 2
Elbow Ephemeral Node
4b. Calls walkNodes on inverse connection
5a. End of chain, adds name to Active Node List and returns
Shoulder Ephemeral Node
Wrist Eph Node MObject
6. Marks every Active Node as constrained
2. Compute fires, calls Build Graph in Master Node
3a. Calls WalkNode on Wrist Eph
13. Call updateLocators
Master Node
3. Is Wrist Eph Node Constrained?
No
Yes
3b. Skip to Step 7
Update Locators

Key:

1. User moves/rotates the shoulder controller.
2. Wrist Ephemeral (Eph) Node fires the compute function, which calls Build Graph in Master Node.
3. Check if Wrist Eph node is already constrained.
   a. If no, Master Node calls walkNodes on Wrist Eph Node to get the names of all upstream nodes from the Wrist node.
   b. If yes, skip to step 7.
4. Wrist Eph recursively calls walkNodes up the chain.
   a. Wrist calls Elbow.
   b. Elbow calls Shoulder.
5. Once at the end (Shoulder) Eph node, function walkNodes adds the node's name to the Active Node List and returns. Returns up the chain, adding the node names to the list.
   a. Adds "shoulder" to Active Node List.
   b. Adds "elbow" to Active Node List.
   c. Adds "wrist" to Active Node List.
6. Master Node marks all nodes in Active Node List as constrained.
7. Find MObjects of both the Wrist Controller and Wrist Locator.
8. Store the world space of both the Wrist Controller and Wrist Locator.
9. Subtract Wrist Locator position from Wrist Controller position to get the change in position.
10. Divide the change in position by number of Middle Nodes + 1 (in this case 2 for the Elbow node).
11. Get the current position of the Elbow Node's controller.
12. Add the divided delta to it and apply the new values to the controller.
13. Call updateLocators on the Elbow Node and Wrist Node.

Figure A4: Pseudo Inverse Details



Key:

1. User moves/rotates the shoulder controller.
2. Wrist Ephemeral (Eph) Node fires the compute function, which calls Build Graph in Master Node.
3. Check if Wrist Eph node is already constrained.
   a. If no, Master Node calls walkNodes on Wrist Eph Node to get the names of all upstream nodes from the Wrist node.
   b. If yes, skip to step 7.
4. Wrist Eph recursively calls walkNodes up the chain.
   a. Wrist calls Elbow.
   b. Elbow calls Shoulder.
5. Once at the end (Shoulder) Eph node, function walkNodes adds the node's name to the Active Node List and returns. Returns up the chain, adding the node names to the list.
   a. Adds "shoulder" to Active Node List.
   b. Adds "elbow" to Active Node List.
   c. Adds "wrist" to Active Node List.
6. Each node makes and places a new joint
   a. A new Wrist Temporary Joint is created and named
   b. The Wrist Controller is queried through the MPlug and the world matrix is read.
   c. Using the transformation MPlugs, the Wrist Joint is matched to the Wrist Controller's position.
   d. Repeats this process for both the Elbow and the Shoulder
7. Joints are parented in the scene hierarchy using their creation order.
8. The Master Node creates an IK handle and names it.
9. The IK Handle is parent constrained to the elbow controller.
10. The Elbow controller is matched to the position of the Elbow Temporary Joint.
11. The locators are updated.
12. The Wrist Ephemeral Node is set as clean.

Figure A5: Inverse Mode Details

Appendix B - Illustrative Code Snippets

```
MStatus ephemeralNode::walkNodes(std::map<std::string, MObject>& graphDict,
                                    std::map<std::string, MObject>& locatorList,
                                    std::map<std::string, MObject>& endNodeList,
                                   MObject& currentNodeMObject, std::string
                                    connectionDirection) {
    MGlobal::displayInfo("walkNodes called!");

    MPlugArray plugArray;
    MPlug inputJointPlug;

    MFnDependencyNode currentNodeMDep(currentNodeMObject);

    //check to make sure this node isn't in the graphDict
    if (graphDict.count(currentNodeMDep.name().asChar()) != 0) {
        return MStatus::kSuccess;
    }

    //add this node to the graphDict
    graphDict.insert({ currentNodeMDep.name().asChar(), currentNodeMObject });

    //Get and store the locator
    MObject currentLocatorMObject = getLocatorMObject(currentNodeMObject);
    MFnDependencyNode locatorMDep(currentLocatorMObject);
    std::string locatorName = locatorMDep.name().asChar();
    locatorList.insert({ locatorName, currentLocatorMObject });

    //Get all nodes connected to whatever direction looking for (forward/inverse)
    if (connectionDirection == "Forward") {
        inputJointPlug = currentNodeMDep.findPlug("forwardConnection", true);//Gets the plug
                                                                      for this attribute
    }
    else if (connectionDirection == "Inverse") {
        inputJointPlug = currentNodeMDep.findPlug("inverseConnection", true);//Gets the plug
                                                                      for this attribute
    }

    inputJointPlug.connectedTo(plugArray, true, true);//Gets a list of all connections to this
                                                    plug

    //If there are no more nodes down this branch, store this node as an end node
    if (plugArray.length() == 0) {
        endNodeList.insert({ currentNodeMDep.name().asChar(), currentNodeMObject });
        MGlobal::displayInfo("End node:");
        MGlobal::displayInfo(currentNodeMDep.name());
    }

    //call walkNodes for every node in the list
    for (MPlug plug : plugArray) {
        MObject connectedNodeMObject = plug.node();
        ephemeralNode* currentNodePointer = (ephemeralNode*)&connectedNodeMObject;
        currentNodePointer->walkNodes(graphDict, locatorList, endNodeList,
              connectedNodeMObject, connectionDirection);
    }

    return MStatus::kSuccess;
}
```

Figure B1: Walk Nodes Code

```
MStatus ephemeralNode::compute(const MPlug& plug, MDataBlock& data) {

    //Check if the plug is one of the active plugs
    if (plug != outputTransX && plug != outputTransY && plug != OutputTrans && plug != 
outputTransZ && plug != EphDependent &&
        plug != outputRotX && plug != outputRotY && plug != outputRotZ && plug != OutputRot) {
        return MS::kUnknownParameter;
    }

    //Add the callback © the joint positions on next idle step
    MObject thisMObject = this->thisMObject();
    MFnDependencyNode thisNodeMDep(thisMObject);
    std::string nodeName = thisNodeMDep.name().asChar();
    dirtyLocatorOnIdleCallbackID = MEventMessage::addEventCallback("idle",
                                 dirtyLocatorOnIdleCallback, (void*)&thisMObject);
    locatorCallbackVector.push_back(dirtyLocatorOnIdleCallbackID);

    if (plug == outputRotX || plug == outputRotY || plug == outputRotZ || plug == OutputRot ||
        plug == outputTransX || plug == outputTransY || plug == outputTransZ || plug == 
       OutputTrans) {

        //Sets the master node pointer
        if (this->masterNodePointer == NULL) {
            setMasterNodePointer(this->masterNodePointer);
            MGlobal::displayInfo(masterNodePointer->internalString);
            //Add this object to the list of all ephNodes in scene on the master node
            MObject currentNodeMObject = this->thisMObject();
            MFnDependencyNode currentNodeMDep(currentNodeMObject);
            masterNodePointer->ephNodeDict.insert({currentNodeMDep.name().asChar(),
                                                   currentNodeMObject});
        };

        //Call the build node graph on the master node
        MObject currentNodeMObject = this->thisMObject();
        masterNodePointer->buildNodeGraphFromConnections(currentNodeMObject,
                                                         masterNodeMObject);

        //If in inverse mode, update the parent controllers
        MDataHandle contraintModeString = data.inputValue(constraintMode);
        std::string constrainMode = contraintModeString.asString().asChar();
        if (constrainMode == "PseudoInverse" || constrainMode == "Inverse") {
            masterNodePointer->midNodeUpdateHelper(currentNodeMObject, constrainMode,
                                                   masterNodePointer);
        }

        //Update locator positions
        updateLocatorPosition(this->thisMObject(), "locator");

        MObject thisNode = plug.node();//Get this node
        MFnDependencyNode nodeFn(thisNode);//Convert to dependency node
        MString nodeName = nodeFn.name();//Get the name
        MGlobal::executeCommand("dgdirty -c " + nodeName);
    }

    data.setClean(plug);
    return MS::kSuccess;
}
```

Figure B2: Ephemeral Node Compute

```
MStatus masterNode::buildNodeGraphFromConnections(MObject& startNodeMObject, MObject&
thisMasterNodeMObject) {

    //If the start node is already constrained, exit
    MFnDependencyNode startNodeMDep(startNodeMObject);
    std::string startNodeName = startNodeMDep.name().asChar();
    MPlug isConstrainedPlug = startNodeMDep.findPlug("isConstrained", true);
    if (isConstrainedPlug.asBool() == true) {
        return MStatus::kSuccess;
    }

    std::vector<std::pair<std::string, MObject>> jointVector;

    ephemeralNode* startNodePointer = (ephemeralNode*)&startNodeMObject;

    if (!checkObjectInScene(thisMasterNodeMObject, "*masterNode*")) {
        MGlobal::displayError("No master node in scene!");
        return MStatus::kFailure;
    }

    //Make the dependency node
    MFnDependencyNode masterNodeMDep(thisMasterNodeMObject);

    //Get constraint mode
    MPlug constraintMode = masterNodeMDep.findPlug("constraintMode", false);
    std::string constraintModeString = constraintMode.asString().asChar();

    //Check if mode is forward
    if (constraintModeString == "Forward") {
        //call walkNodes on the startNode - store result into list (graphDict)
        startNodePointer->walkNodes(activeEphNodeDict, activeLocatorNodeDict,
                                   activeEndNodeDict, startNodeMObject, "Forward");

        for (auto pair : activeEphNodeDict) {
            //Check if the object is the selected one. If it is, don't need to constraint
              it to the parent
            ephemeralNode* currentNodePointer = (ephemeralNode*)&pair.second;

            currentNodePointer->constrainByGraph(activeEphNodeDict, pair.second,
                                                  startNodeMObject, activeConstraintList);
        }
    }

    else if (constraintModeString == "Inverse") {
        //call walkNodes on the startNode - store result into list (graphDict)
        startNodePointer->walkNodes(activeEphNodeDict, activeLocatorNodeDict,
                                  activeEndNodeDict, startNodeMObject, "Inverse");

        //Going to want to create joints in a similar manner to constrainByGraph
        startNodePointer->createJoints(activeEphNodeDict, jointVector,
                                         activeEndNodeDict.begin()->second,
                                         startNodeMObject, activeConstraintList);

        activeJointVector = jointVector;
        //There needs to be more than 1 joint to create the handle
        if (jointVector.size() > 1) {
            //Now there is a chain of all the joints - need to make IK handle
            std::string ikHandleName = startNodeName + "_ikh";
            std::string commandString = "ikHandle -n " + ikHandleName + " - sj " +
                                        jointVector.front().first + " - ee " +
                                        jointVector.back().first;
            MGlobal::executeCommand(commandString.c_str());

            //Constrain IK handle to wrist controller
            startNodePointer->constrainIKHandle(startNodeMObject, activeConstraintList,
                                                ikHandleName);
            //Point constrain to controller
            //Have to connect rotate channels to handle inputs
            activeIKHandleName = ikHandleName;

            MObject inputControllerMObject = startNodePointer->
                     getConnectedNodeMObject(startNodeMObject, "inputTranslateX", false);
            MFnDependencyNode inputControllerMDep(inputControllerMObject);//Creates a
                                        dependency node for getting the name

            //Since we are changing selection, we need to prevent the destruction that
              gets called
            selectionDummyVect.push_back(1);
            selectionDummyVect.push_back(1);
            MGlobal::selectByName(inputControllerMDep.name(),
                                  MGlobal::ListAdjustment::kReplaceList);
        }
        //Since no constraint is made if the chain is too short, a dummy one is created
        else {
            activeConstraintList.push_back("dummy");
        }
    }

    else if (constraintModeString == "PseudoInverse") {
        //call walkNodes on the startNode - store result into list (graphDict)
        startNodePointer->walkNodes(activeEphNodeDict, activeLocatorNodeDict,
                                    activeEndNodeDict, startNodeMObject, "Inverse");
        //Set these nodes as constrained
        for (auto node : activeEphNodeDict) {
            MFnDependencyNode MDep(node.second);
            MPlug isConstrainedPlug = MDep.findPlug("isConstrained", true);
            isConstrainedPlug.setBool(true);
            //The constriant list can't be empty, but this doesn't make an in scene
constraint
            activeConstraintList.push_back("dummy");
        }

    }

    //Suspended mode
    else {
        //Set the node as constrained
        MPlug isConstrainedPlug = startNodeMDep.findPlug("isConstrained", true);
        isConstrainedPlug.setBool(true);

        activeEphNodeDict.insert({ startNodeMDep.name().asChar(), startNodeMObject });

        //get locator MObject and MFnDep
        MObject currentNodeLocatorMObject = startNodePointer->
                                            getLocatorMObject(startNodeMObject);
        MFnDependencyNode currentNodeLocatorMDep(currentNodeLocatorMObject);
        //store into locator list so it updates
        activeLocatorNodeDict.insert({currentNodeLocatorMDep.name().asChar(),
                                      currentNodeLocatorMObject});

        //Add dummy to list so there's something in it for destruction
        activeConstraintList.push_back("dummy");
    }

    MGlobal::displayInfo("All constraints:");
    for (std::string constraint : activeConstraintList) {
        MGlobal::displayInfo(constraint.c_str());
    }

    MGlobal::displayInfo("All nodes in chain:");
    for (auto pair : activeEphNodeDict) {
        MGlobal::displayInfo(pair.first.c_str());
    }

    return MStatus::kSuccess;
}
```

Figure B3: Master Node System Constructor

```
void masterNode::removeNodeGraphConnections(void* data) {
    MGlobal::displayInfo("callback fired!");

    //If there are no active constraints, no need to do anything
    if (activeConstraintList.empty()) {
        return;
    }

    else if (selectionDummyVect.size() != 0) {
        selectionDummyVect.pop_back();
        return;
    }

    MPlug destructPlug;
    destructPlug.setMObject(disableDestruction);
    bool test = destructPlug.asBool();
    if (destructPlug.asBool() == true) {
        return;
    }

    //If selecting an eph or master node, don't delete anything
    MSelectionList list;
    MObject currentSelectionMObject;

    MGlobal::getActiveSelectionList(list);//copy the selection list into temp list
    list.getDependNode(0, currentSelectionMObject);//Get this node path from the new
                                                    entry in list

    MFnDependencyNode currentSelectionMDep(currentSelectionMObject);//Get the dependancy
                                                                    node for this node

    //If selected an eph node or the master node, don't destroy graph. Good for
                                debugging.
    if (currentSelectionMDep.typeId() == 0x00000001 ||
       currentSelectionMDep.typeId() == 0x00000002) {
        return;
    }

    ConstraintFactory constrFactory;

    //for every constraint in constraint list
    //delete the constraints
    for (auto constraint : activeConstraintList) {
        constrFactory.deleteExistingConstraint(constraint);
    };

    activeConstraintList.clear();

    //Delete just the root joint - everything related will also destroy
    if (!activeJointVector.empty()) {
        std::string commandString = "delete " + activeJointVector[0].first;
        MGlobal::executeCommand(commandString.c_str());
    }

    activeJointVector.clear();

    //If there is an IKHandle, delete the name
    if (activeIKHandleName != "") {
        activeIKHandleName = "";
    }

    //for every node in the list of active constraint node
    //mark as not constrained
    for (auto activeEphNodePair : activeEphNodeDict) {
        //Get the MDep node
        MFnDependencyNode activeEphNodeMDep(activeEphNodePair.second);
        //check if object still exists in scene (can be lost during scene shutdown)
        MSelectionList tempList;
        MObject tempMObject;
        tempList.add(activeEphNodePair.first.c_str());
        tempList.getDependNode(0, tempMObject);
        if (checkObjectInScene(activeEphNodePair.second,
                               activeEphNodePair.first.c_str())) {
            MPlug isConstrainedPlug = activeEphNodeMDep.findPlug("isConstrained", true);
            isConstrainedPlug.setBool(false);

            //set the ephemeral node to clean
            MString cleanString = ("dgdirty - c " + activeEphNodePair.first).c_str();
            MGlobal::executeCommand(cleanString);
        }
    }

    //Clear the rest of the dictionaries
    activeEphNodeDict.clear();
    activeEndNodeDict.clear();
    activeLocatorNodeDict.clear();
}
```

Figure B4: Master Node Graph Destructor