Dissertation Type: Enterprise

DEPARTMENT OF COMPUTER SCIENCE

Automating Locomotion Animation in Maya using a Deep Neural Network

Eleanor Cox

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Sunday 19th May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Eleanor Cox, Sunday 19th May, 2019

Contents

Executive Summary

The automation of animation processes is something greatly desired in industry as a way to save time, money and resources. 3D animation is booming, with applications in a wide range of industries. This project looks at the particular task of automating bipedal humanoid character locomotion. Locomotion refers to the act of moving from one place to another, and covers a range of motions such as walking and jogging.

During this thesis we investigate the state-of-the-art in animation automation and choose a suitable neural network from a recent research paper, which we then integrate with the animation software Autodesk Maya. This project hence bridges both the gaps between research and industry and between computer graphics and deep learning.

The primary contribution of this project is a set of scripts which implement a practical and efficient solution to the problem of character locomotion in Maya. Within Maya, a user can select a character and path for the character to walk along, which will subsequently be animated using data from the neural network from the aforementioned research paper. This animation covers a wide range of motion, adapting automatically to different terrains to a high degree of quality. The neural network has been adapted from its original presentation within the research paper to operate on the cloud to provide flexibility and a high degree of uptime. The project has been designed with the intention of being a cloud-based business, with users of Maya communicating with a cloud-hosted neural network, with the necessary data to produce an animation locally communicated back.

During this project, I:

- Investigated a state-of-the art research solution in automated character locomotion.
- Adapted the research solution to a new domain.
- Wrote a set of bespoke tools for manipulating, interacting with and extracting data from the animation software Maya.
- Learned about computational geometry and geometric algorithms, applying this knowledge to improve the implementation of the solution.
- Learned about cloud computing enabling the implementation of the solution to be portable and scalable.
- Tested my implementation in a large variety of settings, affirming its viability as a efficient and high-quality solution to the problem of automated character locomotion.

To see a full demonstration of the designed system please see video `0_full_demo` at the link https://drive.google.com/drive/folders/1fxyfG2KJZksDpBAxe8l9u_PKlUjSoW0b?usp=sharing

Supporting Technologies

The supporting technologies used in this project were as follows:

- I used Autodesk Maya 2018 [1] for the majority of my development
- I adapted the demo code for the Phase-Functioned Neural Network (PFNN) [16], available at `https://github.com/sreyafrancis/PFNN` [12]. I used the network weights from this repo and adapted the code moderately to suit my implementation of the PFNN.
- Languages: Python 2.7 [28], C++ 11 [18]
- Python: I used the Maya Python API 2.0, discussed in Section 2.2.3; `maya.cmds`, a wrapper for the Maya Commands interface, discussed in Section 2.2.3; and the Python standard libraries `json` and `socket` for networking
- C++ : I used the `Eigen3` and `glm` libraries, and the `socket` library and `nlohmann_json` header file [25] for networking
- I used Google Compute Platform for cloud-based hosting of the PFNN

Acknowledgements

I would like to take the time to thank my supervisor Dr Neill Campbell for his help throughout the project. I would also like to thank my friends and family for always being supportive of me, with special thanks to Alessio Zakaria for providing me with many insightful conversations.

Chapter 1

Contextual Background

1.1 Project Topic and Motivation

3D animation is being used increasingly across a number of industries, from film to television, to advertising and computer games. The increased demand for animation has bought with it a need for faster production pipelines. Traditionally, animation is hand-crafted by an animator, but even for the most skilled professionals this is a time-consuming process. The more complex and realistic an animation the harder this process becomes, and the overheads associated with it mean that prototyping an animation is not a trivial task, with most production houses taking great care to thoroughly plan an animation before even setting foot in the animation software they are using.

An obvious solution to this problem would be some form of automation, taking the grunt work out of animating and allowing an animator to focus on the details. This would be particularly useful for repetitive tasks such as animating a character walking. However, automation for artistic endeavours such as animation proves to be hard, with existing solutions often offering little variation in their output and not providing bespoke or high quality results. The current best commercially available solutions are in the form of AI-driven crowd simulators, discussed in more depth in Section 2.3.3, but these have their own problems such as complicated workflows, software-specific training and time-consuming character initialisation processes.

In this project we look specifically at the task of automating bipedal character locomotion, that is, humanoid characters moving from one place to another. There is the need for a simple, intuitive tool that allows an animator to select a character and a path, which then animates the character walking along this path. This task may seem easy but is far from trivial, and in Section 2.3 we discuss a number of existing approaches to the problem. The goal of this project is to create such a tool; an intuitive animation prototyping program that integrates simply with a pre-existing animation software and automates the process of locomotion animation, saving animators and production houses alike both time and money.

To complete this goal we have chosen to integrate a state-of-the-art neural network, designed to synthesise locomotion animations, with an incumbent animation software used consistently throughout the 3D animation world. These are, respectively, the Phase-Functioned Neural Network [16] and Autodesk Maya [1]. The Phase-Functioned Neural Network (PFNN) was chosen for it's superior animation results and it's ability to adapt the animation output to the geometry of the terrain under the character, which adds a great deal of expressiveness to the final animation. While on the cutting edge of research, the results of the PFNN appeared to have useful applications as a piece of commercially available software, and hence this project was developed with the idea that it could eventually be used in industry, either as a piece of standalone software or to eventually be integrated into one of the leading commercial animation solutions. Autodesk Maya, or simply Maya, was chosen due to it's leading position as an animation software for both amateurs and professionals, giving this project a large target market.

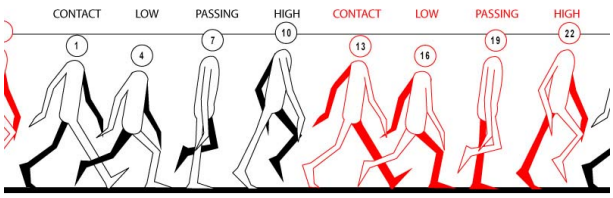


Figure 1.1: A basic walk cycle.

1.2 Animating Locomotion

Locomotion is movement, or the ability to move, from one place to another. In this section we look at two ways to animate locomotion.

1.2.1 Gaits

For any living creature, different types of locomotion can be referred to as *gaits*. Gaits are classified by a number of characteristics including purpose, footfall pattern, limb movement patterns and overall velocity [27]. Gaits are also characterised as either *natural* or *abnormal*. For human adults, the five natural gaits, in increasing order of speed, are walking, jogging, skipping, running and sprinting. Abnormal gaits may be caused by neurological or musculoskeletal disorders, such as a limping gait, or may be learned and practiced behaviour, such as hand-walking or crouch-walking.

1.2.2 Walk Cycle

We can think of a gait as describing a cyclic set of motions - taking a step with one foot, then a step with the other, and repeating these motions over and over. Following from this we can define the idea of a *walk cycle*, a fundamental animation technique wherein an animator defines a two-step sequence and then loops over these, allowing them to avoid animating each subsequent step explicitly.

A walk cycle is defined by eight key poses, as seen in Figure 1.1[1]. These represent important points in the animation, such as the point of contact with the floor, or the highest point of the step. In computer animation the frames between these poses, known as in-betweens, are interpolated by the animation software.

A walk cycle is a simple but effective method of animating character locomotion. If the goal is simple, repetitive motion then it makes sense to use this basic method - for an animator, it is both intuitive and quick to implement. However there are issues with this method. There is no variation between cycles so the animation becomes more noticeable, especially if multiple characters are using the same walking animation. This also means variations in speed are harder to animate. If the character is not walking in a straight line then extra care must be put in to correctly animate the rotation of the character around their centre, and the peripheral motions of the arms and legs may no longer look correct. Most importantly, the more detail and variety you want in your animation, the harder this method becomes to use, meaning that animating realistic human motion that could be used in feature films is almost impossible to do with this method.

1.2.3 Motion Capture

An improved solution to locomotion animation is through the use of motion capture technologies. Motion capture is a technique first used in the 1980's for capturing the movements of characters for video games. The motion capture process has significantly improved in the last 40 years, and now involves filming the movements of an actor and then using specialised computer vision software to take this video and track

[1] Image from Sherman High School, available at: https://bit.ly/2w1YGTW

the actor's position and movements. This data can then be retargeted to a 3D character model in an animation software to allow the computer-generated character to move according to the motions of the actor [22].

Motion capture offers several advantages over traditional animation methods. **Low latency** means results are captured in close to real-time, giving freedom to an actor to try different methods of acting a scene with minimal overheads. **Secondary motions** such as weight shifting or blinking are accurately captured without additional input, which in traditional animation requires both planning the motions themselves and a highly skilled animator to animate them. Motion capture is not just limited to locomotion, but **any physical motion** that an actor can make, such as throwing a ball. Motion capture is also particularly time and cost effective compared to other animation methods, although it is prohibitively expensive. Importantly for this thesis, motion capture has been used in clinical medicine to aid with gait analysis [7] so it is safe to assume that this is an appropriate technique for us to explore with regards to animating realistic and accurate locomotion.

With regards to animation the issues with motion capture arise with the data it produces. Motion capture data is often "messy" and has artifacts in it which, if applied immediately to a character model, can make the character's joints and bones jump around and results in jarring motion. Motion capture data must hence be cleaned before it can be used, a process which involves a professional animator checking an animation frame-by-frame and adjusting the character model when artifacts arise. This is incredibly time-consuming and infeasible to do with every character (especially background characters) on a large-scale project. As such, while direct motion capture for every character in a film is not an option for even the biggest-budget movies, a set of motion capture data could be used cleverly to *generate* animations for characters.

1.3 Phase-Functioned Neural Network

The Phase-Functioned Neural Network (PFNN) is a neural network architecture outlined in a 2017 paper, "Phase-Functioned Neural Networks for Character Control", published in *ACM Transactions on Graphics* [16]. The paper was published by Daniel Holden, a then-PhD student at the University of Edinburgh, and his colleagues Taku Komura and Jun Saito. The PFNN acts as a character controller for characters in interactive scenes such as video games and can be used to animate a character in locomotion in a variety of gaits. The network takes a frame of animation and some user input, for example from a games controller, and produces the next frame of animation. For an in-depth look at the architecture of the neural network see Section 2.4.

The PFNN was developed with the goal of creating a real-time, low memory and data-driven controller for virtual characters. Holden et al.'s work builds on a number of ideas from alternate neural network architectures. They identify issues specific to convolutional and regressive architectures, covering both functional and behavioural issues with the animation output as well as issues relating to time and memory constraints, an important factor for networks that require online learning such as those with potential use in video games. Such networks need to be able to produce animation quickly and with constantly updating input from the user, a problem thoroughly considered by Holden et al. when designing the PFNN. A discussion of alternative neural network architectures for locomotion synthesis can be found in Section 2.3.5.

Holden et al. also build upon an idea by Roland Memisevic who, in 2013, proposed a network where the latent variables directly parameterise the weights of the neural network [21]. The PFNN adopts this concept by taking the phase of the motion, a variable which represents how far through a walk cycle the character is, as a parameter to the network and, using this, calculating the weights of each layer in the network via a phase function. This means the network does not mix data from different phases and allows it to be more expressive whilst retaining a compact structure.

1.4 Aims

The high-level goal of this thesis is to create a system that allows an animator to automatically generate locomotion animations in Maya. This can be split into a number of smaller goals, which are:

- To investigate the state-of-the-art in automating animation. In Section 2.3 we detail a number of solutions to the problem of automating animation, and discuss the pros and cons of each method. In Section 2.4 we look in depth at one particular solution, the Phase-Functioned Neural Network, which we later use in our implementation.
- To integrate an existing solution into the animation software Maya. Our chosen solution, the Phase-Functioned Neural Network, was developed specifically for video games and published as a combined package of the neural network alongside an interactive demo. Adapting this to work in a purely animation setting is a non-trivial task requiring an in depth understanding of the technical implementation of the PFNN.
- To alter the implementation to better fit the target domain. By being specialised for video games, the PFNN is not suited for being ported directly to animation with various limitations and problems that need be solved in order for high-quality animations to be obtained. These problems will be explored and solutions implemented to fully exploit the novel and expressie power of the PFNN.

Chapter 2

Technical Background

This chapter covers the technical background needed to understand the implementation of this project described in Chapter 3. The first section gives a brief introduction to computer graphics, particularly the aspects of this field related to building up and animating a character. The next section covers Autodesk Maya, an animation software used ubiquitously throughout the film, animation and VFX industries. We then look at existing software solutions and research into the problem of locomotion animation, covering from the most basic techniques up to the state-of-the-art. After this we look in-depth at the Phase-Functioned Neural Network, gaining a thorough understanding of it's inputs, outputs and how it infers a frame of animation. We finally cover a set of mathematical formulae and concepts that will be used in the implementation of the project.

2.1 An Introduction to Computer Graphics

In this section we cover the aspects of computer graphics that relate to animating a character. We start with modelling, the process of creating a geometry for a character; we continue with rigging, the process of setting up a character for animation; then finally we look at the process of animating itself.

2.1.1 Modelling

Modelling is the process of developing a mathematical, wireframe representation of a three-dimensional object, known as a model. A model is a form of geometry and can be constructed in different ways. Two common construction methods are polygonal modelling and NURBS modelling, which we will look at in more detail now.

2.1.1.1 Polygons

Polygonal modelling is an approach to modelling objects by representing their surfaces using polygons. A polygon is defined by vertices and a line between two vertices is an edge. Three vertices connected by their edges form a triangle, the simplest Euclidean polygon. A polygon with four vertices is known as a quadilateral or *quad*, and with n vertices we have an n-sided polygon or *n-gon*. A group of polygons which are connected by shared vertices is known as a *mesh*, and each polygon in a mesh is called a face.

In polygonal modelling we most commonly use triangles or quads as our faces. Triangles are particularly useful as any triangle, defined by three points, also defines a plane in 3D space. We can thus easily compute information about the triangle, such as whether a point lies within it. A polygon face is *planar* when it's vertices lie in one plane, and hence triangles are always planar. Higher-dimensional polygons can form *non-planar* faces if one or more vertices do not lie in the same plane, which is undesirable as these are hard to represent and, for an animation software, hard to interpret correctly. Thus with a mesh we often go through a process called *triangulation* wherein we decompose a mesh into a set of triangles

to make it easier to use.

2.1.1.2 NURBS

Non-Uniform Rational B-Splines or NURBS are mathematical representations of curves and surfaces. NURBS provide a high level of control but can be less intuitive to think about than polygons. Due to their basis in analytical mathematics NURBS produce smooth, accurate geometries, and hence NURBS modelling has applications in wider scientific and engineering fields such as aerospace engineering. NURBS curves can be combined to create NURBS surfaces, the NURBS equivalent of a polygonal mesh.

We now define some properties of NURBS that are useful for us to know.

Control Vertices The control vertices of a curve determine it's shape. The most common method of creating a NURBS curve is to directly place control vertices into the scene. These determine how the curve is "pulled" from a straight line.

Spans and Edit Points Large or complex curves may need to be composed of multiple smaller curve sections known as spans. Edit points signify the positions at which these spans meet. There is also an edit point at the start and end of every curve.

Curvature The curvature of a curve signifies it's deviation from a straight line. The higher it's curvature the more curved a curve is.

Parameterisation Parameters are the unique numeric values of points on a curve or surface, akin to coordinates. Just as points in space have three dimensions, X, Y and Z, NURBS curves have a dimension U that describes positions along the length of the curve. Our parameter values are measured along U, where a higher parameter value signifies a point further along the curve.

There are multiple ways to parameterise a curve but we look specifically at uniform parameterisation. Uniform parameterisation assigns parameter values at each edit point and uniformly distributes parameters along the spans between edit points. Thus the first edit point will have parameter 0.0, the second will have parameter 1.0, the third 2.0 and so on. Hence the parameter value of the last edit point is equivalent to the number of spans in the curve. It is important to note that, with uniform parameterisation, the parameter values have no correlation with the length of the curve as edit points may be unevenly distributed throughout the curve.

2.1.2 Rigging

Rigging is the process of setting up a model to be animated. We can rig models of all types, from robotic machines to characters such as a human or a bear. In this thesis we cover humanoid characters only. A *rig* is a set of controls that can manipulate parts of the model. These controls can come in many forms, and we cover a few relevant ones below.

2.1.2.1 Skeletons

A skeleton is a hierarchical structure composed of *joints*. Joints are the building blocks of skeletons and are connected together in *joint chains*. The topmost joint in the hierarchy is known as the *root*, and we can think of joints as having *bones* between them. Bones do not have a physical or calculable presence but are a visual cue to illustrate the relationships between joints.

Skeletons provide a structure for animating a model in the same way that a human skeleton determines how a body can move. Rotating or translating a joint will affect the bones attached to it, and any children of the joint. For example, if we have a simple three-joint chain of a shoulder, elbow and wrist,

then rotating the shoulder will also move the elbow and wrist, whereas rotating the wrist will not affect the other joints. This is because the wrist is a *child* of the elbow, and hence the shoulder, but the shoulder and elbow and not children of the wrist.

2.1.2.2 Control Curves

Control curves are another control we can add to a character rig. Control curves add a layer of abstraction to the joints of a model by acting as a handle for a joint, such that the handle is transformed and the joint reacts appropriately. This allows an animator to animate a control curve instead of directly animating the joint itself, which provides a safety net in terms of accidentally manipulating the skeleton in undesirable ways that may cause issues further down the production pipeline.

2.1.2.3 Skinning

Skinning is the process of binding a mesh to a skeleton so that the geometry of the model can be deformed by the skeleton.

2.1.3 Animation

In the 1800's Eadweard Muybridge demonstrated that a sequence of slowly changing still images can create the illusion of motion [15]. This idea forms the basis of animation. In an animation each of these images is called a *frame* and we have a *frame rate*, expressed in in frames per second (fps), which signifies the frequency at which consecutive frames appear.

Keys are markers that specify the value of an object's attribute, such as it's translation or rotation, at a particular time. Setting keys or *keyframing* is the process of creating these markers to specify actions at certain times in an animation. If we recall Figure 1.1, our basic walk cycle, to create a walking animation we would specify keys at each of these poses. Animation is thus the process of transforming an object and keyframing it throughout time to give the illusion of motion. For the frames at which we have not set a key, known as in-betweens, the attributes of an object are calculated via interpolation of the values at the surrounding keyframes.

2.1.3.1 Kinematics

Kinematics describes the properties of motion in an object. When animating a skeleton we can work in either of two methods: Forward Kinematics and Inverse Kinematics.

Forward Kinematics Forward Kinematics (FK) describes the process of animating a joint chain in a top-down manner. That is, we transform the highest joint in the hierarchy, then it's child joint, then the next child and so forth. This provides very granular control of our animation but is both time-consuming and unintuitive when trying to animate goal-directed motion such as reaching for a doorknob. With FK animation we transform and animate the joints directly. Motion capture gives us FK data as it specifies the joint movements and rotations of every joint.

Inverse Kinematics Inverse Kinematics (IK) is a mathematical process that takes a desired end pose, specified by a position and orientation, and computes the joint angles required to reach this pose. This is obviously an ideal method for creating goal-directed motion. Inverse kinematics requires the use of additional controls knows as IK Handles and IK Solvers. An IK handle is like a puppet wire running through a joint chain, and the end of the handle signifies the goal position. As the handle moves the IK solver calculates how to rotate the joints within the joint chain.

Blending FK and IK A good character rig will enable you to use both IK and FK as this allows for both high-level and low-level control over movements. In our implementation we use forward kinematics, and leave a discussion on inverse kinematics to future work (Section 4.4.1).

2.2 Autodesk Maya

Autodesk Maya is one of the leading 3D modelling and animation packages and is used extensively throughout the visual effects, film and video game industries. Autodesk Maya, henceforth referred to as Maya, has applications in a wide range of computer graphics tasks from modelling, rigging and animating to rendering and performing physics simulations. Maya is an industry standard software and it's position as an incumbent is largely due to how powerful a tool it is. Maya was chosen for this project due to it's widespread use in both industry and amongst amateur artists and animators.

In this section we first define some common terminology used within Maya. We then explore the underlying graph architecture of Maya, known as the Dependency Graph. Next we discuss the various methods of programming in Maya, and finally we cover some of Maya's limitations that we will need to work around during the execution of the project.

2.2.1 Terminology

We define some common terminology used by Maya, which will be used throughout Chapter 3.

Scene In Maya our scene is the 3D workspace in which we create, transform and animate models.

Parenting Parenting is the process of establishing hierarchy by creating a parent-child relationship between two objects. When you make node B the child of node A, we say you have *parented* node B to node A[1]. The parent object is higher than the child in the hierarchy, and transformations on the parent object affect the child object, but transformations on the child do not affect the parent.

World space World space is the coordinate system for the "world", i.e. the entire scene. Its origin is at the centre of the scene, with the X and Z axes forming the horizontal plane, pointing towards the sides and front, and the Y axis forming the vertical, i.e. "up" in world space. The world space coordinate system axes are immutable.

Object space Object space is the coordinate system from an object's point of view. The origin is at the object's pivot point (a point that can be changed by the user, but by default is at the centre of the object) and its axes are rotated as the object rotates. If we have a matrix that transforms an object from world space to object space, then it's inverse matrix transforms the object from object space to world space.

Local space is similar to object space except it uses the origin and axes of a different object, where we describe object A as local to object B. In Maya local space always refers to an object being local to it's parent node. In the PFNN we consider some objects as local to each other, for example we consider joint positions local to the character's root transform.

2.2.2 The Dependency Graph

At it's core Maya is an efficient graph architecture where each element is referred to as a *node* and stores information pertaining to some part of the scene. An object, such as a sphere, is built from several nodes:

[1] Note that node B is parented to node A, but node A is the parent node. This is a quirk of the terminology and not a mistake.

- A *creation node* that records how the object was created
- A *transform node* that records how the object has been translated, rotated and scaled
- A *shape node* that holds a mathematical description of the object's shape

Note that these are just three of hundreds of types of node within Maya, each with a specialised purpose and use.

An *attribute* describes something about a node and controls how the node functions. As an example, an attribute in the transform node would be the object's translation in X. Attributes are what we key when we create an animation.

Nodes exist in an interconnected hierarchical graph structure and this is known as the *depedency graph*. The dependency graph can be thought of as a chain of nodes that can have cyclic connections. The entirity of a scene is stored within this graph and attributes from one object can connect to attributes of another, allowing data to flow through the graph. Each node in the dependency graph represents an action taken to build up the current scene from scratch with the final result being the scene in it's current state.

The dependency graph is implemented such that it enables the instantaneous update of a scene once a change to any part is made. If we change an attribute of an object the graph checks all children of the object to see if they too need to be updated then performs the appropriate operations. For example, if we rotate a shoulder joint, the graph knows to move the elbow and wrist joints through world space to the appropriate locations. This is performed in such a way that only the relevant parts of the scene are updated, for speed and to prevent uneccesary computation.

A special type of dependency graph node is a directed acyclic graph (DAG) node. Every DAG node consists of at minimum two further nodes, a transform and a shape node, where the shape node is always the child of the transform node. DAG nodes describe how to create an object from a piece of geometry. A path through the directed acyclic graph uniquely identifies the location of an instance of a particular node in the dependency graph, for example could provide us access to the nodes of a sphere we have created in our scene, which can then be used to obtain the shape and transformation information about that sphere.

We don't need to go into any more depth about the DAG hierarchy, just to be aware of it, as in our implementation we will traverse this graph to access an object's transform node.

2.2.3 Programming in Maya

Maya allows you to run custom code from within the product itself. There are two options here: scripts, which can be written and run from Maya's inbuilt script editor; and plug-ins, which are written and compiled externally then run in Maya. The choice between writing a script or a plug-in depends on the aim of your implementation; as per the Maya documentation [17], plug-ins are for extending the functionality of Maya, such as creating a new node or rendering pipeline, whereas scripts are for everything else.

There are two interfaces through which a programmer can code in Maya: Maya Commands and the Maya API. The former is an extensive custom library of commonly used commands within Maya, that covers everything from modelling to animating to rendering the Maya GUI. The latter allows low-level access to the dependency graph and node architecture of Maya, allowing for more powerful computations.

To use Maya Commands we must be programming in either the Maya Embedded Language (MEL) or Python. The Maya API can be accessed via Python, C++ or .NET. We discuss the advantages and disadvantages of each language below.

2.2.3.1 Maya Embedded Language

MEL is a scripting language descended from UNIX shell scripting. This means MEL is strongly based around executing commands as opposed to, say, manipulating data structures, calling functions or using

object oriented methods. MEL defines the Maya Commands interface which is powerful for many Maya-specific tasks, however as a language it lacks a lot of useful functionality, for example you cannot create an array within an array. MEL was deemed too limited to be useful for this project.

2.2.3.2 C++

The Maya API is written in C++ and provides better performance than either MEL, Python or the .NET API. This comes at the cost of not being able to run code internally within Maya's script editor, as C++ code must be compiled externally and ported as a plug-in. This in turn means there is no scripting available for C++ . Compiled C++ plug-ins also use different file extensions for different platforms which can be an annoyance, whereas Python plug-ins are platform independent. While C++ is a powerful language and especially useful for performance-critical tasks such as high resolution rendering, the lack of access to the Maya Commands interface meant that this was an unappealing choice.

2.2.3.3 Python

Maya comes packaged with Python 2.7.11 which allows access to both Maya Commands and the Maya API. Python is also a widely used language meaning there are a lot of available libraries that cover additional functionality that may be desired, such as networking. However there are some libraries that do not come shipped with Maya and cannot be imported into a script without expressly installing them - unfortunately this includes some very common and useful libraries such as `numpy` and `scipy`. This is a limitation but can be worked around as not using these libraries seems a better solution than forcing a user to install these packages in their version of Maya themselves, a surprisingly non-trivial task.

maya.cmds Python's `maya.cmds` is a Python wrapper for MEL commands which gives access to the native Maya commands. As with MEL, the Python Maya command interface allows the user to use high-level commands, which is useful for things such as creating a sphere with a single line of code.

Maya Python API 2.0 The Maya Python API, also known as `OpenMaya`, is a wrapper for the Maya C++ API, giving the programmer access to lower level commands and direct access to any node in the dependency graph. The Maya Python API 2.0 is the newer version of the API, which provides a more Pythonic workflow and improved performance upon version 1.0. The API provides access to areas of Maya which the Commands interface cannot interact with directly and hence it is useful to use both when developing, especially as both can be used within the same script. For this reason Python was chosen as the language for our Maya-side code, as the flexibility provided allows us to use both common functions and node-based methods.

2.2.3.4 .NET

The Maya .NET API is similar to the Maya Python API in that it is generated from the Maya C++ API. Most classes in the .NET API have a corresponding class in the C++ API with the same name and with a similar interface (that is, a similar set of functions and classes). .NET only allows the user to create plug-ins and didn't seem to have any benefits over the other languages so was not chosen as the language for this project.

2.2.4 Limitations

There are a few limitations in Maya that we will have to work around. Already discussed was the lack of Python libraries such as `numpy` which would provide a computationally smart way of processing data within our system. Another key limitation is the way Maya's geometries are defined, as vertices and polygons, meaning that it is impossible to directly sample a point on a mesh unless that point is at a vertex. This will be an issue when trying to sample positions on our mesh during our implementation, and the solution to this must be hand-crafted. This is covered in Section 3.3.3.

2.3 Existing Research and Software Solutions

In this section we look at existing solutions to the problem of animating locomotion, from the most basic to the state-of-the-art, covering both academic research and industrial software. We aim to compare the implementations and results of each solution and look for a gap between research and industry where we can produce an interesting solution.

2.3.1 Walk Cycle Scripting

There are a number of scripts available for Maya that automate the process of animating a basic walk cycle. These provide the lowest level of animation, with simple repeated walk cycles that work only on flat terrain and along a straight path. These scripts vary wildly in their implementation, with some keyframing the characters joints and some the control curves, and most offering FK animation but some offering IK. Two examples of such scripts are `AutoWalk for Maya` [24] and `walkTool` [10].

AutoWalk for Maya For each of a rig's control curves, AutoWalk for Maya requires the user to type in a name, an axis of rotation, an axis of movement and other attribute values, which it then uses to create a walk cycle. This is obviously a lot of input required from the user. The animation provided is very basic, lacking some of the complexities of a higher quality animation such as a sense of weight. The programmer himself, Nils Diefenbach, commented in the v0.0.3 release notes that the code has a lot of bugs and doesn't always work[2], and the script additionally only works for Maya 2012 and older.

WalkTool WalkTool features a collection of scripts that take you through the process of creating joints, connecting these in a skeleton, building IK handles and then animating the walk. WalkTool obviously offers a wider range of functionality than AutoWalk but does not offer much better in terms of animation quality. One benefit of WalkTool over AutoWalk is that it allows the user to adjust many suitable parameters of the walk, such as the step size, which allows it to offer more varied animation. Whilst more varied this animation is still not detailed enough for most animator's use. It is also interesting to note that the implementation of these script is, for want of a better word, very strange. As an example, to create the character's skeleton, instead of placing joints within the body the user is instructed to draw NURBS curves signifying the arms, legs and spine, which are then converted into joints and finally connected to form a skeleton - this is a very roundabout and inelegant way of achieving this task considering it is much easier for the user to place a joint accurately than to draw a curve accurately, and the intuition behind placing a joint is more clear than that behind drawing a curve to represent bones. This script is also only developed to work in Maya 2011, so is not appropriate for more modern needs.

In fact, it appears that most of the available walk cycle scripts were developed for Maya 2012 or earlier, suggesting that this method has long outlived it's usefulness.

2.3.2 Blending Pre-Made Animations

A common solution is to take a library of pre-made animations, known as clips, and blend these together to create a final animation. These pre-made animations can be either hand-crafted or based on cleaned motion capture data. Often these animations are crafted in a way such that they can be looped and hence the actual locomotion animations for this method tend to be rather close to basic walk cycles. The looping is done to ensure minimal memory usage is needed when downloading and using an animation clip.

The process of blending refers to interpolation between the final frames of one clip and the initial frames of the next. Consider a case where we want our character to walk for 100 frames, then run for 100 frames. Without blending the transition between these motions will be very apparent, as the character suddenly speeds up and changes gait within 1 frame. With blending the character may spend 50 frames transitioning between these two gaits, and so the animation will look more realistic.

[2] Comment available here: https://www.highend3d.com/maya/script/autowalk-for-maya

Once an animation has been decided upon by the user it is transferred to their character rig through a process known as *retargeting*. Retargeting takes the animation data from the first skeleton, such as the joint positions and rotations at each frame, and transfers these to the target skeleton. For characters with an identical skeleton structure this is a trivial process, but care must be put in when retargeting animation between characters with different skeletons as the results can be unpredictable. There are certain tools to deal with this but those are not relevant to this dissertation. We now look at three examples of software that implement blending of pre-defined animation clips.

Ready-Motion Ready-Motion [29] is an online service that allows you to buy and download pre-made animations. They feature mainly hand-crafted animations that cover a variety of stylised locomotion, such as a light-footed character walking quickly, or a heavy-set character walking with weight.

MoCap Online MoCap Online [8] is a similar service that offers motion capture animation packs as opposed to stylised animation. The two hence have different audiences and potential customers. Each animation is provided as a seamless loop and pose-matched so they can be blended with minimal interpolation.

MoClip MoClip [23] is a plugin available on the Autodesk store. It is mainly a rigging toolbox but comes packaged with a cloud-based animation library which allows an animator to rapidly deploy an animation from their database. MoClip is a more powerful solution than the other two as it is built to work within Maya and so offers some features that the others cannot. For example, MoClip allows you to edit the path you would like your character to move along before sending the animation to be processed, and so characters no longer have to walk in straight lines but can follow a NURBS curve. A user can also see a simulation of their animation before buying it, based on the standard model that their animation library holds. However it is not a perfect solution - the blending between clips is not very good and it is clear when the character transitions between motions, and characters can still only move on flat terrain. MoClip is only supported until Maya 2016.

These solutions still provide a relatively naive solution to the problem at hand. Now, the animation quality depends on the number of clips available and the blending between them - an animation with three varied styles of walking is going to look better than one with a single walk repeated again and again. There is also the issue of terrain traversal as these solutions only provide good quality animation for a character moving along a flat plane. Another problem is a lack of variable paths, as while MoClip allows for a curve to be used as a path, the other solutions only offer straight line paths.

2.3.3 AI-Driven Characters

We now look at the most commonly available commercial solutions used throughout film and television - AI-Driven Crowd Simulators. These are currently the most used solution in industry for animating characters as they can produce a great range of animations, from most types of locomotion to actions such as punching or talking to another character, and are capable of animating a great number of characters at once. Another benefit of these crowd simulators is that they often have the ability to have characters react to one another, adding realism and depth to the animation, and are generally capable of creating interactions between the characters and the environment such as adapting to rough terrain and maneuvering around environmental hazards.

Each character in a crowd simulator such as these has a "brain" and it is here that the artificial intelligence component comes into play. A character brain is a complex graph of action and reaction nodes, defining animation flows by letting a character "think" and react to stimuli. As a simple example a character may be directed to walk towards a goal location, but can recognise that there is another character in it's way and change course to walk around them instead of colliding with them.

AI-based characters can be thought of as an extension of the previous method of blending pre-made animation clips, as each node in the graph is a pre-defined and cyclic animation. Their cyclic nature combined with fuzzy logic obfuscates the seams between each animation clip and allows nodes to be connected in any order. Better animations are created by forming more complex brains and different

brains for different types of character, for example in a war scene there would be one brain architecture for a soldier with a sword and a different brain for a soldier with a spear, as the actions they would need to perform would be different. This means that brains are often graphs with hundreds if not thousands of nodes in them. While there are some standard brains that come packaged with any crowd simulator a production often requires customised characters and hence an animator needs to create this brain themselves, a process which can take days per brain.

One thing all crowd simulation software has in common is that there is a steep learning curve and software-specific workflow for each one. This can be unintuitive and requires a lot of additional training for an animator, particularly if they need to learn a different software for a specific project. Industry standard crowd simulators are also often expensive and require a specialised set-up. There is a great lack of simplicity and intuition with these software.

Three examples of AI-based crowd simulators are **Massive** [20], **Golaem Crowd** [13] and **Miarmy** [2]. The basis of these three are all relatively similar and based on the ideas above and so while their implementations differ slightly we will not go into any further detail, but it is worth singling out Massive as the incumbent and first software to approach character animation via this method.

Massive Massive is the original AI-based crowd simulation software, developed in 1999 for use in Peter Jackson's *"The Lord of The Rings"* film trilogy. Massive was ground-breaking, being the first to use the brain architecture as described above, and is to this day the industry leader for crowd related visual effects and autonomous, procedural character generation. Massive for Maya is the Maya plugin version of the proprietary software Massive, and while it lacks all of the features of the standalone version it is still capable of a wide range of motions and inter-character and environmental interactions.

With AI-driven characters the complexity of the brain structures hides the simplicity of the underlying solution. There is no clever underlying machine learning going on here, nothing being inferred or synthesised - the animations are still clips, and they are still blended using some form of interpolation. Creating an animation thus still requires a lot of input from an animator, even though this may be moved away from moving the joints themselves to programming the "thoughts" in a character's head.

2.3.4 Biophysical Character Simulation

Biophysical characters are the current state-of-the-art in terms of fully functioning character simulation. Biophysical characters attempt to define a character as they are built in reality, from a combination of bones, muscles, tissues, fat and skin, all designed as anatomically correctly as possible. Each of these organ types is then further defined on a deeper level, for example muscles can be defined by individual muscle fibers, which lends to incredibly accurate simulations of character movements.

The problem with this method is clear to even an outside observer. The incredible amount of time, skill and resources it takes to create a biologically-accurate character model is simply infeasible for most productions. The level of anatomical knowledge needed requires an animator to be both an expert in animation and modelling as well as a biologist. Creating characters based on real creatures is hard enough, but if an animator wants a fantasy creature then they have to figure out the internal biology of said creature in order to build a functioning model. The process to create a character is incredibly complicated and detailed, requiring models to be built from the bones upwards and to be anatomically exact, else the models may produce unforeseen results. Simulating a motion with such a model is also incredibly computationally expensive due to the sheer number of calculations required for every single muscle fiber, fatty tissue and skin cell, all of which have distinct and realistic properties.

Of course if done correctly this method produces amazingly high quality animation. Mirroring the fundamental properties of nature lends itself exactly to natural animation, and this sort of simulation is really the end-goal for any animator, but currently this is an infeasible option for most.

Ziva Dynamics Leading this field and far ahead of any competitors is Ziva Dynamics [33], a Vancouver-based company founded in 2015 who are on the cutting edge of character animation software. Ziva's eponymous software is a plugin for Maya and defines a number of materials with different physical properties to aid the process of creating realistic character models.

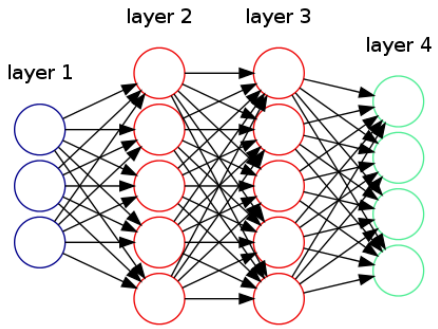


Figure 2.1: A simple neural network

2.3.5 Neural Networks

We now look at research-based solutions to the problem of animating locomotion. These take a greatly different approach to commercial solutions as they are inherently trying out new techniques, meaning that a greater emphasis is placed on the efficiency of producing an animation rather than the quality of the output itself. We look specifically at neural networks and deep learning approaches. Neural networks have changed the face of machine learning, making problems which were previously intractable now trivially easy such as object detection.

A neural network is a biologically-inspired AI consisting of layers of *neurons* chained together in a linear fashion. Figure 2.1 shows a diagram of a simple neural network[3]. The input is passed in at one end and propagated through the network, passing through sets of non-linear transformations and other matrix-vector operations, combining with a set of hyperparameters known as weights. During training the output of the neural network is compared to a reference output using a mathematical expression known as the cost function. The cost function expresses *how inaccurate* the neural network's prediction is. This value can then be used to alter the weights of the layers in a process known as backpropagation. Backpropagation finds the contribution that each neuron makes towards the error and moves the weights in a necessary direction to achieve a lower cost function.

Neural networks can abstract away the details of the problem and learn in an unrestricted manner. They are also capable of learning from very large, high-dimensional datasets, which is useful in terms of motion capture data. An interesting research question is whether these advances can be leveraged within the field of animation automation.

Despite this, there has not been much research into the task of synthesising motion via a neural network. There is currently lots of work being completed on pose estimation and other related subjects, but motion synthesis seems to be slightly before it's time within research at the moment. We cover three attempts to synthesise motion with different neural network architectures, and examine the results of each.

Hierarchical Deep Reinforcement Learning Peng et al. [26] use a hierarchical deep reinforcement learning model, designed around the idea that motion has a hierarchy of control. Their implementation features both low-level controls, where the goal is balance and limb control, and high-level controls where the goal is travelling to a certain location and guiding movement. The reinforcement model learns to control the character at both of these levels simultaneously. It is important to note that, in these experiments, the character is learning to walk through repeated failures, as opposed to being given a dataset of walking data and learning from this. As such, this architecture is useful for examining emergent behaviours in characters, but not for controlling them.

Encoder-Recurrent-Decoder Networks Fragkiadaki et al. [11] define an encoder-recurrent-decoder network that applies an LSTM model in the latent space for predicting the next pose of the body during motion. This network performs best on periodic motions, and an important insight from the paper is that

[3]Image taken from: https://bit.ly/2VKlqXu

both spatial and temporal parameters are important for high quality motion. This model suffers from "drifting", which occurs when the motion gradually picks up noise in it's inputs and eventually converges to an average pose, and from propagated errors due to it's autoregressive nature.

Conditional Restricted Boltzmann Machine Taylor et al. [31] use a conditional restricted Boltzmann machine for predicting the next pose of a body in motion. They use a highly stylised dataset of walks, such as a chicken walk, and allow a style parameter to affect the weights of the network. This encourages high-quality blending between each style and they achieve good results when changing between different styles of motion, but the network suffers from the potential for blow-up from high-frequency noise.

The limitations of the above models were tackled in Holden et al's 2017 work, the Phase-Functioned Neural Network.

2.4 Phase-Functioned Neural Network

In 2017, Holden at al. published a paper titled "Phase-Functioned Neural Networks for Character Control" [16] which presents a novel neural network architecture known as the Phase-Functioned Neural Network (PFNN). The novel contribution of this paper was the use of a phase function to capture the notion of periodic motion, with noted success in locomotion animation. The phase, an input parameter to the system, enumerates how far through a cyclic motion a character is. The work was introduced as a real-time, data-driven character control mechanism with the intention of controlling characters in interactive scenes such as video games. As input the system takes information about a frame of an animation, such as the user controls, the state of the character and the geometry of the terrain, and outputs appropriate motion for the next frame. In this way the character is animated traversing through the scene.

When designing the PFNN there were certain requirements that had to be considered. The network needed to be able to learn from a very large amount of high-dimensional data consisting of combinations of motion capture data and terrain geometries. This is challenging as there can be multiple appropriate movements associated with traversing a terrain patch, for example the character could either step or jump over a small rock. Due to it's intended applications within video games the system also needed to be extremely fast and with low memory requirements.

The PFNN can generate locomotion of six gaits: standing, walking, jogging, jumping, crouching and bumping, where bumping refers to the action of walking into a wall and is used to slow the character and prevent collisions between the character and the environmental geometry. For our implementation we used only four of the possible six gaits, a decision that is discussed in Section 3.1.1.

2.4.1 System Overview

The PFNN is a neural network accompanied by a periodic function known as the phase function. The phase function takes an input, the phase, and outputs a set of weights to be used by the neural network during inference. The periodicity of the phase function is a key component in capturing the cyclic nature of various phenomena, in this context the regularity of locomotion. The phase, a number between 0 and 2π, describes how far through the cyclic motion the character is. As well as the phase the PFNN takes as input information about the character in the current frame, such as the positions and velocities of the character's joints, and the trajectory of the character, which defines the motion the character has taken previously and is predicted to take in the future. The trajectory includes information such as the position of the character and the gait with which the character is travelling. Section 2.4.2.3 explores the system's inputs and outputs in greater detail. See Figure 2.2 for a diagram detailing the PFNN's network architecture[4].

[4]Diagram taken verbatim from the PFNN paper, [16]

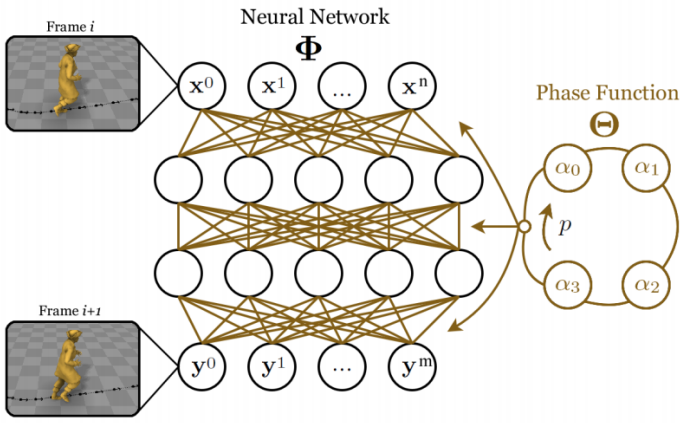


Figure 2.2: The architecture of the PFNN

2.4.2 Training Data

In this section we describe the process of acquiring and pre-processing the data needed to train the PFNN. The PFNN requires a specialised set of data and hence it is important to cover this pre-processing. We cover capturing a motion capture dataset, labelling parameters in the dataset and fitting the dataset to environmental terrain, then specify the inputs required and outputs generated by the network.

2.4.2.1 Motion Capture Dataset

Around 1 hour of raw motion data was captured at 60fps, constituting around 1.5GB of data. This included locomotion in a variety of gaits including walking, jogging and running over obstacles. Also included were specialised actions such as crouch-walking and jumping over obstacles of different heights. The character skeleton used in the dataset consists of 31 rotational joints and follows the design of the skeleton used in the CMU Motion Capture Database [32], which is one of the largest freely-available databases of motion capture data online.

Once the motion data is captured the various control parameters need to be labelled. There are various processes for this, some computational and some manual, which are described below.

Phase Labelling The phase of the motion must be labelled and this is achieved by a computational heuristic which is then manually checked. The phase is labelled such that the frames at which the right foot is in contact with the terrain are assigned a phase of 0, the frames where the left foot is in contact with the terrain are assigned a phase of π, and the next right foot contact is a phase of 2π. The phase is then interpolated for the in-between frames. When standing the phase is given a cycle of 0.25s and allowed to cycle continuously so as to fit within the design of the phase function.

Gait Labelling The gait of the locomotion is labelled as a six-dimensional one-hot vector covering the actions of standing, walking, jogging/running, crouching, jumping and bumping. Note that the motions of jogging and running were at this point combined into one. This process is performed manually.

Trajectory and Terrain Height The trajectory of the character describes the character's past and future motion. First, the character's *root transform* is calculated by projecting the midpoint between the character's hip joints onto the terrain. The facing direction is computed by averaging the vector between the shoulder joints and the vector between the hip joints, then taking the cross product of this with the upwards direction. This gives us the character's trajectory position and direction at the current frame. The heights of the trajectory are then calculated at three points: the current point on the trajectory, and points 25cm to the left and right of the central point perpendicular to the facing direction.

2.4.2.2 Terrain Fitting

As a secondary contribution the authors also present a process to fit motion capture data to a large database of artificial heightmaps extracted from video game environments. While interesting this is not relevant to this thesis and so the process is omitted.

2.4.2.3 Input and Output Parameters

In this section we describe the input and output parameters for the PFNN. Recall that, for a frame of animation i, the PFNN generates frame $i+1$. The PFNN takes as input some vector $\boldsymbol{x}_i$ and the phase p and computes vector $\boldsymbol{y}_i$. The full parameterisation of these vectors will be described below.

First we give an intuition of the parameterisation for $\boldsymbol{x}_i$. This can be split conceptually into two parts; input regarding the current state of the character, and input regarding the trajectory of motion of the character. For the state of the character we take the positions and velocities of the character's joints, local to the character's root transform at frame $i-1$. Note that the skeleton used in the PFNN has $j = 31$ joints. We next consider the trajectory of the character. The trajectory is used to describe the past and future motion of the character and so we observe a window centred at frame i and examine every tenth surrounding frame, choosing $t = 12$ frames in order to cover 1 second of motion in the past and 0.9 seconds of motion in the future. For each of these 12 sampled frames we extract a set of features, including: the character's trajectory position and direction, local to the character's root transform at frame i; the character's gait represented as a six-dimensional one-hot vector; and the heights of the terrain at the central, left and right points of the trajectory, local to the character's root transform at frame i.

The full parameterisation of $\boldsymbol{x}_i$ as described above consists of a vector $\boldsymbol{x}_i = [\boldsymbol{t}_i^p, \boldsymbol{t}_i^d, \boldsymbol{t}_i^g, \boldsymbol{j}_{i-1}^p, \boldsymbol{j}_{i-1}^v, \boldsymbol{t}_i^h] \in \mathbb{R}^n$, where:

- $\boldsymbol{t}_i^p \in \mathbb{R}^{2t}$ are the sampled trajectory positions in the 2D horizontal frame relative to frame i
- $\boldsymbol{t}_i^d \in \mathbb{R}^{2t}$ are the sampled trajectory directions in the 2D horizontal frame relative to frame i
- $\boldsymbol{t}_i^g \in \mathbb{R}^{6t}$ are the semantic variables representing the character's gait, defined as a six-dimensional one-hot vector
- $\boldsymbol{j}_{i-1}^p \in \mathbb{R}^{3j}$ are the character's joint positions relative to frame $i-1$
- $\boldsymbol{j}_{i-1}^v \in \mathbb{R}^{3j}$ are the character's joint velocities relative to frame $i-1$
- $\boldsymbol{t}_i^h \in \mathbb{R}^{3t}$ are the sampled trajectory heights of the left, central and right positions of the trajectory relative to frame i

We thus have $n = 342$, and note that the individual components of the one-hot vector $\boldsymbol{t}_i^g$ are active when the character is travelling at the following gait: 1) standing, 2) walking, 3) jogging, 4) crouching, 5) jumping, 6) bumping.

The output of the PFNN gives us information about the movement of the character, as well as the change in phase, predictions of the future trajectory, and binary labels indicating which whether the heel or toe joints of each foot are in contact with the terrain. This parameterisation can be expressed as $\boldsymbol{y}_i = [\boldsymbol{r}_i^x, \boldsymbol{r}_i^z, \boldsymbol{r}_i^a, \boldsymbol{p}_i, \boldsymbol{c}_i, \boldsymbol{t}_{i+1}^p, \boldsymbol{t}_{i+1}^d, \boldsymbol{j}_i^p, \boldsymbol{j}_v^p, \boldsymbol{j}_a^p] \in \mathbb{R}^m$, where:

- $\boldsymbol{r}_i^x \in \mathbb{R}$ is the translational velocity of the character's root transform in the x direction, relative to the character's forward facing direction
- $\boldsymbol{r}_i^z \in \mathbb{R}$ is the translational velocity of the character's root transform in the z direction, relative to the character's forward facing direction
- $\boldsymbol{r}_i^a \in \mathbb{R}$ is the angular velocity of the character's root transform around the upwards direction
- $\boldsymbol{p}_i \in \mathbb{R}$ is the change in phase

- $\boldsymbol{c}_i \in \mathbb{R}^4$ are the foot contact labels
- $\boldsymbol{t}^p_{i+1} \in \mathbb{R}^t$ are the predicted trajectory positions relative to frame $i+1$
- $\boldsymbol{t}^d_{i+1} \in \mathbb{R}^t$ are the predicted trajectory directions relative to frame $i+1$
- $\boldsymbol{j}^p_i \in \mathbb{R}^{3j}$ are the character's joint positions local to the character's root transform at frame i
- $\boldsymbol{j}^p_v \in \mathbb{R}^{3j}$ are the character's joint velocities local to the character's root transform at frame i
- $\boldsymbol{j}^p_a \in \mathbb{R}^{3j}$ are the character's joint angles local to the character's root transform at frame i, expressed using the exponential map [14]

Thus our output is of size $m = 311$. Note that the values stated here are different to those stated in the PFNN paper [16]; the values given in the paper are inconsistent with the actual training data used by the PFNN, and so the author has taken the liberty of correcting these.

2.4.3 The Network and Training

In this section we discuss the architecture of the neural network and the process by which it is trained. The network is trained end-to-end, and the network weights change dynamically as a function of the phase, which increases the overall expressiveness of the network while allowing it to retain a compact structure. This design choice allows the PFNN to avoid mixing data from different phases which helps keep the animation output clean, seamless and of high quality.

2.4.3.1 Network Architecture

Given input parameters $\boldsymbol{x}_i \in \mathbb{R}^n$, output parameters $\boldsymbol{y}_i \in \mathbb{R}^m$ and a phase parameter $p \in \mathbb{R}$, the three-layer neural network $\boldsymbol{\Phi}$ is defined by the following equation:

$$\boldsymbol{\Phi}(\boldsymbol{x}; \boldsymbol{\alpha}) = \boldsymbol{W_2}\ \mathrm{ELU}(\boldsymbol{W_1}\ \mathrm{ELU}(\boldsymbol{W_0 x} + \boldsymbol{b_0}) + \boldsymbol{b_1}) + \boldsymbol{b_2}$$

where the parameters of the network $\boldsymbol{\alpha}$ are defined by

$$\boldsymbol{\alpha} = \{\boldsymbol{W_0} \in \mathbb{R}^{h \times n}, \boldsymbol{W_1} \in \mathbb{R}^{h \times h}, \boldsymbol{W_2} \in \mathbb{R}^{m \times h}, \boldsymbol{b_0} \in \mathbb{R}^h, \boldsymbol{b_1} \in \mathbb{R}^h, \boldsymbol{b_2} \in \mathbb{R}^m\}$$

Note here that h is the number of hidden units used on each layer, which is set to 512. The activation function ELU is the exponential rectified linear function [6] defined by

$$\mathrm{ELU}(x) = \max(x, 0) + \exp(\min(x, 0)) - 1$$

2.4.3.2 Phase Function

In the PFNN the network weights $\boldsymbol{\alpha}$ are computed each frame by a separate function known as the phase function, $\boldsymbol{\Theta}$. This can be written as $\boldsymbol{\alpha} = \boldsymbol{\Theta}(p, \boldsymbol{\beta})$, where the phase function takes as input a phase p and parameters $\boldsymbol{\beta}$. Theoretically $\boldsymbol{\Theta}$ could take many forms, but in this work $\boldsymbol{\Theta}$ is specified as a cubic Catmull-Rom spline [5]. A Catmull-Rom spline is defined by a set of control vertices we can think of each control point $\boldsymbol{\alpha}_k$ as representing a certain configuration of weights in the neural network. The Catmull-Rom spline then performs a smooth interpolation between these weight configurations with respect to the phase parameter. Additionally to this, a Catmull-Rom spline is chosen as it is easily made cyclic by setting the start and end vertices as the same.

The specific Catmull-Ron spline used by the PFNN is defined below. This spline has a set of four control

vertices, $\boldsymbol{\beta} = \{\boldsymbol{\alpha}_0, \boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \boldsymbol{\alpha}_3\}$. The cubic Catmull-Rom spline is then defined as

$$\begin{aligned}\boldsymbol{\Theta}(p;\boldsymbol{\beta}) = \boldsymbol{\alpha}_{k_1} \\ &+ w(\frac{1}{2}\boldsymbol{\alpha}_{k_2} - \frac{1}{2}\boldsymbol{\alpha}_{k_0}) \\ &+ w^2(\boldsymbol{\alpha}_{k_0} - \frac{5}{2}\boldsymbol{\alpha}_{k_1} + 2\boldsymbol{\alpha}_{k_2} - \frac{1}{2}\boldsymbol{\alpha}_{k_3}) \\ &+ w^3(\frac{3}{2}\boldsymbol{\alpha}_{k_1} - \frac{3}{2}\boldsymbol{\alpha}_{k_2} + \frac{1}{2}\boldsymbol{\alpha}_{k_3} - \frac{1}{2}\boldsymbol{\alpha}_{k_0})\end{aligned}$$

where

$$w = \frac{4p}{2\pi} \pmod 1$$

and

$$k_n = \left\lfloor \frac{4p}{2\pi} \right\rfloor + n - 1 \pmod 4$$

During runtime there is the option to pre-compute the phase function to some degree, giving rise to a tradeoff between speed and memory usage for the network. We discuss three options. The *constant* method computes $\boldsymbol{\Theta}$ for $n = 50$ locations along the phase space, then during runtime selects the weights at the nearest pre-computed phase location. The *linear* method computes $\boldsymbol{\Theta}$ for $n = 10$ locations along the phase space, then at runtime piecewise linear interpolation is performed between these samples. The third option is the *cubic* method, where the full cubic Catmull-Rom spline is computed at runtime, with no pre-computation. The tradeoff here is between faster speeds but higher memory consumption for the constant method, compared to slower speeds but lower memory usage for the cubic method, with the linear method providing a mid-way point for the two.

2.4.3.3 Training

For each frame i the input parameters $\boldsymbol{x}_i$, output parameters $\boldsymbol{y}_i$ and phases p_i are stacked into matrices $\boldsymbol{X}$, $\boldsymbol{Y}$ and $\boldsymbol{P}$. $\boldsymbol{X}$ and $\boldsymbol{Y}$ are then normalised using the means $(\boldsymbol{x}_\mu, \boldsymbol{y}_\mu)$ and standard deviations $(\boldsymbol{x}_\sigma, \boldsymbol{y}_\sigma)$. They are then additionally scaled by weights $(\boldsymbol{x}_w, \boldsymbol{y}_w)$, chosen specifically to diminish the importance of the parameters relating to the joints and increase the influence of the trajectory over the character's motion. This results in a more responsive character.

Training is an optimisation problem with respect to the phase function parameters $\boldsymbol{\beta}$, giving rise to the following cost function:

$$\text{Cost}(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{P}; \boldsymbol{\beta}) = \|\boldsymbol{Y} - \boldsymbol{\Phi}(\boldsymbol{X}; \boldsymbol{\Theta}(\boldsymbol{P}; \boldsymbol{\beta}))\| + \gamma|\boldsymbol{\beta}|$$

Here, the first term gives us the mean squared error of the regression result, and the second term provides regularisation to ensure the weights do not become too large. The regularisation term is controlled by a constant $\gamma = 0.01$.

The model is implemented in Theano [3] and uses the Adam optimisation algorithm [19] and dropout [30], with a retention probability of 70%. The model is trained in minibatches of size 32 and full training takes 20 epochs, taking around 30 hours on a NVIDIA GeForce GTX 660 GPU.

2.4.4 Runtime

The PFNN paper comes with an accompanied Github repo with code to train and run the network [12]. Of particular interest in this project is the demo, a C++ project which runs the PFNN as a controller for a character that can be influenced by user input via an Xbox controller, and renders a game world for the user to test the responsiveness of the network in. This is the code that our implementation of the PFNN was adapted from.

2.5 Mathematical Background

In this section we briefly cover a general set of mathematical formulae and concepts that are relevant to the project and used in the execution and implementation of our code.

2.5.1 Euclidean Distance

The Euclidean distance between two points gives the length of the straight line connecting the points. The Euclidean distance of points $p = (p_0, p_1, \ldots, p_n)$ and $q = (q_0, q_1, \ldots, q_n)$ is

$$\text{dist}(p, q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

2.5.2 Dot Product

The dot product of the vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ is defined as the sum of the products of their corresponding vector components. This is denoted by $\boldsymbol{u} \cdot \boldsymbol{v}$ and returns a scalar value. Formally,

$$\boldsymbol{u} \cdot \boldsymbol{v} = (u_0, u_1, \ldots, u_n) \cdot (v_0, v_1, \ldots, v_n) = u_0v_0 + u_1v_1 + \cdots + u_nv_n$$

2.5.3 Barycentric Coordinates

Barycentric coordinates parameterise a coordinate space by a weighted combination of a set of reference points. Consider a line segment defined by points A and B. A point P on the line can be expressed as

$$\begin{aligned} P &= A + t(B - A) \\ &= (1 - t)A + tB \end{aligned}$$

for some value t. This can be written as

$$P = uA + vB$$

where $u + v = 1$ and point P lies on the line if $0 \leq u, v \leq 1$. Thus u and v are weights at the end points of the line, and written as (u, v) they define the barycentric coordinates of P with respect to A and B. At A the barycentric coordinates are $(1, 0)$, and at B they are $(0, 1)$. Barycentric coordinates are commonly used in computer graphics to parameterise triangles by placing weights at each vertex defining the triangle [9].

2.5.4 Quaternions

Quaternions are a way of expressing rotations by extending the complex number system to include three imaginary dimensions $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{k}$ and one real dimension. A quaternion is defined by four scalars $a, b, c, w \in \mathbb{R}$ and is expressed as

$$w + a\boldsymbol{i} + b\boldsymbol{j} + c\boldsymbol{k}$$

We can think of quaternions as expressing a rotation of θ around an axis defined by the vector (x, y, z). Consolidating this with the above equation we get:

$$\begin{aligned} w &= \cos(\theta/2) \\ a &= x \sin(\theta/2) \\ b &= y \sin(\theta/2) \\ c &= z \sin(\theta/2) \end{aligned}$$

Chapter 3

Project Execution

In this section we will explore our application of PFNNs to Maya. Firstly, the set of design decisions that occurred throughout the project will be outlined with reasoning for given choices. Following this will be given a detailed architecture of the data extraction and animation generation processes within Maya in addition to the process of inference using a cloud-hosted implementation of PFNN. During this section the problems encountered during implementation will be discussed and the solutions presented. Then a concrete set of improvements and innovations implemented during the project to increase the quality of the generated animations will be examined.

3.1 Overview of Solution

First we re-define the goal of our system. We want an animator working in Maya to be able to select a character, and draw a path for the character to walk along. Somehow this information will be fed into the PFNN, which will generate the animation of the character traversing along the path. We feed this output data back into Maya and use it to animate the character. We decompose this end-goal into a number of sub-goals, and approach each individually.

First, we need the user to select a character and draw a path. This can all be achieved with built-in Maya functions, and the implementation of this is described in Section 3.2.1.

Next, we need to extract the relevant information from Maya to send to and feed into the PFNN. We know that at each frame the PFNN requires both information about the character and the trajectory of the character. Regarding the character, we can send the initial information of the character to the PFNN, then feed the outputs of the PFNN at each frame into the next frame in an autoregressive manner. For the trajectory, we will send information about the entire path to the PFNN, then take the subsampled window of frames that define the trajectory on-the-fly from this for each frame. Extracting the joint and path data is covered in Sections 3.2.2 and 3.2.3 respectively. For each frame, the change in phase can be fed in from the output of the PFNN at the previous frame. This covers our inputs to the PFNN needed at every frame to generate the entire animation, the full process of which is further detailed in Section 3.2.4.

We now need to select some useful information to send back to Maya in order to generate the animation. We choose to send back the world space joint positions of the character at each frame, along with the frame number. Using the joint positions will eventually cause us problems, discussed in Section 3.3.4, but for now will suffice. Once this information is in Maya we generate the animation, using inbuilt Maya commands and a buffer to allow the user to undo and redo the animation. This process is discussed in Section 3.2.5. With that, our system is complete.

3.1.1 Design Decisions

In this section we cover some implementation choices that were made, justifying each.

Supported Gaits We chose to support four of the six possible gaits generated by the PFNN: standing, walking, jogging and crouching. These were chosen as they did not depend on additional information from the environment, unlike the final two gaits, jump and bump. In the PFNN demo these gaits gather extra information from the world as the jump and wall locations are semantically labelled, which was an extra step that we did not want our end-users to have to take. It was also unclear how one would go about labelling an object in Maya in this way - while this is an easy task in games engines such as Unity, Maya has no built-in support for a task of this nature, and so a solution here would have to be hand-crafted, which was deemed more arduous than rewarding.

Path Format An important decision was how to represent the path that the user wants their character to follow. We decided upon a NURBS curve as this allows the user incredibly granular control over the path, allowing them to define any path they may require. We assume the curve is created with uniform parameterisation, which is default in Maya so is not a large assumption to make. We also require the curve to vary in the X and Z axes only, and hence to be horizontal - this is specified to the user, and a potential future solution to this is discussed in Section 4.4.4.

Terrain We require the terrain to be a polygonal mesh as opposed to a NURBS surface. This is a fair requirement to ask of the user as it is likely that their terrain will be of this form anyway; generally, NURBS surfaces are used for more scientific modelling than animation.

Cloud-Hosted PFNN While the Maya code must obviously be run on the user side, we decided to remove the PFNN code from the user side and to host this in the cloud instead. This was mainly due to the fact that certain libraries and header files need to be custom installed to run the PFNN code, and it was deemed appropriate to take this overhead away from the user. We used Google Cloud Platform to run the PFNN code on a single CPU with 3GB of RAM running Ubuntu 16.04. We allow ingress traffic from all IPs along TCP port 54321, which enables our communication with Maya.

Skeleton Hierarchy We chose to implement our system according to the 31 joint skeleton hierarchy used when training the PFNN. This means that the system only works for a selected skeleton design. There can never be a perfect solution to this issue, as skeletons are entirely user-defined, but a potential generalisation is discussed in Section 3.3.5.

Plug-in vs Script When deciding to implement our Maya side as a full plug-in or as a script, we looked towards the Maya documentation [17]. This states that plug-ins are best suited for extending the underlying architecture of Maya itself, for example by adding a new node type or adding a new rendering pipeline. This was deemed above what we needed to implement and so following this guidance we chose to implement our software as a script.

Joints vs Control Curves Many animators use control curves to abstract away from the individual joints in their rig, and animate these instead of the joints themselves. We choose to animate each joint individually instead of the control curves as this aligns better with the output of the PFNN. A potential solution to use the output of the PFNN to animate the control curves of the character is outlined in Section 4.4.3 and left for future work.

Networking We choose to implement the networking between our two scripts via the Berkeley sockets interface, using the JSON file format to encode our requests and responses. To achieve this, in Python we used the `socket` and `json` libraries. In C++ we used the `sys/socket.h` and `nlohmann_json` header files.

3.2 Detailed Architecture

In this section we discuss the architecture of our solution in more depth. We first consider how the user creates the character and defines the path they want the character to move along. We then discuss how the system extracts the necessary data to send to the PFNN. We next look at how the PFNN generates the animation from this input, then finally we cover how the data is used once sent back to Maya to create our final animation.

3.2.1 User Input

In this section we first discuss the construction of the character's skeleton, then the drawing of the path the user wants the character to move along.

3.2.1.1 Constructing a Skeleton

Maya has inbuilt tools for creating a skeleton, and we use these to define a 31 joint skeleton identical to the one used in the training data for the PFNN. The hierarchy of this skeleton is not specified in the accompanying paper and so we looked through the PFNN demo code, eventually finding the training data in the form of animation data as BVH files. BVH is a file format developed by Biovision designed to encode motion capture data [4], which specifies files with two sections: the first, a header section which describes the joint hierarchy and initial pose of the skeleton; and the second, a data section comprising of the positions of the joints, with each line representing one frame of an animation. It was from these files that the skeleton hierarchy was extracted. For now we specify this exact skeleton must be used with our implementation, but in Section 3.3.5 we define a mapping to a more standardised skeleton in order to encourage generalisation in our system.

3.2.1.2 Constructing a Path

The user draws a path as a NURBS curve, starting from the character and continuing in any way they desire. The path can be at any height in the world, but is required to be vary in the X and Z axes only, with no vertical component, in order for the data extraction calculations to work correctly. Our implementation is built robustly so that the user can draw any shape path here and will receive an animation of the character travelling along it. There are some caveats to the quality of the produced animation dependent on the path, for example paths with sharp angles will produce lower quality animation, and in Section 4.2.1 we discuss a number of these. In Section 4.4.4 we discuss a potential future feature of the script, a path drawing tool that would enable the user to draw curves that will provide high quality animation every time.

The next two sections will describe the process of automatically extracting the appropriate data from Maya as required by the PFNN.

3.2.2 Extracting Joint Data

The PFNN requires both information about the character and the trajectory as input. As previously discussed, we are adapting the PFNN from an interactive version to one that receives both initial information about the position of the character and a path the character is to walk along. The key difference here is in implementation - the first is able to read information about the character and the trajectory as they are updated in real-time by the person using the demo, whereas the second requires us to gather the appropriate information from the scene beforehand, send this to the PFNN, then use the right information at the right time to generate each frame of the animation correctly.

In this section we cover the process of extracting the inputs for the PFNN from Maya. We first cover character information in the form of joint positions and velocities, then look at the path information, requiring path positions, heights, gaits and directions. We investigate two interesting problems; how to sample points from the path at an appropriate rate for the animation, and how to get the height

information of the terrain, which is non-trivial in Maya as opposed to some other software (such as the PFNN demo).

3.2.2.1 Joint Positions

As previously discussed, in the PFNN we will only need the initial joint positions of the character as for every subsequent frame the joint positions will be taken from the output of the PFNN for the previous frame, in an autoregressive manner. Whilst the PFNN requires joint positions local to the character's root transform we can do the conversion from world space to local space on the PFNN side where we have access to more powerful structures such as matrices, and so at our Maya side we choose to simply gather the world space positions of the joints. To do this we use the Maya Commands interface, querying the `xform` command with the `worldSpace` and `translation` flags enabled.

3.2.2.2 Joint Velocities

Similar to joint positions, the PFNN requires joint velocities to be local to the character root transform at each frame. However finding the initial joint velocities presents a problem - how do we figure out how fast the character is going in the first frame? We choose here to make an assumption - that the initial velocities of each joint, in the world space x, y and z directions, are 0. This may seem like a rather large assumption and so we explain the justification below.

Firstly through experimentation we found that, no matter what the initial velocities are set at, the animation stabilises within around 20 frames and the animation output from this point onwards is not affected by the initial velocities. This means that even if the velocities are initialised to some ridiculously high number, while the animation will initially produce some shocking result such as the character jumping into the air, within 20 frames the character setlles back into a jog - at 60fps, this is just $\frac{1}{3}$ of a second. This happens because the velocities largely only affect the positions of the joints at the next frame, and as the PFNN uses these positions and velocities in an autoregressive manner, after 20 frames it has fed back into itself enough that the input at frame 20 has no correlation to the input at frame 0. For a video example of this, see Appendix A. This experiment confirms an insight from the paper, where Holden et al. state they scale the weights of the inputs relating to the joints down in order to emphasise the importance of the trajectory on the animation output. As such, we know that as long as we choose a value that is appropriate most of the time, then we have an acceptable solution.

Setting the initial velocities to 0 is perfect in the situation where the character begins their locomotion by standing still, and this is assumed to be the case that most animators will be working with - it is less likely that the character will already be in some kind of motion and then the animator chooses to use our system to complete the rest of the motion. Thus, initial velocities of 0 seems to be an appropriate assumption. Having experimented with characters starting in a wide variety of initial poses we concluded that this value for initial velocities produces generally high quality animation. Additionally, if an animator is unsatisfied with the results of the first 20 frames, they could always go in to these frames and manually edit them to suit their desired output.

An alternate solution to this method is suggested. We could check whether the character has any frames of animation before the initial frame our animation is requested at. If there is no previous animation, we assume the character is standing, and set the initial velocities to 0. If there is previous animation we use the joint positions in the previous frame and the joint positions in the current frame to calculate the velocity of the joints between the two frames. We then use these velocities as our initial velocities. This implementation has only been theorised and not implemented but it could potentially increase the general expressiveness of our system.

3.2.3 Extracting Path Data

With the necessary information about the character retrieved we can now gather the appropriate information about the path. For every frame of our animation, we require the position and normalised direction of our current location on the path, as well as the terrain heights at the left, right and central positions of

the path, and the current gait at the point on the path. We hence need to sample the appropriate points on the path such that each point represents the correct desired location for every frame of animation. This gives rise to our first issue, the Path Sampling Problem. Once we have found an appropriate method by which to sample our path, we can extract our path positions, directions and gaits. Extracting the heights gives rise to two more problems, requiring some computational geometry for their solutions. We explore these issues below.

3.2.3.1 The Path Sampling Problem

Each point we sample along the path represents the position of the character's root transform at a frame i, and so we must ensure that we are sampling the path at appropriate points for our final animation. If we sample the path too sparsely the character will "glide" along the path, translating horizontally by too much with each step. If we sample the path too densely the character will not translate enough, causing a limping effect. We now discuss two methods for sampling the path.

Equidistant Sampling Our first implementation sampled the path at equidistant points according to the number of frames of animation the user desired. In this implementation, the user could specify the length of the animation they wanted, and the animation generated would be of that length. The user could also choose multiple gaits and specify which frames they wanted these to start from. For example, if we had a path of length 1000, and the user wanted 2000 frames of animation, then we would sample the path at every 0.5 units. The user could choose to walk for the first 500 frames then jog for the next 1500. This seemed like an acceptable solution as it allowed the user a good amount of control over their animation.

However, there is a significant problem with this approach. Recall that in Section 3.1.1 we chose to support four gaits: standing, walking, jogging and crouching. Note also that these gaits travel at different speeds. If we sample the path equidistantly then when the animation is fed back from the PFNN the character has the same horizontal velocity throughout the animation, because the sampled path points correspond exactly to the position of the character's root transform in the output of the PFNN. Hence with equidistant sampling our character travels at a constant speed based on the number of frames the user chose. The first issue here is that it is now down to the user to find a good number of frames to provide them with quality animation. If they choose too few frames the character will appear to be travelling too fast and will glide across the floor, not taking proper steps. If they choose too many frames the character will travel too slowly, taking multiple steps in the space for one step. See Appendix A for examples of this. Hence, while we have given the user granular control over the desired animation length, we force them to experiment with different frame numbers until they find an appropriate animation, which is a problem that we should be able to solve for the user instead.

The second issue is that this looks even worse if the user wants multiple types of locomotion along one path. Say the user wants their character to walk for half the path, then run for the other half. They find a number of frames that makes the first half of the animation look good, but the second half is now too slow. Conversely, if they find a frame number that makes the running section look good, then their walking section is sampled too sparsely and the character glides along the path.

Gait-Specific Sampling Rates Both of these problems can be solved if we take away one control from the user, the animation length, in return for the guarantee of high quality animation. The alternative solution is based on the idea that each gait travels at a set speed, and we can leverage this information to choose a *sampling rate* that is appropriate for each gait. The sampling rate defines the appropriate number of world space units between each sampled point on the path, and differs for each gait. Now, the user chooses how much of the path they want the character to move along in a specific gait, and the code calculates the number of frames this animation will take, telling this to the user. For example the user could choose to walk for half of the path then run for half of the path. This is arguably more intuitive than the previous implementation, as it is likely an animator will want to define their changes in locomotion in terms of space as opposed to time. If the user wants a different length of animation they can change the length of the path or the gait the character is travelling at.

The sampling rates are chosen such that they provide the best animation quality for a specific gait - this

means there is no gliding or foot sliding and the character translates an appropriate amount with each step. Thus, while the animator can no longer choose exactly how long the animation they receive is, the quality of the animation they receive is guaranteed with no experimentation needed - this improves upon the previous solution as either way the animator would have come to roughly the same number of frames, just through trial and error and inspecting the animation quality themselves. The sampling rates were found through experimentation; we chose a fixed path length for the equidistant implementation, then tested different total frames for each gait until we reached a high animation quality. The sampling rate was then calculated as $\frac{\text{path length}}{\text{total frames}}$. Videos of these experiments can be found in Appendix A.

To see details of the implementation of this, see Section 3.3.1. Note that with this implementation we extract the path gaits from user input, which are then used to calculate the total frames of the animation.

3.2.3.2 Extracting Positions and Directions

We now discuss the extraction of the path positions and directions.

Path Positions Now we have an appropriate sampling rate for our path we can extract the rest of the relevant path information from our scene. Recall that our path positions are defined by the world space 2D (X, Z) coordinates of the path at any point. We can think of this as a projection of our path down onto the terrain, and hence it does not matter how high the path is drawn as the X and Z positions will stay the same. Note that the user must draw a path that is horizontally flat, changing in X and Z but not in Y, otherwise our calculations throughout do not provide the correct results. At this stage we simply inform the user that the path needs to be horizontal, but in Section 4.4.4 we discuss a path drawing tool that forces the user to draw a horizontal path. The positions of are path are found using the path sampling rate as above.

Path Directions For any point on the path, the direction is found by calculating the normalised tangent of the curve from that point and then taking the X and Z values of this tangent. Note again that this relies on our path curve being horizontal in Y. We use the same sampling method as defined in above.

3.2.3.3 Calculating Height Projections

For a given point on the path we are looking for three heights. These are the world space heights of the terrain at the given point, and the heights 25cm to the left and right of the point, with left and right taken perpendicular to the forward direction of the curve. We discuss how to retrieve the (X, Z) coordinates of these left and right points, then three methods for sampling the heights at each point.

Height Positions We have already discussed how to sample our path at the appropriate positions for our animation and that has given us the world space locations of our central points, those that are directly on the path itself. For each of these points we also need the world space positions of two additional points, one 25cm left and one 25cm right of the central point. These left, right and central points will then be projected onto the terrain and the heights of the terrain calculated.

To calculate these positions we can calculate the normal of the central position on the curve, then use this to find the points 25cm left and right of the point perpendicular to the forwards direction of the point. However we come across a problem when trying to classify whether a calculated point is on the left or right of the curve. The solution to this is given in Section 3.3.2, but for now we elide the mathematics and assume we have found the correctly sorted lists of left, right and central points along the path. With the (X, Z) positions of these points retrieved, we can now look at three methods for sampling the terrain heights at these points.

Naive Sampling We now discuss three methods for sampling the path heights. As a first attempt we could simply take the height of the curve itself at each point. There are a multitude of problems

here: most significantly, this method requires the path to align exactly with the terrain else the heights will be offset by some amount, causing the character to not walk on the terrain but above it. This can be achieved easily if the terrain is entirely flat but as soon as this is not the case an appropriate path curve for this method is essentially impossible for an animator to implement, as it would have to run exactly across the surface of the terrain geometry. With the standard Maya curve drawing tools this would be both unintuitive and require a level of granular precision that is not readily supported. Hence this method is too naive for our implementation.

Nearest Vertex Sampling We next discuss an improved but still naive method of height sampling. In an ideal world we would be able to query a mesh within Maya at any specified (X, Z) coordinates and have Maya return the Y value at this point. This is not a functionality that is built into Maya, and hence we must write our own code to achieve this. In this method we choose our path height as the height of the nearest vertex in the terrain mesh.

We have the (X, Z) position of a point that we want to sample the terrain height of. We project this point onto the terrain, independent of the height of the path curve. Next, we use the Maya Python API to obtain information about the terrain mesh. We traverse the DAG until we reach the terrain's `mFnMesh` node, a node class which gives us access to a set of mesh-based functions. We use the `getPoints` function to retrieve a list of the world space positions of all vertices in the mesh. We traverse this list, checking for each vertex what the Euclidean distance is between our desired point and the vertex, and saving the vertex with the shortest Euclidean distance - this is the closest vertex to the point. We take the height of this vertex and use that as our path height for the point.

This method has it's own issues, primarily that it's performance is heavily dependant on the number of polygons wihin a mesh. The less polygons in a mesh the further away the vertices may be, meaning that for point between vertices the character is choosing heights either above or below where appropriate. This further results in a stepping effect, as evidenced in videos available in Appendix A, as the character is choosing the previous vertex for points up until halfway between the two vertices, then the next vertex for points after that. These results are imprecise and heavily dependent on the structure of the terrain geometry, so we look for a better method of sampling heights.

Barycentric Sampling Our third method is based on polygonal mesh triangulation as described in Section 2.1.1.1. Maya stores any polygonal mesh as a list of vertices and a list of real numbers describing the number of edges per polygon in the mesh. This second list could, for example, be $[3, 4, 4]$ to indicate a mesh with three faces, one triangular and two quads. Under the surface however these quads are decomposed and stored as triangles, allowing us to do planar calculations on them. These triangles are stored in an additional list which, for each triangle, holds the indexes of the three vertices which comprise said triangle. All these lists are available if we find the correct node in the DAG hierarchy.

The intuition behind this method is as follows. For a point (X, Z) we want to find the point on the terrain mesh with these coordinates and extract it's Y position. We first traverse the DAG hierarchy to find the mesh's `mFnMesh` node as before, and use this to access two lists. The first list contains all the vertices in the mesh, and the second is a list of triangle indices, indicating which vertices comprise which triangles.

We now iterate through the world space positions of the vertices, calculating their Euclidean distance to point (X, Z) and finding the closest vertex to our point, V. We check whether our point lies exactly on this vertex; if so, we take the Y value at this vertex, and can move on to the next point on the path. If not, then the point must lie in one of the triangles that has this V as one of it's vertices. We look in our list of triangles and create a separate list of any possible triangles that our point may lie within. Now, to find exactly which triangle our point lies in, we use barycentric coordinates to parameterise the triangle, then check whether the point lies within the triangle. If so, we interpolate between the height values of the triangle's three vertices to find the height of our point. If not, we move on to our next possible triangle, until we find the triangle our point lies within.

This method provides us with an accurate way of calculating the height of a point, despite the fundamental limitation that Maya does not store information about every possible point on a mesh and only stores information about the mesh's vertices. The accuracy comes from the planar triangles leaving no room for misinterpretation when interpolating the heights. For the mathematics behind this implementation,

see Section 3.3.3. For a comparison of height sampling methods, see Section 4.2.2.

3.2.4 Inference

We have now gathered the appropriate character and path data from Maya and sent this to the PFNN. We feed the data in to the network which can now be used in inference to produce a set of appropriate movements for a given path. The PFNN generates this animation frame-by-frame, and in this section we detail the process of gathering input to the network for each subsequent frame of the animation.

We update the inputs regarding the character autoregressively, using only the output of the PFNN and no external information. For a frame i the PFNN outputs the joint positions and velocities at frame $i+1$, so when for frame $i+1$ we can use these to generate the positions for frame $i+2$ and so on.

We update the phase based on both the current state of the character and the suggested change in phase, as output by the PFNN. Namely, if the character is standing, the phase cycles at a minimum duration of around 0.25s to ensure the phase function is allowed to cycle continuously, and so we account for whether the character is standing when computing our updated phase. The phase p is then modulated to loop in the range of $0 \leq p \leq 2\pi$.

The updated phase is calculated as follows:

$$p_{i+1} = (p_i + (s + 0.9 + 0.1) * 2\pi * \delta_p) \bmod 2\pi$$

where p_i represents the phase at frame i, δ_p represents the change in phase as output by the PFNN, and s represents a numerical value indicating how much the character is standing, calculated by:

$$s = (1.0 - g_i^s)^{0.25}$$

where g_i^s is the binary value representing whether the character is standing or not, taken from the gait variables at frame i.

Where our character is updated solely from the output of the PFNN, we update the trajectory based solely on the request from Maya. This is an implementation decision based on the belief that an animator will want their final animation to be in strict adherence to the path they've drawn for the character to traverse along. An alternative implementation would be to take the predicted future trajectory from the PFNN and blend these predictions with the request. This is how the PFNN demo works, blending the predicted trajectory with the trajectory calculated from the user input. Here, this is appropriate as the user may change the locomotion whenever and choppily, meaning some care is needed - you cannot use purely user input for the future trajectory as it may change too often, and you cannot use purely PFNN output because the user input needs to be considered. However with our implementation we do not have this problem; the user has already pre-defined their input, so while the PFNN still outputs it's predictions for the trajectory we are safe to ignore these.

The implementation here is relatively simple. Centred at frame i, we take a subsampled window of $t = 12$ frames from the path, with 6 frames of past motion, the current frame, and 5 frames of future motion. This defines our trajectory and is outlined in more detail in Section 2.4.2.3. If we are near the beginning or end of the path and do not have enough frames to sample the motion from, we simply take the first or last frame respectively as the additional frames we need.

In this manner, upon every iteration the PFNN is able to take both input from Maya and it's own output from the previous iteration and create a new input, enabling it to infer the next frame of animation. Once a frame of animation is generated, the relevant information for an animation to be produced is extracted and sent to Maya, at which point the frame is discarded by the PFNN, ensuring that it is not using unnecessary memory. We choose this "relevant" data to be the joint positions of the character in the frame, but this decision eventually causes problems as we try to bind a mesh to our character. The issues and a potential solution to this are discussed further in Section 3.3.5. The process of inferring a frame of animation, sending it to Maya, and inferring the next frame is repeated until the animation is complete.

3.2.5 Animation Generation

Each time the PFNN generates a frame of animation it extracts the relevant data and sends this back to Maya. This means we are receiving a large number of commands and it is logical for us to buffer these before executing them. One reason for this is that it allows Maya to finish receiving responses over it's network port before having to move anything within the scene - if we didn't do this and executed each frame as they came, we could get into a situation where Maya accepts a new command before the previous one has finished executing, causing the animation to generate incorrectly and having a knock-on effect to every subsequent frame. There are ways around this that involve additional checks and a handshaking system set up for our networking, but this adds unnecessary complexity where a buffer is a suitable alternative.

Another important reason to have a buffer is that it allows the user to undo and redo the animation if they want to change something about it. In Maya, every time a command is executed, the user can undo this using the "Undo History" command. If we execute an animation command every time we receive one, i.e. at every frame, then to undo the animation the user would have to undo each command frame-by-frame. This is obviously undesirable for even short animations in the range of tens or hundreds of frames. Additionally to this, the default Maya "History" buffer is only 50 commands long, so for animations longer than this the user would actually lose the ability to remove the animation past the final 50 frames. This would be unacceptable behaviour for our implementation to exhibit. The solution to this problem is a buffer; we buffer every frame of the animation, then at the final frame execute the buffer. The user can undo this "executeBuffer" command just once, allowing them to undo or redo the entire animation with one click. This is because the Undo History takes into account only the highest level command you are calling and not the individual commands called within that, so it can remember a 30,000 frame long animation if wrapped in one execute command, but not if done frame by frame.

We have now completed our pipeline and received an animation automatically generated by the Phase-Functioned Neural Network. The solution works perfectly if we use using just a skeleton, but when we bind a mesh to our character suddenly the animation output no longer looks correct - the underlying skeleton renders correctly, but the mesh deforms horribly. We discuss this problem and it's solution in Section 3.3.4.

3.3 Improvements

This section details the particularly strenuous sections of our implementation, from the mathematically complex to the fiddly. These can be seen as some of the most challenging aspects of the implementation, or those that provide improvements upon the basic system.

3.3.1 Path Sampling

We want to sample points along the curve according to our sampling rate. There exists a Maya command `pointOnCurve` which allows us to retrieve information about a point on a NURBS curve as defined by a parameter value. Recall from Section 2.1.1.2 that a parameter allows us to access a point on a curve. In Maya by default curves are created with uniform parameterisation which means that the parameter value increases by 1.0 at each edit point. That is, the first edit point has parameter 0.0, the second has parameter 1.0, and so on. Decimal parameters represent points of the curve between edit points.

So to access a point on the curve we simply need to find the correct parameter, and this is where our sampling rates are used. We iterate over the total number of frames in the animation and, at each frame, check the desired gait and select the sampling rate assigned to that gait. The parameter at any frame is then the value of the previous parameter plus this sampling distance. We use `pointOnCurve` with our parameter, and now have a set of points along the curve.

However there is a problem with this method: uniform parameterisation does not mean that the *distances* between edit points are uniform. What this means in turn is that we may have a curve with large distances between certain edit points and small distances between others. In general, the higher the curvature of the curve the closer together the edit points are. Imagine we have a curve with three edit points. The

distance between the first and second edit point is 100 units, and the distance between the second and third is 10 units. If we want 10 points along the curve using the parameterisation above, `pointOnCurve` will give us 5 points spread every 20 units along the first span, then 5 points spread every 2 units along the second span. This is not the desired result.

There is a flag we can enable in `pointOnCurve` called `turnOnPercentage` which reparameterises the curve from 0.0 to 1.0, however this still samples the curve in the same way between edit points. One potential solution to this problem would be to redistribute the edit points throughout the curve. This is harder than it seems, as edit points are a by-product of the placement of the control vertices, and to redistribute them you need to rebuild the entire curve. While this can be done via a Maya command the results are not guaranteed to return you a curve with exactly the same curvature, and most of the time the output curve will look slightly different to the input curve. This could be an acceptable solution however we wanted to use the exact path provided by the user and not an approximation to this path.

To overcome this we used the OpenMaya API to access the curve via the `MFnNurbsCurve` class. Within this class there exists a function `findParamFromLength` which allows us to find the exact parameter value of a point based on it's length from the start of the curve. We take our parameter from before and multiply this by the curve length, then use this value in the function to get the correct parameters we need. This new parameter can then be used in `pointOnCurve` as before, with `turnOnPercentage` disabled. A visual example of these solutions is shown in Figure 3.1. On the left we have a curve sampled with our original parameter, and to the right we have one sampled correctly with our improved method. If using the left method then our character would speed up and slow down according to the curvature of the curve, but the right method ensures the character always moves at the appropriate speed. Combined with our path sampling rates we now have accurate locations along the path at which to sample positions, heights, directions and gaits.

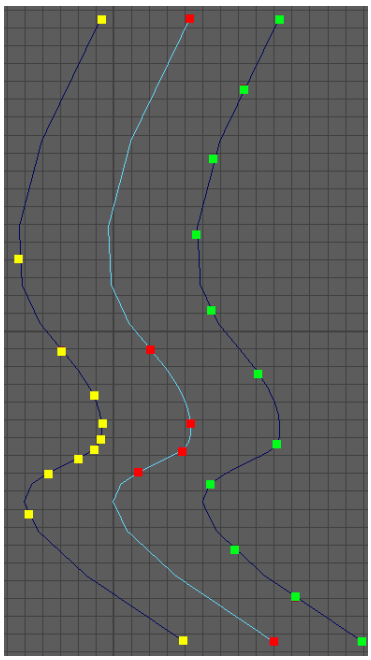


Figure 3.1: A visualisation of curve parameterisation problem. In the middle we have our curve with it's edit points shown in red. Left, we sample the curve at 11 points using our original naive parameter, shown in yellow. Right, shown in green, we sample 11 points using our new parameter.

3.3.2 Height Positions

To retrieve the terrain heights for our path, first we need to retrieve the coordinates of the positions 25cm left and right of a every point on the path. These are the points orthogonal to the curve at the central point, so we can use the command `pointOnCurve` with the flag `normalizedNormal` to get the normal of the curve, $\boldsymbol{n}$, at each central point c, normalised to a length of 1 unit. In Maya the default units are centimeters, so if we multiply this vector by 25 and add/subtract it from our central point c we get our left and right points p and q:

$$p = c + 25\boldsymbol{n}$$
$$q = c - 25\boldsymbol{n}$$

Thus, for each central point on the path, we now have a point 25cm to the left and a point 25cm to the right of it. However there is a problem: we may assume that point p is always the left point, and point q is always the right point, but this is not the case. Depending on the direction of curvature of the curve the normal will project in a different direction. That is, the normal of the curve always points inwards towards the centre of radius of the curve, meaning our point p will always be on the interior of the curve, and q will always be on the exterior. This in turn means that a point p may swap between sides of the curve as it changes direction, and hence needs to be properly classified as left or right of the curve.

To determine which side of a line from $A = (x_1, z_1)$ to $B = (x_2, z_2)$ a point $P = (x, z)$ falls, we compute the following equation:

$$d = (x - x_1)(z_2 - z_1) - (z - z_1)(x_2 - x_1)$$

The sign of d tells us which side of the line the point lies. If $d < 0$ the point lies on one side, if $d > 0$ the point lies on the other, and if $d = 0$ then the point lies on line. To determine which way round these equalities work we calculate d with some known point. This is known colloquially as the determinant method as d is the determinant of column vectors AP and AB concatenated as a 2x2 matrix.

We follow this method for our implementation. Here, we take our line segment as $\boldsymbol{t}$, the tangent to the curve from our central point c. P is then the point p that we wish to classify. In our implementation we find that if $d > 0$ then the point p is to the left of the curve, and hence the point q is to the right of the curve, and vice versa if $d < 0$. With this, we now have the correctly classified left, right and central points for every point along the path, and can use these to calculate the heights of the path.

3.3.3 Height Sampling

Given the (X, Z) coordinates of the left, right and central positions for a sampled point on a path, we want to calculate the height of the terrain under the path at these points. We can imagine projecting the (X, Z) coordinates down onto the terrain and using the Y value at this point as our height. This task is much easier said that done. Maya has no inbuilt way of accessing a point on a mesh given a set of coordinates, and only stores information about a mesh in terms of it's vertices and polygons. As such, we must define a custom workaround for this.

In Section 3.2.3.3 we defined a Barycentric Height sampling method, and here we cover the mathematics behind this solution in more detail. Recall that we have a point (X, Z). We calculated the nearest vertex to this point in our mesh, and now have a set of triangles who all have this vertex as one of their defining vertices. We now define this triangle using barycentric coordinates, check whether our point is within the triangle, and if so we interpolate it's height based on the vertices of the triangle. This process is described as follows.

We observe a triangle defined by three vertices A, B and C. Its edges are defined by the vectors $\boldsymbol{AB}$, $\boldsymbol{BC}$ and $\boldsymbol{CA}$ and the triangle defines a plane in 3D space. If we select vertex A we can describe every other point on this plane relative to A, essentially taking A as our origin. We take $\boldsymbol{AB}$ and $\boldsymbol{AC}$ as our basis vectors that define this new coordinate space. We can now describe any point on the plane as

$$P = A + u(C - A) + v(B - A)$$

where $\boldsymbol{AB} = B - A$, $\boldsymbol{AC} = C - A$ and u and v are scalar parameters, with (u, v) giving our barycentric coordinates for point P. We now rearrange the above equation to give:

$$\begin{aligned} P &= A + u(C - A) + v(B - A) \\ P - A &= u(C - A) + v(B - A) \\ \boldsymbol{AP} &= u\boldsymbol{AC} + v\boldsymbol{AB} \end{aligned}$$

Note we have substituted $C - A$ for $\boldsymbol{AC}$, $B - A$ for $\boldsymbol{AB}$ and $P - A$ for $\boldsymbol{AP}$. We have two unknowns, u and v, so we form a system of linear equations by taking the dot product of both sides with both $\boldsymbol{AC}$ and $\boldsymbol{AB}$:

$$\begin{aligned} \boldsymbol{AP} \cdot \boldsymbol{AC} &= (u\boldsymbol{AC} + v\boldsymbol{AB}) \cdot \boldsymbol{AC} \\ \boldsymbol{AP} \cdot \boldsymbol{AB} &= (u\boldsymbol{AC} + v\boldsymbol{AB}) \cdot \boldsymbol{AB} \end{aligned}$$

The dot product is a linear operator so we can rearrange as follows:

$$\begin{aligned} \boldsymbol{AP} \cdot \boldsymbol{AC} &= u(\boldsymbol{AC} \cdot \boldsymbol{AC}) + v(\boldsymbol{AB} \cdot \boldsymbol{AC}) \\ \boldsymbol{AP} \cdot \boldsymbol{AB} &= u(\boldsymbol{AC} \cdot \boldsymbol{AB}) + v(\boldsymbol{AB} \cdot \boldsymbol{AB}) \end{aligned}$$

To find equations in u and v we can now substitute each equation into the other, giving:

$$u = \frac{(\boldsymbol{AB} \cdot \boldsymbol{AB})(\boldsymbol{AP} \cdot \boldsymbol{AC}) - (\boldsymbol{AB} \cdot \boldsymbol{AC})(\boldsymbol{AP} \cdot \boldsymbol{AB})}{(\boldsymbol{AC} \cdot \boldsymbol{AC})(\boldsymbol{AB} \cdot \boldsymbol{AB}) - (\boldsymbol{AC} \cdot \boldsymbol{AB})(\boldsymbol{AB} \cdot \boldsymbol{AC})}$$
$$v = \frac{(\boldsymbol{AC} \cdot \boldsymbol{AC})(\boldsymbol{AP} \cdot \boldsymbol{AB}) - (\boldsymbol{AC} \cdot \boldsymbol{AB})(\boldsymbol{AP} \cdot \boldsymbol{AC})}{(\boldsymbol{AC} \cdot \boldsymbol{AC})(\boldsymbol{AB} \cdot \boldsymbol{AB}) - (\boldsymbol{AC} \cdot \boldsymbol{AB})(\boldsymbol{AB} \cdot \boldsymbol{AC})}$$

Thus, to find the u and v values for a point P, we need to know the points A, B and C which define the triangle. How do these values relate to our problem of telling whether point P is within the triangle? We use some intuition here: if u or v are less than 0, then we are travelling along our axes backwards and are thus outside our triangle, and if $u + v > 1$ then we have travelled past the outer edge of the triangle and are outside in the other direction. We can check these conditions and tell whether our point lies within the triangle, using our point at (X, Z) as point P and the three vertices of the triangle as A, B and C.

If our point lies within the tested triangle then we can interpolate the height of the point from the heights of the three vertices A, B and C. Plugging our values into our original equation we get:

$$\text{height} = A_y + u(C_y - A_y) + v(B_y - A_y)$$

where A_y is the y component of A, which in our coordinate space corresponds to the height.

With this method we can thus find the height of a (X, Z) point projected onto the terrain. We do this for every left, right and central point along the path and then have our full path heights data to send to the PFNN.

3.3.4 Rotations and The Mesh Problem

Our current implementation uses the joint positions output by the PFNN to move the character's joints to their required place in the world. This solution works when we are viewing our character as just a skeleton but causes issues when we bind a mesh to the character. As can be seen in Figure 3.2, when a character has a mesh attached to it translating the joints causes the mesh to deform in undesirable ways, meaning that while the character's skeleton looks correct the overlying animation of the mesh looks completely wrong. For a video of this please see the supplementary videos in Appendix A. The solution to this is to use joint rotations instead of joint translations. Importantly, we can still translate one joint - the root joint, which is the highest joint in the joint hierarchy - but any other joint should be rotated only.

There are many issues with rotations in 3D space that are faced by those who work in graphics. A major issue is known as *gimbal lock*. In a Euclidean rotational space we have three orthogonal rotation axes, X, Y and Z which operate independently of one another. Gimbal lock occurs when two of these axes rotate in such a way that they overlap with one another exactly, at which point they have been driven into parallel and rotate in sync with one another, causing us to lose a degree of freedom in our rotations.

Gimbal lock is inevitable if we are using Euler angles, which describe the classic notion of X, Y and Z rotations. The problem here is due to the fact that Euler rotations must occur in some order, defined by the person or software using them - in Maya the default order is XYZ, but this differs in different fields. As you progress through the rotations you are affecting the other axes; in the first rotation we affect the X, Y and Z axes; the second rotation affects just the Y and Z axes; then the final rotation affects just the Z axis. This pushes the axes towards each other in undesirable ways, meaning eventually your first and third axes will overlap, at which point we have gimbal lock.

There are however methods of rotating an object that do not incur gimbal lock. Rotations achieved via a *rotation matrix* or *quaternions* are two such methods. A rotation matrix provies a way of encoding Euler rotations that does not incur gimbal lock, and can be combined with scale and translation matrices to form a 4x4 transformation matrix known as an affine transformation matrix. Quaternions are a given as a set of four numbers x, y, z and w and can be used to define rotations of a given angle about a rotation axis (see Section 2.5.4 for the mathematics behind this).

We now consider an implementation to use joint rotations to move the character's joints in Maya. Along with joint positions, which we are using in our current implementation, the PFNN outputs joint angles.

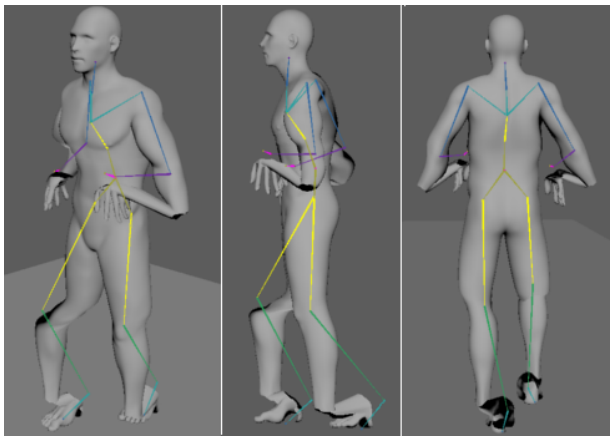


Figure 3.2: In these images, translating the joints has resulted in the character's mesh being deformed incorrectly, although the skeleton exhibits the correct pose.

These are output local to the character's root transform and are expressed using the exponential map [14], which like rotation matrices or quaternions is an alternative way of expressing rotations. The mathematics behind the exponential map are omitted from this thesis but it is influenced by quaternions and as such an exponential map expression can be easily converted into quaternion form. We do this conversion, then a conversion from quaternion to rotation matrix. In this form we can transform our rotations from local space to world space by pre-multiplying with the rotation matrix of the character's root transform. We then convert this world space rotation matrix back to a quaternion. This is purely an implementation choice, and either the rotation matrix or quaternion could be used in Maya; we choose a quaternion to reduce the networking load, as each quaternion requires only four numbers to be expressed, whereas a rotation matrix requires nine. The quaternions are then sent to Maya and are processed utilising the OpenMaya API, and at each frame we now key the joint's rotation instead of it's translation.

We now assume we are complete. We have a working pipeline from the PFNN to Maya that sends and correctly processes rotations as quaternions. However there is a problem with this implementation - it depends on how the axes of the character's joints are oriented in order to rotate the joints appropriately in Maya. That is, we must know how the character's joints were oriented in the training data in order to implement the solution fully. Unfortunately the author has been unable to decipher this information as it is not specified anywhere within the paper or the demo code. We have experimented with seemingly every possible combination of axis orientations however cannot find the one which gives the correct animation. Whilst we realise that this may be indicitative of an issue with the implementation of extracting the rotations from the PFNN, the animation generated is cyclic and not far off from the correct animation so we are reasonably convinced that the implementation of the PFNN to Maya pipeline is correct and that the orientation axes are the problem, particularly as the animation output changes greatly when experimenting with different axes.

3.3.5 HumanIK - A Generalised Skeleton

One problem with the PFNN is that is dependent on a very specific skeleton hierarchy with 31 rotational joints in a set order, meaning that our solution is also very specialised. This is a problem faced with any implementation of any solution, as there is not one standardised skeleton used throughout the industry and animators can define a skeleton as freely as they wish. There is however one commonly used skeleton hierachy that is built in to Maya, to the point that it may be considered the standard by which to work with. This is the HumanIK skeleton. HumanIK is Autodesk's attempt at standardisation and the skeleton comes with many useful inbuilt features such as optional IK rigging and control curves. The skeleton itself can be edited to suit the animator's needs, for example bones can be added to the spine to create greater curvature and flexibility, or to the hands to represent each finger bone to a number of degrees of accuracy. In this way we can specify a HumanIK skeleton that correlates closely to the PFNN skeleton.

We define a mapping between the PFNN's 31 joint hierachy and a HumanIK hierarchy with 21 joints,

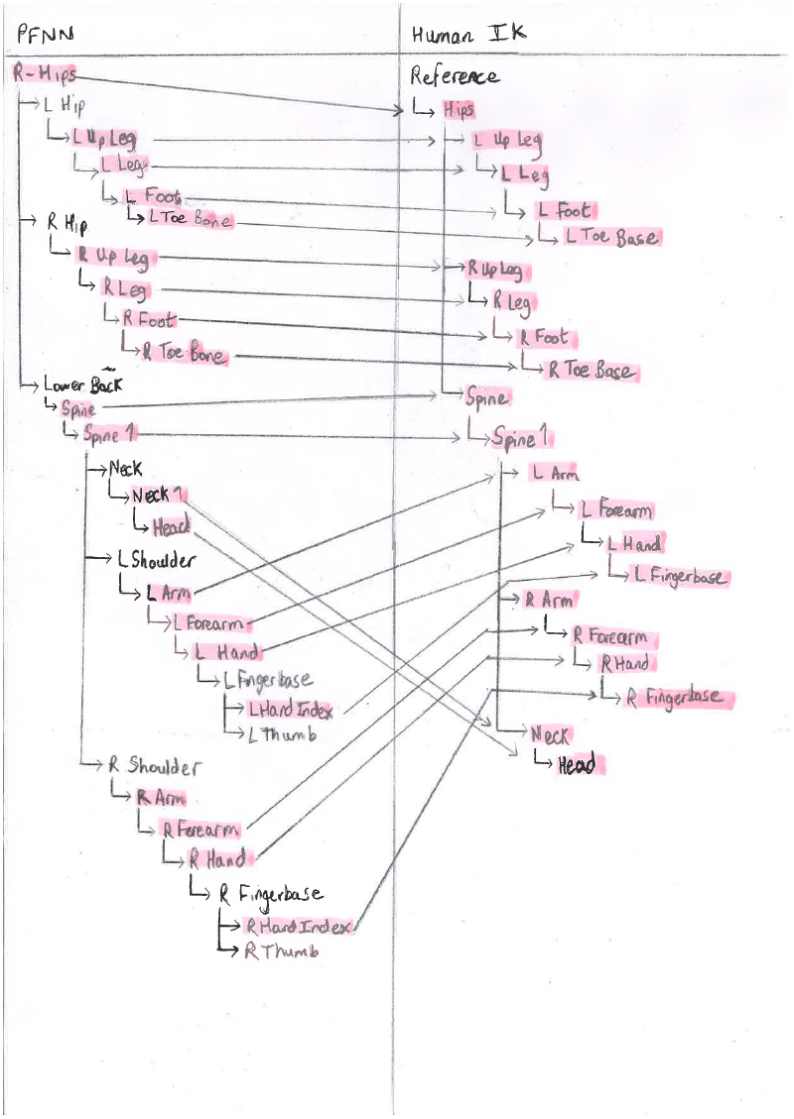


Figure 3.3: A mapping between the PFNN skeleton hierarchy and the HumanIK skeleton hierarchy.

seen in Figure 3.3. This mapping is possible due to a number of "empty" joints in the PFNN, which are positioned exactly at their parent and hence whose locations can be calculated simply by taking the location of the parent joint. These empty joints are required as input to the PFNN but actually have no real influence on the output generated - when generating the training data from the motion capture dataset these joints were effectively masked out by being weighted incredibly lowly in comparison to the more significant joints. As such their existence in the PFNN is really just an artifact of the motion capture markers in the motion capture dataset. Hence, when mapping from our HumanIK skeleton to our PFNN skeleton, we can specify the positions of the empty joints as exactly the position of their parent, and when mapping the output from the PFNN back to the HumanIK skeleton we can simply ignore these joints as they have no effect on the final animation. We must also reorder the joints as they are required in a specific order for the PFNN, but this is a trivial task of array indexing.

The implementation of the mapping described here is left to future work, however one can imagine a GUI where the user clicks a button, "Create Skeleton", which creates the 21 joint HumanIK skeleton defined here and behind-the-scenes maps the skeleton onto the PFNN skeleton.

Chapter 4

Critical Evaluation

In this chapter we evaluate our implementation. First we discuss our original aims and our progress towards these. Then we perform a comprehensive set of tests, discussing the success and failure cases of the system. We then discuss some limitations of our implementation, and finally we look at some potential future work.

4.1 Aims Revisited

We first recount the aims for our project as outlined in Chapter 1 and evaulate the progress we made towards these. These goals were as follows.

To investigate the state-of-the-art in automating animation In Section 2.3 we detailed a number of solutions to the problem of automating animation, and discussed the pros and cons of each method. In Section 2.4 we looked in depth at the Phase-Functioned Neural Network, our chosen solution for our implemenatation. As such this goal was completed.

To integrate an existing solution into the animation software Maya This goal covered the integration of the Phase-Functioned Neural Network and Maya. This integration was completed, with most of the features of the PFNN being used in our implementation. Two noticeable differences were the lack of the "jump" and "bump" gaits; in a further implementation of the project these could be included, along with some method of implementing them in Maya which is a non-trivial task.

To alter the implementation to better fit the target domain Altering the implementation of the PFNN to suit the context of an animation software required a number of bespoke improvements to be made. These were covered in Section 3.3 and allowed the quality of the animation produced by the system to be higher.

4.2 Testing

In this section we thoroughly test our solution. We first do comprehensive functional testing of the system, investigating the success and failure cases, and discussing the reasons behind these. We then test and evaluate two specific aspects of the solution, looking specifically at our implementation of height sampling and evaluating the speed vs. animation quality of using different methods of calculating the phase function at runtime in the PFNN.

4.2.1 Functional Testing

In this section we complete a number of functionality tests of the system and discuss a number of success and failure cases.

4.2.1.1 Different Path Styles

For these tests we varied the path that the character was directed to walk along. For all tests we used a flat terrain and the character was instructed to walk along the path, beginning from a neutral rest pose. We ran the system on a wide variety of path types, including a stright path, paths with different amounts of curvature, paths with mixtures of curvature, a spiral path, a circular path, a square path and a zigzag path. The results of some of these experiments are in Appendix A.

In general, the system performed well across most of the paths, however there were clear cases in which performance was improved. Paths with tight curves performed worse than those with wider curves, and this becomes even more apparent with paths that have precise angles in them such as the zigzag path and the square path. In these paths, when the character comes to an angle, instead of blending the motion and veering slightly off the path to better traverse around it, the character instead sharply and instantly turns over the course of one frame. This of course provides unrealistic animation as there is no continuity between the character's facing direction in one frame and the next.

The reason for this is down to the implementation of the PFNN. In the original demo code, the predicted trajectories as output by the PFNN are blended with the trajectory as defined by the user controlling the character. This allows the motion to be responsive to the user input but gives the animation time to blend from it's expected future trajectory into the new trajectory as defined by the user. When altering the PFNN code for our system the decision was made that this blending was not needed as there would be no interactive input, with the data all being provided beforehand in an offline manner. However this did not account for the secondary benefit of the blending, which not only blends between predicted and given input, but also blends any sharp turns into more realistic looking motion. Without the blending, the character simply follows exactly the direction of the path at any point, and hence the instant turning when faced with a sharp angle.

4.2.1.2 Different Gaits

The next set of experiments tested the different gaits we chose to implement. The first experiments were ran on a flat terrain along a straight path with a single gait per path. We observed high animation quality for all gaits, arguably due to the specialised path sampling rates as discussed in Section 3.2.3.1. These were chosen specifically to provide high quality animation so this result is to be expected. We then ran a number of other experiments on different terrain types and continued to observe high quality animation on a number of terrains, with common issues for certain terrain types irregardless of gait, which will be discussed below. Here, high quality animation was defined as that with minimal to no foot slippage, and no gliding.

We also ran a number of experiments testing multiple gaits on one path. These produced some interesting results, seen in Appendix A. We found that there was a distinct seam between animations of different gaits. This does not occur in the PFNN demo, and so we looked deeper into the code. Whilst the gait variables are specified in the paper as a six-dimensional one-hot vector, meaning every gait either has a value of 0 or 1, the PFNN code implemented interpolation between the gaits. This occurs extremely quickly, within around 10 frames, but allows enough time for the motion transitions to look blended. Thus as future work our system would also include some interpolation between the gaits when a change in gait is detected, which would provide a smooth blended motion output.

4.2.1.3 Different Terrain Types

These experiments tested the system's output on a number of different terrains. A wide variety of terrains were generated, including: flat, flat at a slight incline,flat at a steep incline, and a variety of

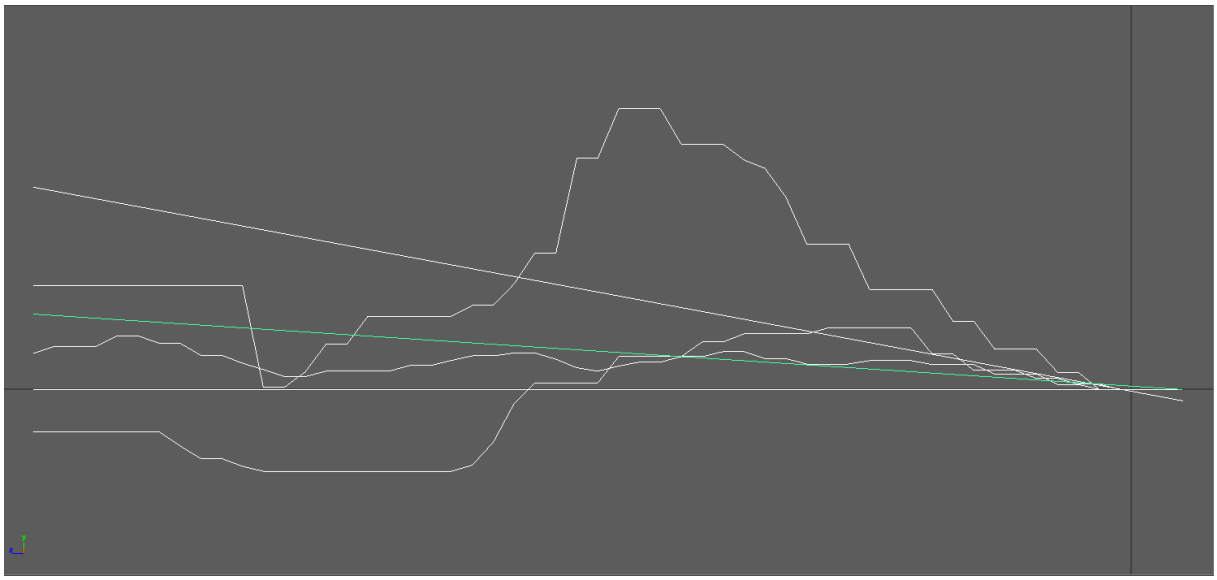


Figure 4.1: A number of different terrain cross-sections

rough terrains from shallow to steep rockiness. The terrains were tested with a straight path for both walking and jogging. See Figure 4.1 for some examples.

In general, the flatter the ground the better the animation output, as can be expected. The animation on the flat and mild incline terrains was arguably perfect. On the steep incline the character appeared to not realise the terrain was smooth and occasionally attempted to "climb" as if scaling a rocky terrain. This is due to the training data. When the training data is pre-processed it is fitted to a number of terrain patches generated from video games, however the vast majority of these terrain patches feature rocky terrain, and no terrain that is flat but at an incline. Hence, when faced with any sort of change in height, even if the change is smooth the PFNN assumes it is seeing a rock of some sort and produces animation as such. This was likely not thought about by Holden et al. as the PFNN was originally designed for video games where the terrains are usually more natural and rocky, hence for their context this pre-processing made sense. A fix for this would be to use a different method of pre-processing that included what could be considered as more "urban" terrains, with sharp edges and flat surfaces.

We then tested a number of rocky terrains and again the results were similar. The animation output on shallow rocky terrain was of a high quality, whereas as the terrain got steeper, with sharper inclines and drops, the animation quality suffered. Figure 4.2 shows an example of this, where the character's feet end up going through the terrain. This happens relatively consistently with sudden drops from great heights, which is likely to do with the fact that any training data would have included drops from small heights but not from large heights. This means that the PFNN overcompensates during these large drops and pushes the character far through the floor.

4.2.1.4 Different Initial Poses

The character was tested from a number of different initial poses, such as a resting pose, a pose mid-way through a jogging animation, a "T-pose" (where the character has their arms and legs outstretched in a "T" shape) and a bent over pose. In each case, within around 20 frames the animation corrects the character's pose to one that would be seen when in locomotion and the pose stabilises during the locomotion animation.

4.2.1.5 Robustness and Edge Case Testing

The system works irregardless of the position of the terrain, character and path in the world space, meaning these can hold negative values and be in any quadrant of the world and the animation produced is the same. The path can be any height above the terrain and does not need to lay directly on the

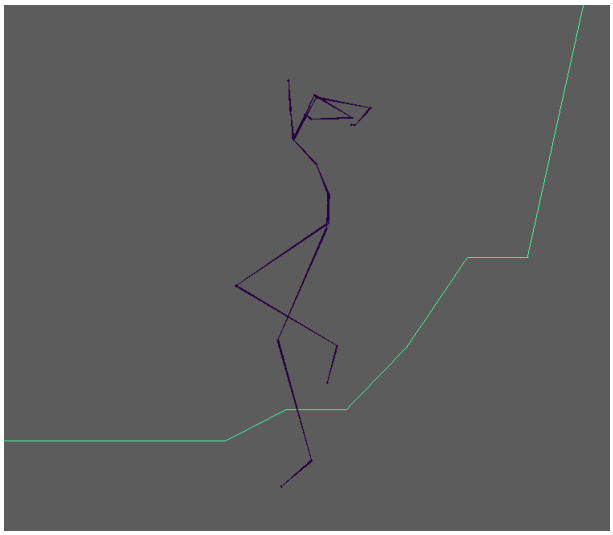


Figure 4.2

terrain, and can travel in any direction and will still work as desired.

An interesting edge case is what happens if the path starts somewhere away from the character. In our implementation we assume that the user will draw the path from where the character stands, but we test the case where they draw the path from somewhere else in the world space. Here, the PFNN essentially teleports the character to the start of the path, and within 5-10 frames the animation stabilises and the character traverses the path as normal. The teleportation to the start of the path takes one or two frames, and produces an interesting, if not desirable, effect. The solution to this is to force the user to draw paths that start from the character's position; in fact, it is best if the path starts from the character's root transform as this is how the training data or the PFNN was defined. Code has been written in Maya to identify the location of the root transform, although it is not currently being used - Section 4.4.4 covers a potential implementation of a Path Drawing Tool that would use this.

Another case of interest is what happens when the character is scaled to a size not used by the PFNN. Similarly to the teleportation problem, here the PFNN quickly corrects the scale of the character to the scale it was trained on, and after around 5-10 frames of animation the character is now at a different scale that it initially was and the animation stabilises around this scale. This is caused by the PFNN overfitting the training data due to it's reasonably specific training set. This is also caused in part by our implementation, as we are using the output joint positions to animate the character. If we used the joint rotations then the character would be able to stay at their appropriate scale. However, if using joint rotations we would still need to translate the root joint, and the PFNN would have "corrected" this to be at a different scale and hence would not provide the right output. An alternative way around this problem would be to create some scaling function that takes the joint positions from Maya and scales them up to the PFNN's skeleton, then takes the output from the PFNN and scales these back to the initial size in Maya. An implementation of this is left for future work.

4.2.2 Height sampling

In this section we evaluate and compare two methods of height sampling, the barycentric method as used in our final implementation, and the naive method, both originally described in Section 3.2.3.3. We use a number of different testing terrains, and for each experiment our character begins in a neutral position and is instructed to walk or jog along a straight path. We then run each test environment with both the barycentric and naive height sampling methods, and inspect the animation produced. First we perform a visual inspection, noting any points of particularly bad animation, such as feet clipping through the floor. We then run some numerical experiments for a quantitative comparison of the two methods. We created a number of testing terrains as follows: flat, shallow incline, steep incline, shallow, rough, steep, shallow low poly, rough low poly, steep low poly.

First we visually inspect our animation outputs. In general, we see that the animation produced by the barycentric method is of a higher quality, with smoother looking motions. The nearest vertex animations suffer from "stepping", which is when they change from being nearest one vertex to another and thus immediately adapt to the height of the new vertex, jumping up or down in the world suddenly. Neither animation is perfect however, with particular trouble when the terrain gets particularly steep - traversing up or down a steep surface leads to low quality animation.

For our second comparison we take the animation produced and, at every frame, calculate the difference in height between the terrain and the character's heel and toe joints on either foot. From this data we can generate a number of graphs as seen in Figure 4.3. These graphs show a number of interesting things. Graph (a) shows normal locomotion on a flat surface, where the character has a regular cyclic motion.

Graph (b) shows the difference between the height sampling methods when tested on a single-polygon plane at an incline. Here, the middle set of curves represents the barycentric method, and we see that the error in height between the joints and the terrain are cyclic and consistent, as in Graph (a). More interesting is the other set of curves, which represent the barycentric method. We see that the error increases negatively until the character is far below the terrain, then suddenly over the course of a frame the error becomes very high above the terrain - this represents the point at which the character is reading it's height from the vertex at the end of the plane. We can also see that just before and just after the jump the cyclic motion changes slightly, which correlates to the character attempting to jump up at the start and overcompensate for the sudden increase in height at the end. See Appendix A for a video of this.

We can further see that the barycentric method is an improvement on the nearest vertex method from Graphs (c) - (f). These each show an individual joint travelling over the steep low-poly terrain. Low-poly terrains exemplify when the nearest vertex method works worst. We can see that the barycentric method in general provides smoother (less jagged) curves than the vertex method, which correlates to smoother looking motion. The error between the joints and the terrain is also generally smaller with the barycentric method, meaning closer to the 0 error line. In the vertex method we can see many examples of points at which the character suddenly changes in height, indicated by vertical drops or inclines - these are the points at which the vertex height being read changes, and they result in jarring motion. Videos of this can be found in Appendix A. As such, it is clear that the barycentric method provides better animation than the nearest vertex method.

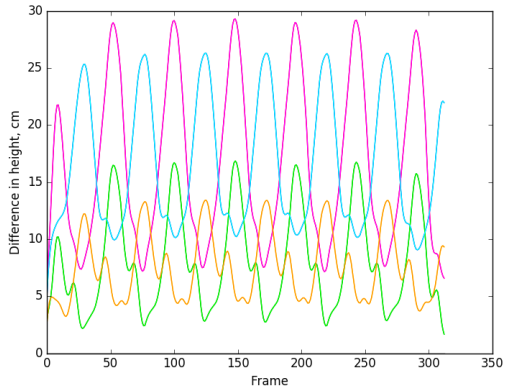
4.2.3 PFNN Mode

As discussed in Section 2.4.3.2, the PFNN can be run in one of three "modes", with each mode referring to a different method by which the phase function can be pre-computed and evaluated at runtime. The first method is to pre-compute the phase function at 50 intervals and use the nearest value to the phase at runtime. The second option is to pre-compute 10 intervals and, at runtime, to linearly interpolate between these. The third option is to compute the full cubic Catmull-Rom spline at runtime. These are the constant, linear and cubic modes respectively. We have thus far used the constant mode throughout our experiments, but we decided to experiment with the different modes in an attempt to distinguish two things:

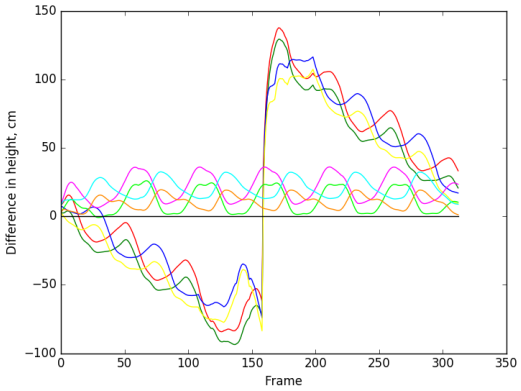
- Which mode runs fastest, and by a factor of how much?
- Is there a significant difference in animation quality between the modes?

The answers to these questions could motivate us to use a different mode in our implementation.

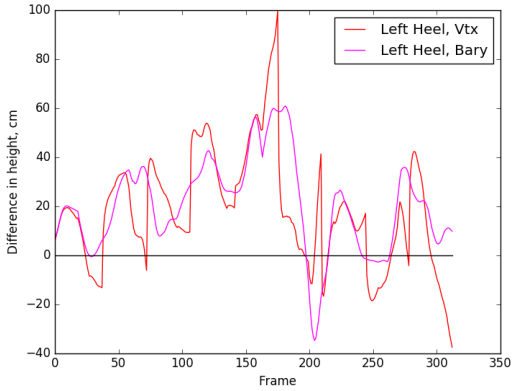
We first experimented with the runtime of the modes. This question can be split further into two: the time taken to initialise the PFNN due to the pre-computation of the phase function, and the time taken to process an animation request. We ran $n = 10$ experiments per mode and generated the data given in Table 4.1. From this table we can note a few points of interest. First, as expected the cubic mode takes the shortest time to start up, requiring no pre-computation of the phase function. However it is interesting to note that, although constant mode requires more computations than linear mode, on average the start up time was marginally faster, albeit by a small margin. It is more interesting to note the average processing time for each mode, as this is arguably more important to our end-user and hence



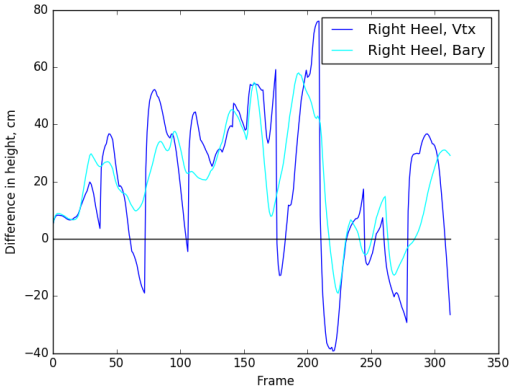
(a) All Joints, Flat



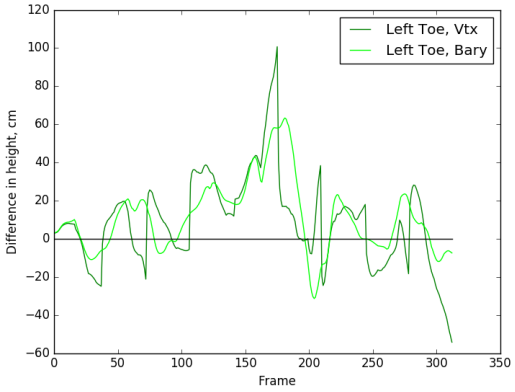
(b) All Joints, Incline



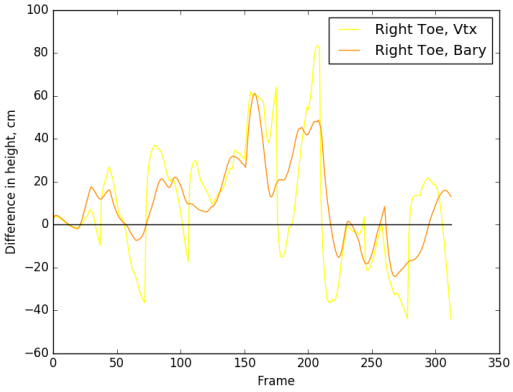
(c) Left Heel, Steep LowPoly Terrain



(d) Right Heel, Steep LowPoly Terrain



(e) Left Toe, Steep LowPoly Terrain



(f) Right Toe, Steep LowPoly Terrain

Figure 4.3: These graphs show the difference in height between the character's foot joints and the terrain. Each graph shows both height sampling methods, with "Vtx" referring to the nearest vertex method and "Bary" referring to the barycentric method. For each of these experiments the character was directed to jog along a straight path.

Mode	Average Start-Up Time	Average Processing Time (per frame)
Constant	3.341s	0.0061s
Linear	3.377s	0.0169s
Cubic	0.268s	0.0696s

Table 4.1: A comparison of start-up and processing times for the various modes of the PFNN

Figure 4.4: Testing the three modes of the PFNN

should have more influence over our decision. We see here that constant mode is ten times faster than cubic mode, and 2.77 times faster than linear. This means an animation of 1000 frames would take on average 6.1 seconds using constant mode, but 69.6 seconds, i.e. over a minute, using cubic mode. It is likely that this decrease in speed would make the processing time unacceptable for an end-user, particularly on a long animation.

We next consider the quality of the animation produced by running the PFNN in each mode. To do this we create three characters in the same scene and animate each via a different mode. We then visually inspect the animations ensuring each is of good quality on it's own, then compare the animations produced between the three. See Appendix A for videos of these experiments.

Each of the three modes produces animation of a high quality, and when played separately there is no significant distinction between the three. When imposed on top of each other we begin to see that the animation produced by each is slightly different. The animations slowly become out of phase with each other, although at such a small rate that even at 1000 frames they are only slightly out of sync. In Figure 4.4 we see a frame of the animation and the positions of the skeletons, with the blue skeleton's motion generated by the constant PFNN, the yellow skeleton's motion generated by the linear PFNN, and the pink skeleton's motion generated by the cubic PFNN. Interestingly, the skeleton which is most out of phase with the others is the linear skeleton, an unexpected result as one would assume the cubic and constant would have the most variation from one another with linear falling somewhere in the middle. This is likely due to the piecewise linear interpolation used by the linear mode, which may be too naive of a method when evaluating a complex Catmull-Rom spline. For the PFNN implementation it seems that linear interpolation of the phase function would require more sample points to interpolate between for similar results to the other two methods.

The visual difference between the constant and the cubic modes is marginal at best, however the constant mode requires much less computation time during runtime, by a factor of 10. It's start-up time is longer than for cubic mode however as this is a server-side time cost and not one that affects the user it was considered an acceptable cost. With the aim of keeping the runtime for the user as low as possible, the

decision to continue to run the PFNN in constant mode was made.

4.3 Limitations

There were a number of limitations with our project, which we will now discuss.

The strict nature of the inputs and outputs to the PFNN made this network architecture useful perfect for a small set of animations. For example, we require an exact skeleton hierarchy and a framerate of 60fps for the animation provided to be valid. This means that our final system is lacking in generalisation, and is not a fully accessible solution that works in all circumstances. The PFNN paper postulates that the network is best used in interactive settings and this appears to be true, if only for the fact that, given a custom-built setting the PFNN will perform at it's best, whereas when we have tried to integrate the network with Maya we have had to make a fair few exceptions. In general animators have a great amount of variation in the way they choose to animate and so a more general solution would have been preferred. Some ideas with regards to this are described in Sections 3.3.5 and 4.4.2. A different choice of neural network would of course have impacted these limitations, but would also have provided a different quality of animation.

Another limitation of our system is that characters have no way of detecting one another. This is something done by AI-driven characters and that is invaluable to a production house, and so would be a valuable addition to our software. Alternatively to developing this ourselves, our system could be integrated with one of these softwares and a hybrid solution that utilises their character detection solution with our improved animation technique could be created.

Our implementation uses OpenMaya to list the vertices in our terrain mesh when calculating the terrain heights. In a mesh with a lot of polygons this has the potential to be time and resource-expensive. One way to combat this would be to port the scripts to a C++ plug-in, as the C++ API is around 10x faster to the Python API. This would come under future work.

4.4 Future Work

4.4.1 Inverse Kinematics for Terrain Clipping

The most obvious choice for future work would be the transition from forward kinemantics to inverse kinematics. This would combat a problem with our implementation and a fundamental issue with the PFNN itself, which is that the network allows the animation output to include animation where the feet of the character are below the terrain, known as "clipping" the terrain. From Figure 4.3 we can see that there are numerous cases where the character clips the terrain, identifiable as the points where the error between a joint and the terrain is negative. There are a few naive approaches to solving this problem, but they would not be suitable for our end goal. The first could be to translate the clipped joints directly upwards until they are above the terrain by some pre-defined amount; this however involves direct translation of a joint instead of rotation, which causes problems with regards to the character's mesh as described in Section 3.3.4. Another naive method could be to translate the entire character upwards via the root joint, which avoids the previous issue as the root joint is able to be translated. This is a method often used by 2D video games to avoid terrain clipping. However this method has it's own issues in that it will cause the animation to lose continuity over each frame, causing choppy upwards motions on frames where the character has to be translated.

The solution to this is inverse kinematics. Recall from Section 2.1.3 that inverse kinematics facilitates goal-directed movement, allowing an animator to define a goal pose and then calculating the appropriate joint rotations to reach this pose. Integrating inverse kinematics with our solution would allow us to implement the animation output from the PFNN in a forward kinematics style, then move the IK handle of any clipped joints above the ground, at which point the IK solver would rotate the character's knee and hip joints to facilitate and compensate for this movement. Inverse kinematics is well integrated into Maya and particularly with the HumanIK skeleton discussed in Section 3.3.5 and hence the basis for integrating this with our model depends mostly on completing the work on joint rotations as discussed

in Section 3.3.4.

4.4.2 Generalisation

There are some improvements that can be made to our system with regards to creating a generalised solution that works despite a number of possible changed variables. In Section 4.2.1 we discussed a number of situations in which the systems works or, more importantly, fails. For our solution to be generally applicable to most animator's needs we need to approach and fix these failures.

One such issue is the problem of scaling, where the character's skeleton, if not at the scale used to train the PFNN, is transformed to be of this scale within 5-10 frames of output animation. As discussed in Section 4.2.1 the solution to this problem lies in animating our character's joint rotations as opposed to joint translations.

Another potential feature is a choice of framerate for the final animation. The PFNN takes inputs and generates outputs at a rate of 60fps, however it is feasible a user may want a different frame rate for their animation - for example, films are commonly shot at 24fps. Changing the framerate may seem like a simple solution of upsampling and downsampling as appropriate, for example if we desire a framerate of 30fps we could sample the path twice as much to get 60fps inputs and downsample the output by half to get 30fps outputs. However, this becomes a larger problem when we want framerates of higher than 60fps. A possible solution to this would be to interpolate the outputs of the PFNN, or to animate every other frame instead of every frame. Here experimentation would be required to find the best solution.

4.4.3 Animating with Control Curves

One design decision we made in Section 3.1.1 was to animate (i.e. key) the joints of the character's skeleton. This was chosen due to the direct correlation with the output of the PFNN, which outputs an animation in terms of joint positions, velocities and rotations. In Section 2.1.2.2 we introduce control curves, an addition to a character rig that abstracts the control of a joint away from the joint itself, usually by parenting the joint to a NURBS curve. This curve is then transformed and keyed, as opposed to the joint itself. Control curves are commonly used in industry to ensure an animator does not mistakenly edit the skeleton in undesirable or unchangeable ways, causing issues in the production pipeline. As such, it would be useful in terms of industrial application for our solution to allow an animator to choose between keying the joints themselves or keying the control curves of the joints.

A potential method for implementing this is described as follows. At each frame of the PFNN, we are given future joint positions and rotations and can move the joints appropriately according to this output. Currently we move the joints themselves then key these attributes. Ideally, we want to move the control curve of a joint so that the final position of the joint is as defined by the PFNN output, but achieved through only transformations on the control curve and not the joint itself. The two important transformations for us to consider here are rotation and translation. For the first, we can assume that a rotation of angle x for the joint will be identical to a rotation of angle x for the control curve - this is basic geometry. Translation however provides us with a non-trivial problem, due to the fact that control curves are entirely user-defined. There is a somewhat standard method of creating control curves where a curve is a circle or ellipses directly aligning with and surrounding the joint. In this case, we can assume that the centre of the control curve aligns directly with the centre of the joint, and can translate the control curve as we would translate our joint. However it is entirely possible that the user defines control curves that are not of this standardised format, for example they could prefer to have curves that are away from the skeleton altogether. This would make animating the control curves much harder as finding the correct position for the curve that places the joint in the right position would be a large task. For this reason, a generalised implementation of this is left to future work.

4.4.4 Path Drawer Tool

A useful feature for our scripts would be a path drawing tool. This would be a simple tool that aids a user in drawing a path that will provide them with high quality animation. The main two features of

this tool would be to force the path to start at the character's root transform, calculated from their hips and shoulder positions, and to ensure that the path drawn is flat with no variation in Y. These small features ensure a better quality of animation. In the scripts we have already implemented the code to calculate the character's root transform.

Chapter 5

Conclusion

This project has culminated in a high quality and efficient solution to the problem of automating locomotion animation within the popular and industry-standard animation software Maya. This was done by utilising a modern deep learning architecture known as PFNN which exploited the phasic nature of walking. This architecture was implemented as a cloud-based solution in order to be robust and practical as a business venture. This was a challenging project as the architecture was not designed with the domain of Maya in mind. To adapt the architecture, the following research problems had to be solved:

- Finding an appropriate rate at which to sample a path based on a variable user input of the gait of a character
- Crafting a method of sampling points from a polygonal mesh surface in Maya
- Adapting the implementation of the neural network to allow it to both use it's outputs autoregressively whilst taking a static, offline (provided fully at runtime) input from Maya
- Generating an appropriate animation in Maya based on the output from the neural network

A thorough analysis has been undertaken into the effectiveness of the solution provided within this project. The performance of the solution has been evaluated and shown to be both a visually and analytically high quality answer to the problem of character locomotion. Bespoke solutions to the aforementioned problems yield positive results, such as:

- Improved height sampling led to a higher quality animation output as the error between the character's feet and the terrain was minimised
- A large variety of successful testing environments for the animation, such that the solution has proved to be robust throughout a number of tests
- A fast solution, generating frames of animation at 0.0061s per frame

In conclusion, this project has investigated the problem of automating character locomotion and has provided a practical and high-quality solution that incorporates state-of-the-art research in deep learning with a focus on the commercial viability of a hypothesised product through the use of cloud computing.

Bibliography

[1] Autodesk, Inc. Autodesk maya 2018. https://www.autodesk.co.uk/products/maya/overview.

[2] Basefount Software Ltd. Miarmy. http://www.basefount.com/miarmy-70.html.

[3] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

[4] J. Thingvold Biovision. Biovision BVH. https://research.cs.wisc.edu/graphics/Courses/cs-838-1999/Jeff/BVH.html, 1999. Accessed: 2019-02-12.

[5] Edwin Catmull and Raphael Rom. A class of local interpolating splines. In *Computer aided geometric design*, pages 317–326. Elsevier, 1974.

[6] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[7] Teunis Cloete and Cornie Scheffer. Benchmarking of a Full-Body Inertial Motion Caputre System for Clinical Gait Analysis. In *2008 30th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 4579–4582. IEEE, 2008.

[8] Motus Digital. MoCap Online. http://mocap.cs.cmu.edu/, 2019. Accessed: 2019-03-20.

[9] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann series in interactive 3D technology. Taylor & Francis, 2004.

[10] Fatima. walktool: a tool for rigging and animating walking characters. https://www.highend3d.com/maya/downloads/character-rigs/c/walktool-a-tool-for-rigging-animating-walking-characters-for-maya.

[11] K. Fragkiadaki, S. Levine, P. Felsen, and J. Malik. Recurrent network models for human dynamics. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 4346–4354, Dec 2015.

[12] Sreya Francis. Phase-functioned neural networks for character control, github repo. https://github.com/sreyafrancis/PFNN, 2017.

[13] Golaem. Golaem crowd. http://golaem.com/.

[14] F Sebastian Grassia. Practical parameterization of rotations using the exponential map. *Journal of graphics tools*, 3(3):29–48, 1998.

[15] G. Hendricks. *Eadweard Muybridge: The Father of the Motion Picture*. Dover Photography Collections. Dover, 2001.

[16] Daniel Holden, Taku Komura, and Jun Saito. Phase-Functioned Neural Networks for Character Control. *ACM Transactions on Graphics (TOG)*, 36(4):42, 2017.

[17] Autodesk Inc. Autodesk Maya 2018: Maya API Introduction. http://help.autodesk.com/view/MAYAUL/2018/ENU/?guid=__files_API_Introduction_htm, 2018. Accessed: 2019-04-30.

[18] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. Third edition, September 2011.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[20] Jürgen Lind. Massive: Software engineering for multiagent systems. 1999.

[21] R. Memisevic. Learning to relate images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1829–1846, Aug 2013.

[22] Thomas B Moeslund and Erik Granum. A Survey of Computer Vision-Based Human Motion Capture. *Computer vision and image understanding*, 81(3):231–268, 2001.

[23] NeoReel inc. Moclip. https://apps.autodesk.com/MAYA/en/Detail/Index?id=1536524026079954339&appLang=en&os=Win64.

[24] Nils Diefenbach. Autowalk for maya. https://www.highend3d.com/maya/script/autowalk-for-maya.

[25] nlohmann. nlohmann_json, github repo. https://github.com/nlohmann/json, 2019.

[26] Xue Bin Peng, Glen Berseth, Kangkang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Trans. Graph.*, 36(4):41:1–41:13, July 2017.

[27] Jacquelin Perry, Jon R Davids, et al. Gait Analysis: Normal and Pathological Function. *Journal of Pediatric Orthopaedics*, 12(6):815, 1992.

[28] Python Software Foundation. Python. https://www.python.org/.

[29] Ready-Motion.com. Ready-Motion. https://www.ready-motion.com/, 2019. Accessed: 2019-03-20.

[30] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[31] Graham W. Taylor and Geoffrey E. Hinton. Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1025–1032, New York, NY, USA, 2009. ACM.

[32] Carnegie Mellon University. CMU Graphics Lab Motion Capture Database. http://mocap.cs.cmu.edu/, 2019. Accessed: 2019-05-08.

[33] Ziva Dynamics. Ziva vfx. https://zivadynamics.com/ziva-vfx.

Appendix A

Supplementary Videos

This Appendix contains links to some supplementary videos for my dissertation. These explain better than words the animations the system produces.

The videos are in the following Google Drive folder: https://drive.google.com/drive/folders/1fxyfG2KJZksDpBAxe8l9u_PKlUjSoW0b?usp=sharing