DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2017

Planning bipedal character locomotion in virtual worlds

ARON GRANBERG

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Planning bipedal character locomotion in virtual worlds

ARON GRANBERG

Bachelor in Engineering Physics
Date: May 20, 2017
Supervisor: Christopher Peters
Examiner: Martin Viklund
Swedish title: Planering av tvåfots-karaktärers förflyttning i virtuella världar
School of Computer Science and Communication

Abstract

This paper presents a locomotion system suitable for interactive use that can plan realistic paths for small numbers of bipedal characters in virtual worlds. Earlier approaches are extended by allowing animations to be arbitrarily blended to increase the range of motions that the character can produce and our system also achieves greater performance compared to the earlier approaches. The system uses a graph of valid footprints in the world in which is searched for a path that the character should traverse. The resulting sequence of footprints are smoothed and refined to make them more similar to the character's original animations. To make the motion smoother the curvature and other parameters of the path are estimated and those estimates are used to interpolate between different sets of similar animation clips. As the system is based on footprints it allows characters to navigate even across regions which are not directly connected, for example by jumping over the gaps between disconnected regions. We have implemented the system in C# using the Unity Game Engine and we evaluate it by making the character perform various actions such as walking, running and jumping and study the visual result.

Accompanying material can be found at `http://arongranberg.com/research/thesis2017`.

Sammanfattning

Detta arbete presenterar ett system för att beräkna rörelsebanor för virtuella ett litet antal tvåfotskaraktärer tillräckligt snabbt för att kunna användas i interaktiva sammanhang. Tidigare tillvägagångssätt utvidgas för att hantera animationer som interpoleras godtyckligt och på detta sätt kunna utöka vidden av rörelser som en karaktär kan utföra. Dessutom så minskar vi beräkningskraften som krävs jämfört med tidigare system. Systemet baseras på fotsteg och bygger upp en graf av dessa i vilken en väg för karaktären söks. Den resulterande sekvensen av fotsteg är utslätad och förfinad för att göra de mer lika de ursprungliga animationerna. För att göra rörelsen mjukare så uppskattar vi kurvaturen samt andra parametrar av rörelsebanan och använder dem för att interpolera mellan olika mängder av liknande animationsklipp. Då systemet är baserat på fotsteg så tillåter detta karaktären att navigera även mellan regioner som inte är direkt ihopkopplade, till exempel genom att hoppa mellan gapen mellan de olika regionerna. Vi har implementerat systemet i C# med spelmotorn Unity och vi utvärderar systemet genom att betrakta karaktären när den utför diverse rörelser som till exempel att gå, springa och hoppa.

Contents

Chapter 1

Introduction

1.1 Background

Animations are the key to realism in interactive media such a games. An animation is a representation of how an object moves, rotates or changes in other ways over time [14]. With this information an animation can be played back, much like a movie, and a user observing it perceives the motion. Character locomotion concerns itself with how virtual characters, in particular bipedals like humans, move.

Modeling character locomotion

As humans we have a lot of experience in watching people move, therefore it is still hard to generate completely procedural and still believable animations from just a physical model [7]. What is commonly done instead is to use humans as a source for the animation by either letting an animator make believable animations manually or, as is becoming more common, use motion capture [13] to record human actors performing the actions one would want a virtual character to perform. This is done by letting the actors wear a special suit which allows the rotations and positions of their limbs to be precisely recorded. In practice it is usually impossible to record all possible ways a character might have to move. For this reason shorter animation clips are made which are then stitched together or blended as needed in various ways. For example if one wants a character to be able to walk in a world one may create a few different very short (on the order of one or a few seconds) walking animations such as walking forwards, walking in a curve to the left and walking in a curve to the right. Using just these animations one can produce a wide range of motion for the character by playing the animation clips one after another. One can also produce something in between by blending different animations. Blending means that one calculates a weighted average of some kind for two or more animations, in essence taking a little bit of one animation and a little bit of another animation to produce a new animation that looks like a mixture of both.

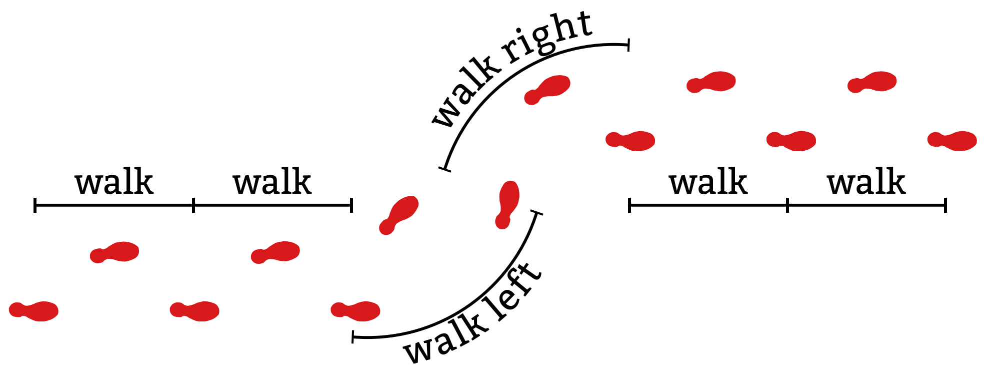


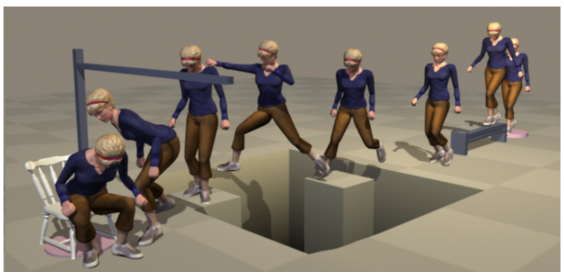
Figure 1.1: Several short walking animations played in sequence to produce more complex motion.

Just being able to play different animation clips one after the other doesn't get you very far however. Most virtual characters have some kind of goal, we don't want them to just wander around aimlessly, we want them to move somewhere. This requires a more thought out strategy and it is where this project comes in. We want to solve the problem of how we should move and animate the character as it moves to some goal position that we have determined.

The distinction between systems that have to respond quickly is crucial. Characters that are controlled directly by a user – using for example arrow keys on a keyboard to determine the character's movement direction – need to respond very quickly to user input to provide a good feel. In [9] it is observed that even seemingly small delays on the order of 150 ms can significantly affect the perceived ease of control and enjoyment of the game or simulation. On the other hand for characters that are not directly controlled by a user the delays are harder to notice and thus a small delay in order to produce more realistic motion is often a good trade-off. Non directly controlled characters also includes characters that are indirectly controlled by a user, for example by the user specifying where it wants the character to move, but does not control the motion in detail.

Another important distinction is between realistic looking motion, as this system generates, and actually physically realistic motion, as is required for e.g robots and the like. The motion generated by this system may not be physically realistic even if the input animation clips are. This is primarily because our system favors control when possible. For example if a manual sequence of footprints is used as input, our system will produce a reasonable looking path even if the footprints are placed such that the character cannot move physically correct between them. We see this as necessary as for interactive media such as games it is usually much more important to be able to follow some animation even if the input happens to be bad than preventing the system from producing a physically incorrect animation. In robotics on the other hand there needs to be an emphasis on ensuring that the robot can actually perform the motions as not doing so could lead to damaged equipment.

Applications and other approaches


(a) Image from [15].

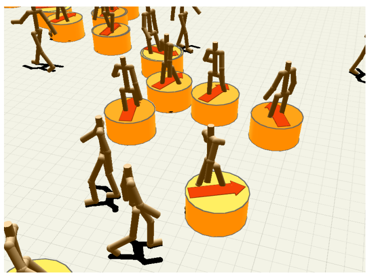

(b) Image from [19].

Figure 1.2

Character locomotion is widely used in interactive and non-interactive media such as video games, movies and simulations. Everywhere there is a need for an animated virtual character to navigate a world that may be changing dynamically or taking routes that cannot be predetermined there is also a need for a locomotion system. For games in particular there has always been a demand for ever more realistic animation, but even today when we have come far, the results are not too convincing. This is both due to performance constraints, many of the techniques that have been developed are simply not suitable for interactive use [10], [15], [6], as well as production cost constraints as many techniques require very large databases of animations to work, which is not feasible for many use cases. Other approaches are very performant but lack guarantees that the character will be able to move to a target correctly. This is in particular a problem for systems based on real-time controllers [19], [12]. One example of what can happen is that the controllers are slightly too slow to respond and that results in the character for example walking off the edge of a platform or missing the opening of a door and instead hitting the wall. These systems are much better suited to characters that are directly controlled by a user.

This paper aims to make a contribution by presenting an animation driven locomotion system suitable for interactive use.

1.2 Overview

The main approach of our system is based around footprints. We first build a graph of valid footprints, i.e positions in the virtual world where it is valid for a character to place its foot. The footprints in the graph are connected with edges if there is an animation that takes a character from one footprint to another. When a character needs to move somewhere we search for a path in this graph and the resulting sequence of animations and footprints is processed in various ways to make it more realistic and aesthetically pleasing. To improve the realism of the animation we smoothly interpolate between different walking and running animations based on an estimated measure of curvature of the path as well as boundary constraints on the speed of the character. Finally an inverse kinematics stage is used to prevent the feet from sliding around on the ground.

Chapter 2

Related work

Various strategies for character locomotion have been explored. In "Motion Graphs" [10] (and expanded upon in [15], [6] and many other papers) a method is described for synthesizing high quality character locomotion from a set of animations by representing each motion as an edge between two nodes representing the start and end poses. Poses that are very similar can have a transition edge added between them and then the motion synthesized by searching for paths in this motion graph. While these methods generate very high quality results and are very flexible, they are too slow to use for real time and resource constrained scenarios (such as in video games). See [4] for an overview of the relevant papers.

In "Near-optimal Character Animation with Continuous Control" [19] a different approach is taken where value functions are learned so that at each point in time, the character only needs to perform a few fast evaluations of a value function to decide which animations to blend to. This yields very performant and responsive controllers that may be suitable for direct user control. However while some obstacle avoidance was demonstrated, it is difficult to scale up to more complex environments. Furthermore it does not lend itself well to interacting with a fixed environment (for example opening a door) which may require the character to stop at a very precisely defined position and orientation. "Real-time Planning for Parameterized Human Motion" [12] takes a similar approach where reinforcement learning is used, but unfortunately has the same limitations.

In "Planning Biped Locomotion Using Motion Capture Data and Probabilistic Roadmaps" [2] instead of trying to blend animation clips such that the character moves along the desired path, footholds (for the right foot) are instead sampled in the world. A graph is constructed between the footholds such that an edge means that there is at least one animation clip which has the first foothold at the start of the clip and the second foothold at the end of the clip. Using normal path planning techniques a path can be generated that goes from the start position to the target position with information about how the feet should be placed. This sequence of footholds in then post processed to smooth it out and the animation corresponding to the edges between the clips are transformed so that they match the post processed footholds. This produces high quality locomotion which can take into account precise boundary conditions (such as stopping at a precise location). It is not real time, but not that far off (calculation times are on the order of 1 second). The method does not lend itself well to scenarios where responsiveness is a high priority as planning takes place several seconds before the character reaches a particular point.

Chapter 3

Implementation

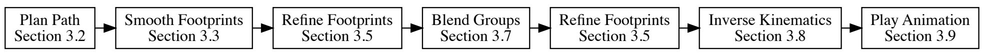
3.1 Outline

The system is partially based on the work by [2]. We extend their approach to produce more realistic animation by allowing animations to be arbitrarily blended instead of just stitching together sequences of animation clips that start and stop at a few fixed poses, as well as improving the performance significantly by taking advantage of how different motion deviations happen at different frequencies.

Allowing animations to be blended arbitrarily has a large positive impact on the perceived realism of the resulting animation in many situations. For example it allows the character to turn with any angular speed in the range that the input animation clips encompass instead of at only a few fixed angular speeds.

The system is well described as a pipeline with several stages (see image below). We will describe the stages in more detail in the later sections, but first we will give a high level overview of how it works.

First we plan a path in a navigation map much like in [2] which produces a sequence of footprints for the right foot together with the animations that moves the character from one footprint to the next. Then we smooth the footprints, primarily to make it shorter. The smoothed footprints no longer correspond that well to the animation clips however, so the next stage is to refine the footprints so that they locally match their corresponding animation clips better. We then approximate blending parameters for the animations that can be blended. These blending parameters can for example be the turning speed. We estimate the parameters using various measures, for example the curvature of the curve that traces the center of gravity of the character. We then replace each animation clip that can be blended in a group by the corresponding blend group (for example a clip in which the character walks in a circle to the left is replaced by an animation group that blends between all the different walking animations). After this the footprints again no longer match the animation clips that well, so we run a second iterations of the refinement stage. Finally we apply inverse kinematics (IK) [18] to minimize foot sliding.


Plan Path Section 3.2
Smooth Footprints Section 3.3
Refine Footprints Section 3.5
Blend Groups Section 3.7
Refine Footprints Section 3.5
Inverse Kinematics Section 3.8
Play Animation Section 3.9

The different stages will be described in detail below.

3.2 Navigation graph

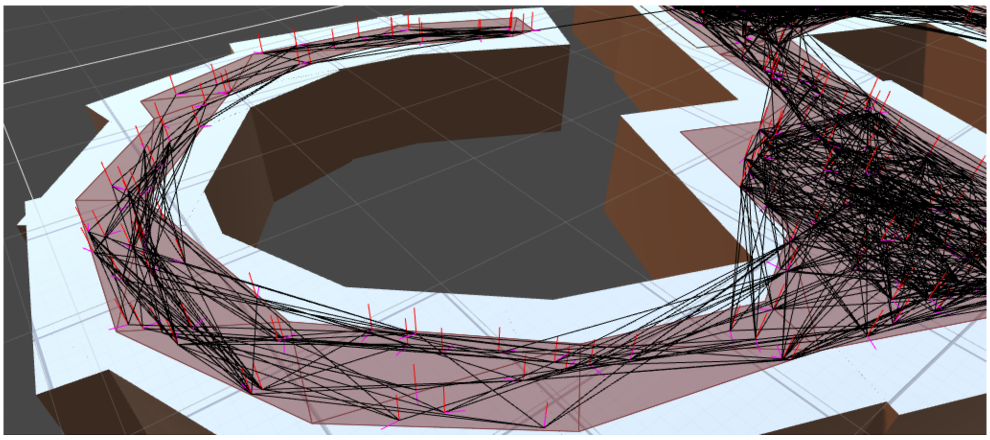


Figure 3.1: Example of a navigation graph. The black connections represent animation clips moving between the footprints in red. The traversable surface which the navigation graph is based on is visible in dark red.

If we have a character that needs to move in some way from one part of the world to another we need to plan a path for it. We construct a navigation graph by sampling points/rotation pairs randomly all over the traversable surface of the world and then connect nearby points by an edge if there is an animation clip in the database which matches the edge reasonably well when the points are treated as the position and rotation of the right foot at the start and end of the animation. This stage is very similar to [2] so for further details we will refer to that paper.

After a path has been planned in the navigation graph we continue by smoothing out the kinks in it in the next section.

3.3 Smoothing

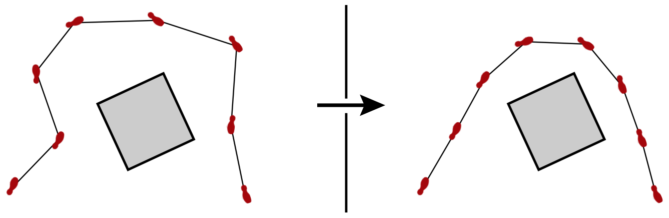


Figure 3.2: Smoothing a sequence of footprints. Note that the footprints are prevented from entering the obstacle.

The sequence of footprints produced by the path planning in the navigation graph tends not to be very smooth. We smooth the footprints using a simple smoothing kernel that we apply a small number of times (in our tests we used 4 iterations). If we label the footprints in the sequence as $\{p_i \in \mathbb{R}^3\}$ and the rotations of the footprints as $\{q_i \in \mathbb{S}^3\}$ (represented by quaternions [5]. See appendix A.2) we can smooth them using a binomial smoothing kernel as

$$\boldsymbol{p}_i' = \frac{1}{16} \cdot \big(\boldsymbol{p}_{i-2}, \boldsymbol{p}_{i-1}, \boldsymbol{p}_i, \boldsymbol{p}_{i+1}, \boldsymbol{p}_{i+2}\big) \cdot (1, 4, 6, 4, 1)^T$$

$$q_i' = q_i \cdot \exp\left(\frac{1}{16} \cdot (\boldsymbol{\omega}_{i-2}, \boldsymbol{\omega}_{i-1}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_{i+1}, \boldsymbol{\omega}_{i+2}) \cdot (-1, -5, 0, 5, 1)^T\right)$$

Where $\boldsymbol{\omega}_i = \log\big((q_i)^{-1} q_{i+1}\big)$ or in other words the angular difference between the rotation of footprint i and $i + 1$. Both smoothing kernels have the effect of making the positions and rotations of the footprints more similar to their adjacent footprints. After each smoothing iteration we snap the position of each footprint to the closest point on the surface of the world that a footprint can be placed on to make sure the path is still valid. This can be done in various ways, which way is not particularly important for our use case. We represent the traversable surface of the world using a set of polygons (see figure 3.1) and simply find the closest point on any of those polygons.

3.4 Parametric blending

Here we take a detour to discuss how animations are blended which is an often overlooked detail but which produces large artifacts if not handled properly. We will need this information for the next stage in the pipeline.

To provide smooth transitions we need some method of combining a set of animations with their corresponding blend weights. We define the set of all animations as $\{a_i\}$ and their corresponding blend weights as $\{w_i\}$ such that $\sum w_i = 1$. If we for example have two animations a_0 and a_1 which were walk forward and walk to the right in a circle we could make our character walk in a wider circle by blending the two animations with for example $w_0 = 0.5$ and $w_1 = 0.5$. Each animation consists of, for each frame, the local offset $\Delta\boldsymbol{r} \in \mathbb{R}^3$ and an angular offset $\Delta q \in \mathbb{S}^3$ that is used to move the character as well as the rotations of all the bones in the character. Thus if the character is positioned at position $\boldsymbol{r}$ with rotation q and we are playing an animation a at time t then we can calculate where the character will be at the next frame as

$$\begin{aligned} \boldsymbol{r}' &= \boldsymbol{r} + q \cdot \Delta\boldsymbol{r}(a,t) \cdot q^{-1} \\ q' &= q \cdot \Delta q(a,t) \end{aligned}$$

Or in other words that the character moves in the direction of $\Delta\boldsymbol{r}$ for that frame, but note that $\Delta\boldsymbol{r}$ is relative to the character so it needs to be converted from local space to global space by rotating the vector with the character's current rotation. The rotation is done using a quaternion-vector multiplication, for a vector $\boldsymbol{v}$ and rotation q the operation $q \cdot \boldsymbol{v} \cdot q^{-1}$ corresponds to $\boldsymbol{v}$ being rotated using the rotation. The character is also simply rotated by Δq.

One may think that to blend multiple animations we could take a weighted average of $\Delta\boldsymbol{p}$ according to the blend weights

$$\boldsymbol{r}' = \boldsymbol{r} + q \cdot \left(\sum_i w_i \Delta\boldsymbol{r}(a_i, t_i) \right) \cdot q^{-1}$$

This does not produce satisfactory results however as this does not account for terms that come from how the blending weights change over time.

The character's position is determined by its center of gravity, however it does not always make sense that the center of gravity should stay fixed when blending between animations. If we take the example in figure 3.3 in which we blend quickly from running in a circle to the left to running in a circle to the right. The character tilts significantly inwards into the circle and thus the center of gravity is also offset inwards. When blending to the other animation we would expect the feet of the character to stay approximately where they are, but unless we add extra terms for that it is the center of gravity that will stay fixed and the feet will slide a great distance across the ground which will not look particularly realistic. We define the *blend point* as the point that stays fixed when blending between a set of animations.

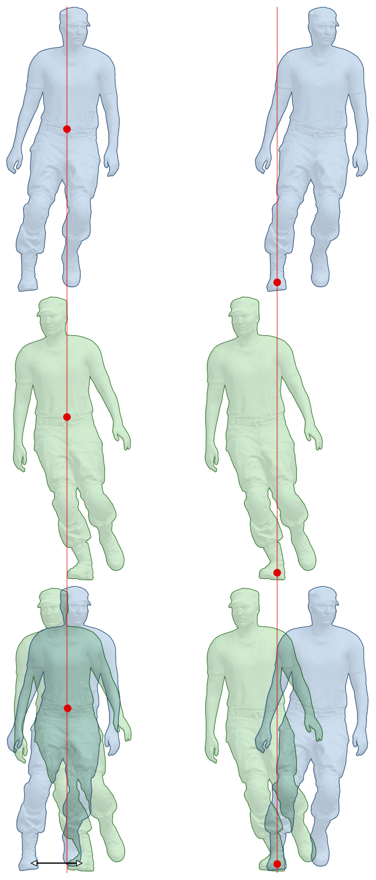


Figure 3.3: Blending from a walk to the left (blue) to a walk to the right (green) with different blend points (red).

We observe that when a single foot is on the ground, it is the position of that foot that we want to keep fixed. To extend this to arbitrary situations we introduce the *most stable point* p_s between the two feet. We draw a virtual line from the left foot to the right foot of the character and study how points on that line move as the time passes. We define the most stable point as the point on that line which moves the least over time. If only one foot is stationary, then this will be the position of that foot. Assume the positions and velocities of the feet are $\boldsymbol{p}_L, \boldsymbol{v}_L$ and $\boldsymbol{p}_R, \boldsymbol{v}_R$ respectively for the left and right feet, we then want to find the point

$$\boldsymbol{p}_s = \boldsymbol{p}_R\alpha + \boldsymbol{p}_L(1-\alpha)$$

that lies on the line from $\boldsymbol{p}_L$ to $\boldsymbol{p}_R$ which minimizes the velocity at that point.

$$\operatorname*{argmin}_{\alpha} |\boldsymbol{v}_R\alpha + \boldsymbol{v}_L(1-\alpha)|$$

Here α is an interpolation parameter which defines p_s.
This is equivalent to

$$\underset{\alpha}{\operatorname{argmin}} |\boldsymbol{v}_R\alpha + \boldsymbol{v}_L(1-\alpha)|^2$$

which we can expand and simplify to

$$\underset{\alpha}{\operatorname{argmin}}\big(|\boldsymbol{v}_R - \boldsymbol{v}_L|^2\alpha^2 + 2\big(\boldsymbol{v}_R\boldsymbol{v}_L - |\boldsymbol{v}_L|^2\big)\alpha + |\boldsymbol{v}_L|^2\big)$$

We differentiate with respect to α and set the derivative to 0 to find the minimum

$$\frac{d}{d\alpha}\Big[|\boldsymbol{v}_R - \boldsymbol{v}_L|^2\alpha^2 + 2\big(\boldsymbol{v}_R\boldsymbol{v}_L - |\boldsymbol{v}_L|^2\big)\alpha + |\boldsymbol{v}_L|^2\Big] = 0$$

which leads to

$$\alpha = \frac{|\boldsymbol{v}_L|^2 - \boldsymbol{v}_R\boldsymbol{v}_L}{|\boldsymbol{v}_R - \boldsymbol{v}_L|^2}$$

To make sure this is actually a point between the left and right foot we clamp α to be between 0 and 1.

$$\alpha = \min\left(1, \max\left(0, \frac{|\boldsymbol{v}_L|^2 - \boldsymbol{v}_R\boldsymbol{v}_L}{|\boldsymbol{v}_R - \boldsymbol{v}_L|^2}\right)\right)$$

This means that if we in an instant change the interpolation parameter from α to α' we would have to make the following adjustment to the character's position to keep the blend point $\boldsymbol{p}_s$ fixed

$$\boldsymbol{r}' = \boldsymbol{r} + (\boldsymbol{p}_s - \boldsymbol{p}_s') = \boldsymbol{p} + (\boldsymbol{p}_L\alpha + \boldsymbol{p}_R(1-\alpha)) - (\boldsymbol{p}_L\alpha' + \boldsymbol{p}_R(1-\alpha')) \Leftrightarrow$$

$$\boldsymbol{r}' = \boldsymbol{r} + (\boldsymbol{p}_R - \boldsymbol{p}_L)(\alpha' - \alpha)$$

To calculate the final movement in a way that avoids cyclical dependencies we divide the movement into two phases. As usual x for some variable x refers to the value for the previous frame while x' refers to the new value calculated during this frame.

In the first phase we calculate the movement without any adjustment based on the blend point and move the character using that information. We sample the new positions of the feet $(\boldsymbol{p}_L', \boldsymbol{p}_R')$ and calculate the velocities of them.

Note that for the rotational part we linearly interpolate quaternions which does not in general make sense mathematically. However if the quaternions are very close to each other – which in our case they are since the character will not rotate that much in a single frame and thus all Δq will be close to the identity quaternion – then linearly interpolating between quaternions is approximately correct so we can actually do that.

$$\begin{aligned}
\boldsymbol{r}' &\leftarrow \boldsymbol{r} + q \cdot \Delta\boldsymbol{r}(a,t) \cdot q^{-1} \\
q' &\leftarrow q \cdot \left(\sum_i w_i \Delta q(a_i, t_i)\right) \\
\boldsymbol{v}'_L &\leftarrow \frac{\boldsymbol{p}'_L - \boldsymbol{p}_L}{\Delta t} \\
\boldsymbol{v}'_R &\leftarrow \frac{\boldsymbol{p}'_R - \boldsymbol{p}_R}{\Delta t}
\end{aligned}$$

In the second phase we calculate the new α and finally reposition the character based on the new blending point.

$$\begin{aligned}
\alpha' &\leftarrow \min\left(1, \max\left(0, \frac{\boldsymbol{v}'^2_L - \boldsymbol{v}'_R\boldsymbol{v}'_L}{|\boldsymbol{v}'_R - \boldsymbol{v}'_L|^2}\right)\right) \\
\boldsymbol{r}' &\leftarrow \boldsymbol{r}' + (\boldsymbol{p}'_R - \boldsymbol{p}'_L)(\alpha' - \alpha)
\end{aligned}$$

For any set of animations with associated weights, the motion of the character can now be calculated. In the next section we will use this to improve the quality of the path.

3.5 Refinement

At this stage we have a smoothed sequence of footprints with accompanying animation clips and we are in this stage interested in making the desired footprints more locally similar to the original animation clips. As a rule animation clips to not line up perfectly at the time of transition unless a large amount of care is taken by the animator. Making sure the animation clips line up perfectly also limits the animation clips that can be used without intermediate transition animations. For example when a character is running in a circle it is natural for the character to lean slightly inwards into the circle, however that causes problems if the animation should line up perfectly with the running forwards animation in which the character does not lean in any direction. To solve this we transition between the animations over a short amount of time (we used 0.3 seconds) by interpolating the blend weights.

Our method is similar to the one used by [2] but we have generalized it to handle blended animations as well as tweaked how the rotation was handled to, in our view, be more realistic. The method in [2] could end up not correcting for unnatural rotations in the original footprints which could lead to a final animation of low quality.

We let the animation play out without any adjustments whatsoever (as described in section 3.4) and track the positions of the footprints that the character leaves. These will be different from the footprints that we want the character to use, often significantly so. We denote the footprints that the character left when playing the original animation clips as $\boldsymbol{p}_i^O$ and q_i^O for the positions and rotations respectively. We similarly denote the desired footprints as $\boldsymbol{p}_i^D$ and q_i^D. For each footprint we look at its two adjacent footprints (this includes all footprints, so for a footprint by the right foot the adjacent footprints would typically have been made by the left foot) and create an affine transformation that maps from the positions of the original footprints to the desired footprints such that $\boldsymbol{p}_{i-1}^O$ maps to $\boldsymbol{p}_{i-1}^D$ and $\boldsymbol{p}_{i+1}^O$ maps to p_{i+1}^D. We can view the transformation as first applying an offset $\boldsymbol{p}_{i-1}^D - \boldsymbol{p}_{i-1}^O$ which moves the previous original footprint to the position of the previous desired footprint. Then we apply a rotation so that the direction from the previous footprint to the original next footprint is parallel with the direction from the previous footprint to the next desired footprint. Finally we apply a uniform scale so that the next original footprint ends up at the same position as the next desired footprint. See figure [todo].

We then transform the current footprint, $\boldsymbol{p}_i^O$ and q_i^O, with the transformation which produces a new desired position and rotation for the footprint that makes it very similar to the original animation clip. We apply this transformation in several iterations to all footprints (except for the first and last footprints) and make sure to always snap the positions of the footprints to the closest points on the traversable surface of the world. This process converges relatively slowly, but we have found that on the order of 10 iterations is sufficient for our use case.

The original sequence of footprints have now been both smoothed, which took away some detail, and then refined to add back the detail that we really want, i.e that the footprints should be similar to the animations that the character uses at that time. However we only have the footprints, not a body trajectory that determines where the character should be positioned and rotated at each point in time, so in the next section we will describe how to calculate that.

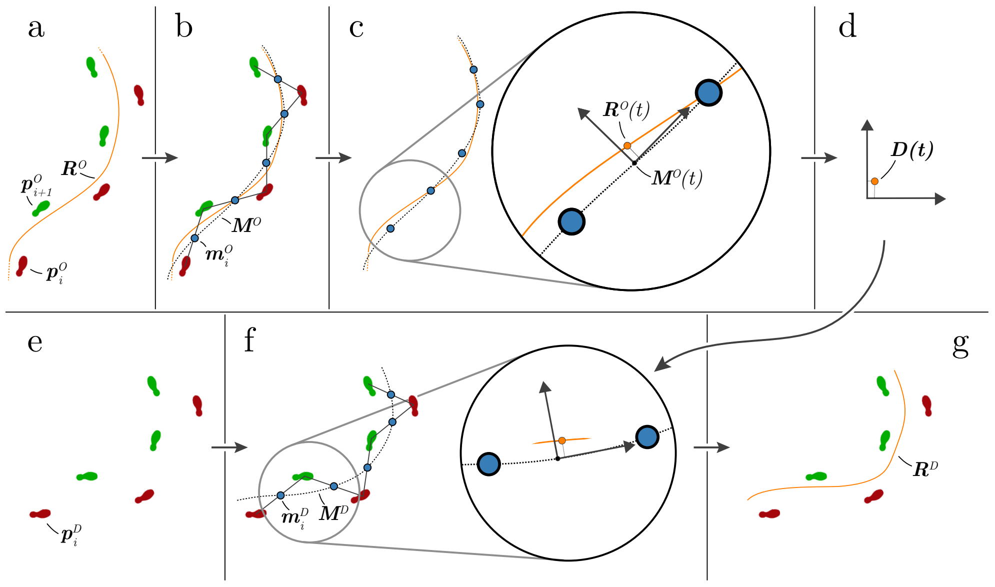


Figure 3.4

3.6 Body trajectory

We now have a sequence of footprints as well as blending parameters for the whole animation. For the final IK stage we will adjust the path of the character to minimize foot sliding, however for it to be effective we need to have a good initial estimate of how the character is going to move. We extend the method introduced by [2] to handle arbitrarily blended animation sequences.

To reiterate, for the final result we need we need to know, for each frame, where the character (i.e its center of gravity) will be as well as its rotation. As it stands right now we have the desired footprints that we want the character to leave (fig 3.4.e) but not the exact body trajectory. However we do also have the blending weights for the animation clips for each frame, so we can let the animation play out without any adjustments whatsoever and record the center of gravity of the character as well as the footprints the character leaves (fig 3.4.a). It turns out that we can transfer the body trajectory from a trajectory following the original footprints to a trajectory following the desired footprints (fig 3.4.g) using the approach detailed below.

Let us denote the original body trajectory of the character $\boldsymbol{R}^O(t)$ and the corresponding rotation of the character $Q^O(t)$ (fig 3.4.a). Further let t_i^O be the times when the character left each footprint when letting the animation play out. In [2] it is observed that a reasonably good estimate for the center of gravity is right between two adjacent footprints of different feet. Since we know both the original footprints and the desired footprints we can calculate the midpoints for both cases. Let us denote these midpoints as $\boldsymbol{m}_i^O$ and $\boldsymbol{m}_i^D$ for the original and desired midpoints respectively (fig 3.4.b,e) such that $0 \leq i \leq n$. For each midpoint we also store the time ($m_{t,i}^O$ and $m_{t,i}^D$) in the animation that it corresponds to as well as a rotation ($m_{q,i}^O$ and $m_{q,i}^D$) for it. We define these values as the mean of the values for the two footprints they were created from. For ease of notation we will use j as a placeholder for both O and D.

$$\begin{aligned}
\boldsymbol{m}_i^j &= \frac{\boldsymbol{p}_i^j + \boldsymbol{p}_{i+1}^j}{2} \\
m_{t,i}^j &= \frac{t_i^j + t_{i+1}^j}{2} \\
m_{q,i}^j &= \operatorname{slerp}\left(q_i^j, q_{i+1}^j, \frac{1}{2}\right)
\end{aligned}$$

Here $\operatorname{slerp}(q_1, q_2, x)$ is the **s**pherical **l**inear int**erp**olation defined in the usual way [16].

To extend the sequences $\boldsymbol{m}^j$ to cover all frames in the animation we fit a sequence of centripetal Catmull-Rom splines (see appendix A.1) through the points (fig 3.4.b,e). We denote the Catmull-Rom splines as $\boldsymbol{M}^O$ and $\boldsymbol{M}^D$.

$$\boldsymbol{M}^j(t) = \boldsymbol{p_{c,r}}\left(\boldsymbol{m}_{k-1}^j, \boldsymbol{m}_k^j, \boldsymbol{m}_{k+1}^j, \boldsymbol{m}_{k+2}^j, \frac{t - m_{t,k}^j}{m_{t,k+1}^j - m_{t,k}^j}\right)$$

where

$$k = \begin{cases} 0 & \text{if } t < m_{t0}^j, \\ n & \text{if } t > m_{tn}^j, \\ \max\{i : t \geq m_{ti}^j\} & \text{otherwise.} \end{cases}$$

To ensure the spline is defined at the first as well as last points we extend $\boldsymbol{m}$ with $\boldsymbol{m}_{-1}^j = \boldsymbol{m}_0^j + \left(\boldsymbol{m}_0^j - \boldsymbol{m}_1^j\right)$ and $\boldsymbol{m}_{n+1}^j = \boldsymbol{m}_n^j + \left(\boldsymbol{m}_n^j - \boldsymbol{m}_{n-1}^j\right)$.

For the rotation we use a simpler interpolation

$$M_q^j(t) = \operatorname{slerp}\left(m_{q,k}^j, m_{q,k+1}^j, \frac{t - m_{t,k}^j}{m_{t,k+1}^j - m_{t,k}^j}\right).$$

Let us now define the *displacement* of the body trajectory from the Catmull-Rom spline (fig 3.4.c,d)

$$\begin{aligned}\boldsymbol{D}_p(t) &= \left(M_q^O(t)\right)^{-1} \cdot \left(\boldsymbol{R}^O(t) - \boldsymbol{M}^O(t)\right) \cdot M_q^O(t)\\ D_q(t) &= \left(M_q^O(t)\right)^{-1} \cdot Q^O(t)\end{aligned}$$

It can be seen as defining a new coordinate system at position $\boldsymbol{M}^O(t)$ with rotation $M_q^O(t)$ and extracting the coordinates of the point $\boldsymbol{R}^O(t)$ in that coordinate system.

Finally we determine the desired body trajectory by treating $\boldsymbol{D}_p(t)$ as the local coordinates in a coordinate system at $\boldsymbol{M}^D(t)$ with rotation $M_q^D(t)$ (fig 3.4.f,g)

$$\begin{aligned}\boldsymbol{R}^D(t) &= M_q^D(t)\boldsymbol{D}_p(t)\left(M_q^D\right)^{-1} + \boldsymbol{M}^D(t)\\ Q^D(t) &= M_q^D(t)D_q(t).\end{aligned}$$

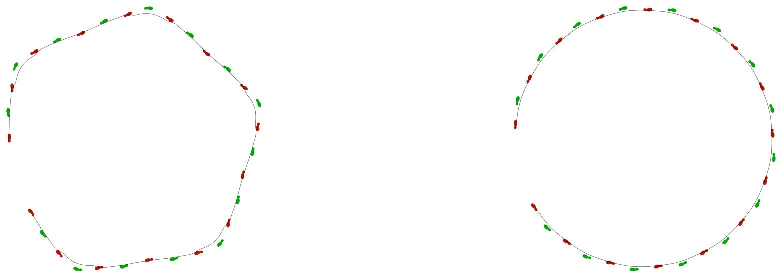
This section has described how we can calculate a reasonable body trajectory for the character. We can improve this trajectory however and this is described in the next section.

3.7 Blend groups

So far we have had a sequence of animation clips that we have transitioned between over short durations of time. This may however not lead to the best animation quality as the animation clips in the database may not cover all behavior that we may want. We will describe two ways to calculate better blend parameters for the animations, one for the turnings speed of the character and one for the speed of the character.

3.7.1 Turning

A common case is that the character is moving in an arc. In the animation clip database one would typically have a moving forwards animation as well as a few animations of the character moving forwards and turning by varying amounts (so that if the clips would loop, the character would walk in a circle). However since we only have animation clips for a few discrete turning radii, any movement approximating an arc would have to be composed out of animation clips which do not precisely match that arc. For example a wider arc could be achieved by playing the clips walk-forward, walk-right, walk-forward, walk-right etc. The results of this are not particularly pleasing however as in real life no person would do that, instead one would walk in a smooth arc with the correct turning radius. This is illustrated in figure 3.5.



(a) Repeating sequence of walk-forward, walk-forward, walk-right.

(b) Blend of $\frac{2}{3}$ walk-forward and $\frac{1}{3}$ walk-right.

Figure 3.5

Other cases where this can be a problem is jumping over a crevice, where the animation clip database may have different animations for different lengths of the jump. If the animation database included strafing, the strafing direction would be a good candidate as well.

To solve this we group animations into *blending groups*. A blending group consists of a set of animations that can freely be interpolated between using one or more parameters that can be calculated from an existing path. In our tests we have for example a blending group consisting of 5 animations: walk-left, walk-left-wide, walk-forward, walk-right-wide and walk-right. They are parametrized by curvature $\tau = \frac{1}{r}$ where r is the turning radius. In our sequence of animation clips we replace all animation clips that are included in the blending group by the blending group itself. What remains now is to supply the blending group with a blending parameter (e.g τ) for the path.

We do this by approximating τ at several points along the path and then fitting a b-spline to the data. For each frame in the animation we pick one point on the body trajectory a fraction of a second in the past and one point a fraction of a second into the future. Then we calculate the radius R_c of the circle that passes through those 3 points which is well known how to do.

$$\tau(t) = \frac{1}{r} = \frac{1}{R_c(\boldsymbol{R}(t - \Delta t), \boldsymbol{R}(t), \boldsymbol{R}(t + \Delta t))}$$

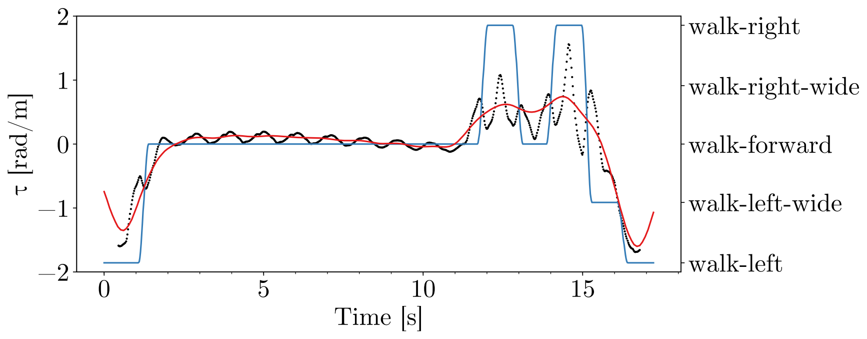


Figure 3.6: Approximation of τ for use as a blending parameter in a simple test case. The original blend parameter (when transitioning between individual clips) is in blue. The approximated values of τ are in black and the smoothed b-spline for τ is in red.

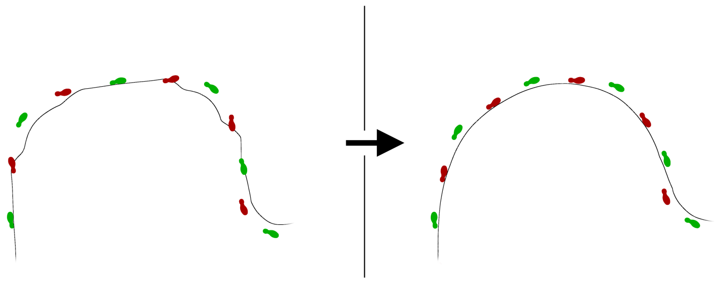


Figure 3.7: Body trajectory before and after using blend weights determined by τ. This corresponds approximately to the time range [10, 16] in figure 3.6 with the left side corresponding to the blue blend weights in that figure and the right side corresponding to the red blend weights.

We fit a b-spline to $\tau(t)$ and use that as the blending parameter. The b-spline is used as a low-pass filter to smooth out any high frequencies which could cause the character to unnaturally quickly transition from one animation clip to another. That it is a b-spline is not particularly important, any other interpolation method which acts as a low-pass filter (e.g a moving average filter) would work as well. The amount of smoothing in the filter is set by a user defined constant.

In figure 3.6 this is done for a simple test case and in figure 3.7 the differences in body trajectory can be seen. We can see that the body trajectory when blending groups are used is significantly smoother. One can note that the character sways slightly to the left and to the right when walking which is visible as small oscillations in the black curve in figure 3.6, these oscillations are smoothed out by the b-spline.

To get the final blend weights, let τ_i be the curvature for animation clip i in the group. Let a denote the animation clip with the greatest τ_i that is still smaller than the current value of $\tau(t)$. Similarly let b denote the animation clip with the smallest τ_i that is still greater than $\tau(t)$. We then let the animation weights for the animations in the group be determined by linear interpolation

$$\begin{aligned}
w_a &= w_{\text{group}} \cdot \frac{\tau(t) - \tau_a}{\tau_b - \tau_a}, \\
w_b &= w_{\text{group}} \cdot \left(1 - \frac{\tau(t) - \tau_a}{\tau_b - \tau_a}\right), \\
w_i &= 0 : i \neq a \wedge i \neq b.
\end{aligned}$$

If it is not possible to find either a or b because $\tau(t)$ is outside the range allowed by the animations. We let the animation with the closest τ_i get the full weight and the rest of the animations get a weight of 0. w_{group} is the weight for the whole group. The group can now be blended together with the rest of the animations as if it was a normal animation and not a set of them.

3.7.2 Speed

Another important case is the speed of the character or more specifically if it is walking or running. We will differentiate here between animations that are purely walk or run animations, and those that are not, for example a jump animation, open-door animation or any other kind. The key differentiator is that we will, like in the previous section, allow

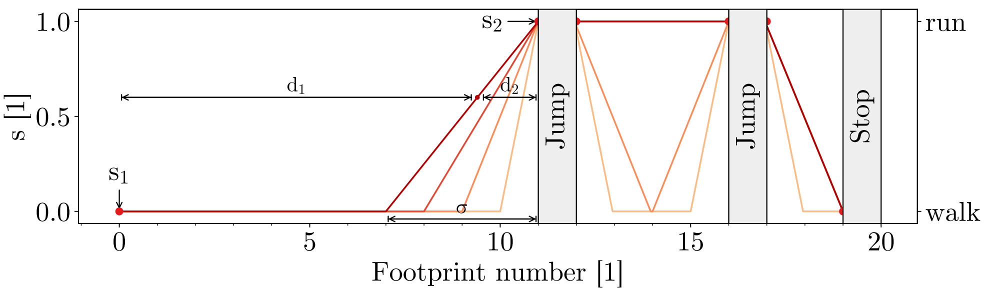


Figure 3.8: Values of s in the Jumps scenario (see section 4) for $\sigma = 1, 2, 3, 4$.

walk and run animations to be freely blended between each other, but we will also allow walk animation to blend freely with run animations. On the other hand it does not make much sense to blend say a jump animation with a run animation as the poses in them may be quite distinct. For ease of writing, we will let *special animations* denote the animations that are not purely walking or running. The special animations may start and end with the character either walking or running and as is often the case they will either be preceded or followed by a walking or running animation (which we in the previous section combined to a single blend group). To make the animation as smooth and natural as possible it is desirable that the character has the correct speed (i.e the correct blend between walking and running) when starting to transition to the special animations (or starts with the correct speed when transitioning away from the special animations).

We will let the blend between walking and running to be parametrized by a blending parameter s. If it is 0 the character is walking and if it is 1 the character is running. In some situations it may make sense to have more than two speeds, for example walking, jogging and sprinting. In this case jogging would end up somewhere between 0 and 1.

We can divide the whole animation up into sections at the transition points between special animations and walking/running animations. At the transition points we know the value that needs to have s based on if the animation that is transitioned from or to is in a walking or running state at that time (this information can be manually annotated or automatically inferred based on the velocity of the character in the animation). We also specify the desired values of s at the start and end of the animation. The problem now consists of filling in the values of s

in between these fixed points. See figure 3.8 for an illustration.

We make s vary linearly such that it reaches the desired speeds at the right times but slows down to a walk if it can. If we look at a single section between two known values of s. Let the distance in *normalized time* (todo: define) to the earlier known value be $d1$ and the distance to the later known value be $d2$. Further let the known values themselves be $s1$ and $s2$. We then define the in-between values as

$$s(d_1, d_2, s_1, s_2) = \begin{cases} \frac{s_1 d_2 + s_2 d_1}{d_1 + d_2} & \text{if } d_1 + d_2 < 2\sigma(s1 + s2), \\ \max\bigl(0, s_1 \cdot \frac{\sigma - d_1}{\sigma}, s_2 \cdot \frac{\sigma - d_2}{\sigma}\bigr) & \text{otherwise} \end{cases}$$

where σ is a user defined parameter which controls how long it takes for the character to speed up or slow down. We used $\sigma = 2$ in our tests. When there is not enough time to slow down completely between the two known values (the first case) then we linearly interpolate between s_1 and s_2 over all available time. This is visible between the two jumps and after the last jump in figure 3.8. We do this because from empiric testing it feels unnatural if the character seems to slow down to a walk, only to start running again before it has finished slowing down.

A more complicated interpolation scheme was also tested that ensured that the derivative of s was continuous. However after evaluating it, it turned out that if the derivative of s is continuos or not is not easily perceived and the linear interpolation looked at least as good or possibly better.

When we have s we use that to blend between the walking and running animation groups very similarly to what we did earlier for the turning speed. Let w_{walk} and w_{run} denote the weights for the walking and running animation groups. We let their weights be determined by

$$w_{\text{run}} = w_{\text{group}} \cdot s, \; w_{\text{walk}} = w_{\text{group}} \cdot (1 - s).$$

As previously w_{group} is the weight for the whole group. In this case itself consisting of two sub-groups.

After we have calculated these new blending parameters, we run the refinement (see section 3.5) stage again to make the footprints more similar to the new animation. We will continue in the next section by describing how to ensure the feet do not slide around on the ground as all our modifications to how the character moves may have caused it to deviate significantly from its natural motion (i.e without any adjustments whatsoever, just playing the animation).

3.8 Inverse Kinematics

At this stage we have a reasonable path for the character to move along, however if we would make the character follow this path while playing the original animations then the character's feet would slide significantly. To solve this we post process the path using inverse kinematics (IK) [18] to position the character's feet where they should be. The IK solver may offset bone rotations as well as the position and orientation of the character. Once we have these offsets we use the method presented in [11] to fit a b-spline to them to make sure that the result is smooth.

In [2] a conjugate gradient descent method was used to optimize both the bone rotations as well as the position and rotation of the character at the same time. The major drawback of this approach is however that it is slow. In fact the time it took to run the conjugate gradient descent was usually greater than the combined time it took to run all other stages of the algorithm, something which we have been able to replicate.

To improve the performance we argue that low frequency changes should primarily happen by changing the character's position and rotation while high frequency changes should happen by modifying the characters bone rotations. Here low frequencies are frequencies lower or around the same as the frequency of the character's footsteps. A bipedal character always strives for balance, so if the bones were for a longer duration biased in a certain direction (say for example that the legs were biased to on average point more to the right of the character) then a real character would in most cases try to move them back to a more neutral position, possibly moving or rotating the entire body while doing so. The result would be that any low frequency changes would be transferred to the body trajectory instead. This is of course only a qualitative argument and it doesn't hold in all cases. It does however empirically work well as a first approximation and allows us to significantly speed up the algorithm so the trade-off seems reasonable. Furthermore by allowing animations to be arbitrarily blended (see section 3.7) the adjustments made by the IK stage (either using the algorithm that will be described here or the one used in [2]) have empirically been made much smaller as the animation itself is much smoother. Therefore the difference between the results of the technique in [2] and the one described here is not large while the differences in performance are between one and two orders of magnitude depending on the test case.

The base approach is the same as in [2]. We use hierarchical b-splines as introduced by [11] to build up the offset from the original values over several iterations. The reason for using hierarchical b-splines instead of a single b-spline is to both make it possible for the character to 'anticipate' having to e.g place a foot at a particular position but still allow make it possible to represent enough detail to satisfy the IK goals well. The main difference is that we separate the adjustment of the characters position and rotation with the adjustment of the bone rotations. First we will adjust the position and rotation of the character once, then we will proceed to adjust the bone rotations over a few iterations. This allows us to use fast methods in both cases instead of having to fall back on more general but slower methods such as conjugate gradient descent.

3.8.1 Body offset

For each frame in the animation either one foot or both feet will be touching the ground. If at least one of them are we want to adjust the character's position and rotation so that the foot is more accurately placed exactly where we want it to be. We do this by modeling the character with a simple differential equation.

Let $\boldsymbol{p}_{\text{ik}}$ be the ik target position (i.e where we want to place the foot, for details see section 3.8.3). Let $\boldsymbol{p} = \boldsymbol{R}^D(t)$ be the position of the character and $q = Q^D(t)$ be the rotation. Further let $\boldsymbol{p}_{\text{loc}}$ be the local position of the foot relative to the character. This means that the global position of the foot is

$$\boldsymbol{p}_{\text{glob}} = \boldsymbol{p} + q \cdot \boldsymbol{p}_{\text{loc}} \cdot q^{-1}.$$

Ideally then we would like that

$$\boldsymbol{p}_{\text{glob}} = \boldsymbol{p}_{\text{ik}}$$

or in other words that the foot's global position lines up perfectly with the ik target position.

Using this we define an error

$$\boldsymbol{F} = \boldsymbol{p}_{\text{ik}} - \boldsymbol{p}_{\text{glob}}.$$

and

$$\boldsymbol{T} = \boldsymbol{F} \times \left(q \cdot \boldsymbol{p}_{\text{loc}} \cdot q^{-1}\right)$$

Then we use an iterative process to bring the character closer to the desired position. In each step we modify the position and rotation of the character as

$$\boldsymbol{p} \leftarrow \boldsymbol{p} + \alpha \boldsymbol{F}$$
$$q \leftarrow \mathbf{exp}(\beta \boldsymbol{T}) \cdot q$$

Where α, β are user defined constants. The fraction $\frac{\beta}{\alpha}$ determines how much the character will rotate towards the target as opposed to translating to reach it. For each iteration it moves and rotates the character closer to the target ik position. With α and β set to suitably high values it converges quickly (on the order of 10 iterations). We used $\alpha = \frac{1}{2}$ and $\beta = 1$. Should it happen that the character has both feet on the ground we execute the process twice, once for each foot, and then take the average of them.

The physical intuition for the iterative process is that $\boldsymbol{F}$ can be seen as a force, dragging the character closer to the desired point, and $\boldsymbol{T}$ can be seen as the torque applied to the character when the force $\boldsymbol{F}$ pulls the character at the position of the foot. It does not behave exactly like a physical system would do, but it is close enough that it can help with the understanding.

Let the final position and rotation after the last iteration be $\boldsymbol{p}^f$ and q^f. We record the offset of the final position and rotation from the original data as

$$\boldsymbol{p}^o = \boldsymbol{p}^f - \boldsymbol{p}$$
$$q^o = q^{-1} \cdot q^f$$

Once we have done this for each frame in the animation we have $\boldsymbol{p}^o$ and q^o for a subset of all frames. We fit b-splines to this data using the method described in [11] and we let the interpolated offsets be $\boldsymbol{D}_p^{\text{body}}(t)$ for the position and $D_q^{\text{body}}(t)$ for the rotation.

3.8.2 Bone rotations

After the body position and rotation has been fixed for each frame we will adjust the bone rotations of the character. For each frame we check if any of the feet are touching the ground. If a foot is touching the ground we need to move it and rotate it such that it is placed on the desired footprint. A leg has 7 degrees of freedom: 3 for the rotation of

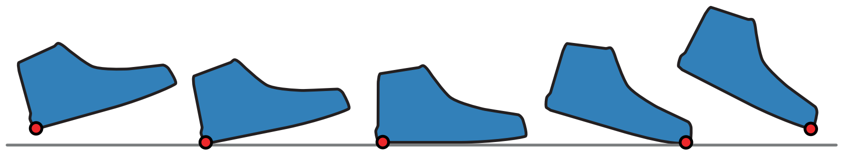


Figure 3.9: Foot IK point

the upper leg, 1 for the rotation of the lower leg as the knee is a hinge joint, and 3 for the rotation of the foot. The constraints that we want to apply, namely moving and rotating the foot, only constrains 6 of those degrees: 3 for the foot position and 3 for the foot rotation. Therefore there is one free degree of freedom which in this case turns out to be the rotation of the leg around the line from the hip to the foot. Most of the possible rotations are unrealistic for a human, but any IK algorithm will still have to find some way of determining this last degree of freedom. In our case it is reasonable that we want the rotations of all bones to be as similar to the original rotations of the bones as possible. We use the same hierarchical approach as described in [11] with the important difference that we only have to optimize for a single variable since the position and rotation of the character is already fixed. This brings down the number of variables from ≥ 7 to 1 which improves the performance significantly. Let the final rotation of bone i after this stage be given by $Q_i^b(t)$.

3.8.3 Foot position

To provide the greatest amount of realism, we cannot simply use IK to move for example the center of each foot to where it should be. If we study how a person walks (figure 3.9), we see that initially the heel is placed down, then the whole foot and finally the heel is raised and only the toes are still in contact with the ground. Using only the feet's bone positions, which are usually located at the ankle or heel, we risk constraining the feet and for example preventing the heel from leaving the ground for a short duration of time. Alternatively we would have to shorten the duration at which we apply IK to the feet, which could increase the amount of foot sliding. Therefore we propose that the optimal position used for IK is determined by the closest point on the foot's surface to the ground. More practically we use whichever one of the heel position and the toe position that is closest to the ground.

In figure 3.9 we indicate the ik point using a red dot. Note that in the center of the image the foot is horizontal, so the ik point could just as well have been close to the toes. A similar system is further discussed in [8].

3.9 Putting it all together

We now have everything we need to be able to animate the character. We know how the body should be positioned and oriented at each point in time and we know the bone rotations. To summarize, when we let the character play out its animation, we let for each point in time t

- the position of the character be $\boldsymbol{D}_p^{\text{body}}(t) + \boldsymbol{R}^D(t)$,
- the rotation of the character be $D_q^{\text{body}}(t) \cdot Q^D(t)$,
- and the rotation for each bone i in the character be given by $Q_i^b(t)$.

As the description of the implementation is now finished, we will continue by describing how the system was evaluated and how it performs in various scenarios.

Chapter 4

Evaluation

We have implemented the system in C# using the Unity Game Engine [1]. For the purpose of evaluation we have constructed a few different test scenarios. We describe these below and list the performance in table 4.1. The accompanying video shows the same test cases. The system has been evaluated on a computer with an Intel i7-4790K processor at 4 GHz. Since the graph searching stage was not an area of focus for this paper, not much work has been put in to optimize it and therefore we do not include it in the performance results.

4.1 Scenarios

Jumps

In the "Jumps" scenario (figure 4.1) the character starts walking forwards, then reaches a bend, starts running and then jumps over a gap in the platforms. The character the rounds a sharp bend, jumps a second time and finally slows down and comes to a full stop. The jump animation that was used starts and ends with the character running, so our system has ensured that the character smoothly starts to run before it reaches the point where it transitions to the jumping animation.

This scenario has a sharp bend between the two jumps which is deliberate. In previous tests other experimental locomotion systems were tested with this scenario and that sharp bend turned out to be particularly difficult to handle well, in particular for real-time controllers (as discussed in the introduction).

[1]Unity 5.5.2, see `http://unity3d.com`

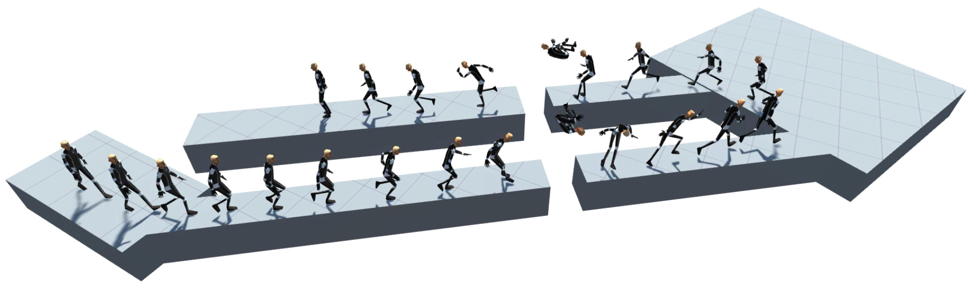


Figure 4.1: The Jumps scenario.

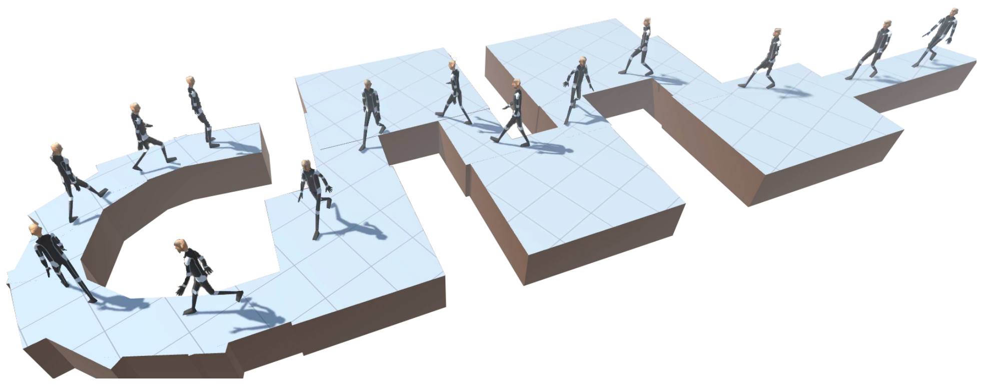


Figure 4.2: The Turns scenario.

Turns

In the “Turns” scenario (figure 4.2) the character walks along a sequence of bends and ends with walking along a curved platform. The character has been prevented from running in this scenario.

This test is primarily used to evaluate how natural it looks when the character transitions between walking in different directions.

We show how the motion looks and how the system performs when the character is tasked with moving in the above scenarios in the next section.

4.2 Results

In the Jumps scenario it can be noted that when we disable the speed adjustment of the character (see section 3.7.2) by preventing the walking

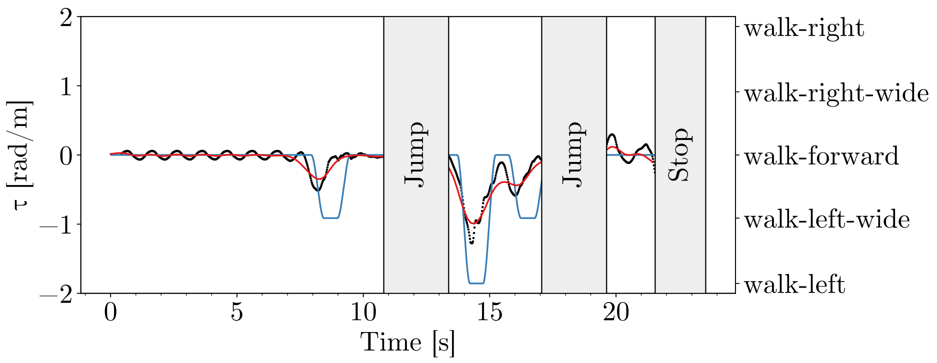


Figure 4.3: Curvature for the Jumps scenario. The original blend parameter (when transitioning between individual clips) is in blue. The estimated curvature is in black and the final blend parameter after the low-pass filter has been applied is in red.

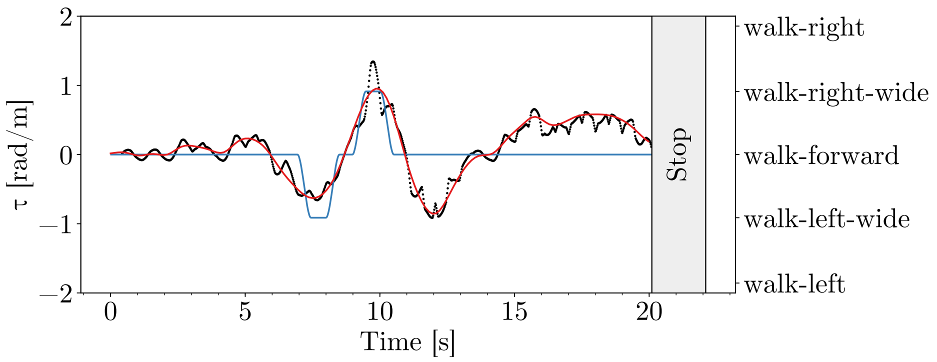


Figure 4.4: Curvature for the Turns scenario using the same colors as in figure 4.3.

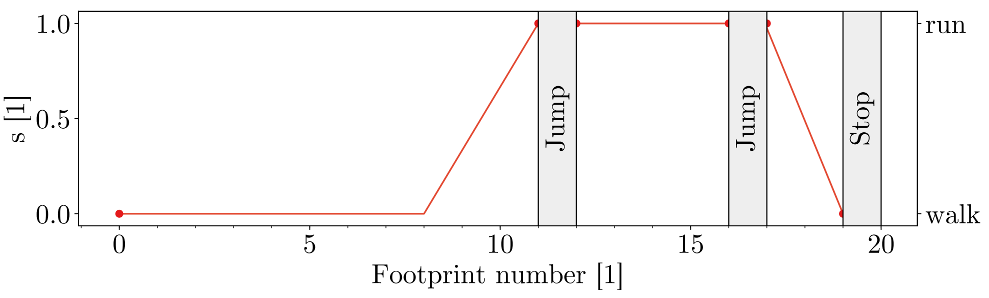


Figure 4.5: Speed blend weights in the Jumps scenario.

and running animations to be freely blended between, it looks distinctly unnatural as the character has to quickly transition from walking to running right before the jump starts as well as quickly transition to walking after the last jump ends.

In the turns scenario we note that if we disable the interpolation between different types of walking animations, the system will use the walk-forward animation for most turns in this scenario since the turns are not that sharp. The result is that the character does not lean slightly inwards when turning which does impact realism negatively. It is not a large effect, but it is noticeable. This is clearly visible in figure 4.4.

If we allowed the character to run in the turns scenario it turned out that the system simplified the path so that the character didn't follow the curved shape of the platforms, but instead took slightly longer steps and ran across the gaps between the platforms. However since the simplified path was significantly shorter than the initial curved path, it turns out that this produced a slightly uncanny animation. This effect is further discussed in section 4.3.

As images rarely do animations justice, the reader can take a look at the accompanying video to see a better visualization of the results. There are several interesting conclusions that we can draw from the results and we will discuss them in the next section.

4.3 Discussion

When implementing our system we have observed a few behaviors and effects that are of particular note. There are also several potential improvements that can be made to the system to make it both higher quality as well as improving the performance of it, both of which are

	Jumps	Turns
Number of footprints	51	43
Numer of frames	1384	1314
Smoothing	$< 1\,\text{ms}$	$< 1\,\text{ms}$
Refinement	11 ms	7 ms
Trace Body Trajectory	30 ms	52 ms
Blend Groups	3 ms	3 ms
Inverse Kinematics	159 ms	185 ms
Total time	204 ms	248 ms
Average time/frame	0.15 ms	0.19 ms

Table 4.1: Performance results for the different scenarios.

important for practical applications.

Footprint count

As our system uses blend groups to allow walking animations and running animations to be freely blended between, it often happens that the character runs instead of walks when traversing a sequence of footprints originally constructed from walking animations. As observed in [17] there is a highly linear relationship between the speed of a human and the distance between each footprint (the stride length). However when we adjust the character's speed by making it run we do not alter the *number* of footprints and thus the distance between them will remain roughly the same (unless the refinement stage in section 3.5 can manage to move them further apart). This can also happen if the smoothing stage shortens the path a significant amount. The result is a character that is running but with the same distance between the footprints as a walking character. We have observed that this gives an uncanny feel to the animation. In future work it may be possible to extend the system to adaptively remove or add footprints when necessary to resolve these cases.

Navigation graph

For the system to work well the navigation graph (see section 3.2) has to be relatively dense. This uses a lot of memory and is quite slow to generate which makes it infeasible for large worlds and worlds which change frequently. Even though it is very dense, a significant amount of information is simply thrown away when smoothing and refining the path. This raises the question of if it might be possible to generate the initial sequence of footprints in some other way as very little of the detail in the navigation graph is actually used. We think a promising approach would be to first plan a path (a sequence of points in this case) using a more high level navmesh [3] and then convert this path to a sequence of footprints and animations by finding the animations that best matches the path locally. For more difficult regions such as very tight spaces or jumping over a gap, some pre-processing could be done to add these to the navmesh. This would likely have to be combined with some adaptive approach as described above. It is however unclear if this approach would be able to handle as complex environments.

Dynamic environments

The main limitation for using this system in dynamically changing environments is currently the navigation graph (see section 3.2) as it takes a relatively long time to calculate it (on the order of seconds or tens of seconds, which is a long time in the context of a world that is constantly being changed). So as above we note that an important avenue for future work would be an alternative way of either constructing the navigation graph or finding the initial path in some way that does not require a navigation graph.

Speed limitations

In section 3.7.2 we adjust the speed of the character. In our system we only consider some types of boundary values that determine the speed of the character. However there are more of them. For example a running character can usually not turn as quickly as a walking character, so if the character has to round a corner in a tight corridor it should naturally have to slow down. Our system does not consider this however and would round the corner at full speed. It would look decent, but slightly unrealistic. Future work could be done to take this limitations into account.

Performance

As performance is a very important part of any game. The algorithms used in one need to have a focus on high performance. This system has a high initial cost of calculating a path. However once that is done, the character can follow the path with almost no overhead for several seconds. Therefore it makes more sense to look at the amortized performance, or the average CPU time per frame. This system requires just below $0.2\,\mathrm{ms}$ per frame on average which brings it well within the range for a high performance game, though it can likely not be used with more than a very small number of characters at a time. However it is not unreasonable to handle different characters in the game differently, with characters further away using lower quality locomotion.

Inverse Kinematics performance

Inverse kinematics is still the part of the system that takes the most time. Even though the number of variables were reduced to 1, we still use a search to find the best rotation for the legs. However there exists some measures of the error which can be solved analytically [20]. This means the solution probably would not be optimal in the sense that we use in this paper, but it would be pretty close. Most importantly, right now the solver converges in about 15-20 iterations, however with an analytical solution we would only have to run a single iteration. This would reduce the required CPU time for the inverse kinematics stage by an order of magnitude.

We conclude this thesis with a brief summary in the next section.

Chapter 5

Conclusions

We have presented a locomotion system that can be used in various interactive media to produce natural motion for bipedal characters. The amortized performance of the system (just below $0.2\,\mathrm{ms}$ per frame) is high enough to be used in for example games, even though it is not fast enough for any large group of characters. We have identified several improvements that build on top of the work by [2], but there are still many opportunities for further improvements. Compared to systems based on real time controllers [19, 12] this system can guarantee that the character will reach the goal, however it cannot react to dynamic environments as easily. If we on the other hand compare it other motion graph based approaches [10, 15, 6] this system is relatively performant but does not produce as high quality motion as those approaches.

Chapter 6

Bibliography

[1] E. Catmull and R. Rom. A class of local interpolating splines. In R. E. Barnhill and R. F. Riesenfeld, editors, *Computer Aided Geometric Design*, pages 317 – 326. Academic Press, 1974.

[2] M. G. Choi, J. Lee, and S. Y. Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Trans. Graph.*, 22(2):182–203, Apr. 2003.

[3] D. Demyen and M. Buro. Efficient triangulation-based pathfinding. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1*, AAAI'06, pages 942–947. AAAI Press, 2006.

[4] M. L. Gleicher. Graph-based motion synthesis: An annotated bibliography. In *ACM SIGGRAPH 2008 Classes*, SIGGRAPH '08, pages 49:1–49:11, New York, NY, USA, 2008. ACM.

[5] F. S. Grassia. Practical parameterization of rotations using the exponential map. *J. Graph. Tools*, 3(3):29–48, Mar. 1998.

[6] R. Heck and M. Gleicher. Parametric motion graphs. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 129–136, New York, NY, USA, 2007. ACM.

[7] I. Horswill. Lightweight procedural animation with believable physical interactions. In *In Proceedings of the Artificial Intelligence and Interactive Digital Entertainment*, 2008.

[8] R. S. Johansen. Automated semi-procedural animation for character locomotion. Master's thesis, Aarhus University, 2009.

[9] S. Jörg, A. Normoyle, and A. Safonova. How responsiveness affects players' perception in digital games. In *Proceedings of the ACM Symposium on Applied Perception*, SAP '12, pages 33–38, New York, NY, USA, 2012. ACM.

[10] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. *ACM Trans. Graph.*, 21(3):473–482, July 2002.

[11] J. Lee and S. Y. Shin. A hierarchical approach to interactive motion editing for human-like figures. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 39–48, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[12] W.-Y. Lo and M. Zwicker. Real-time planning for parameterized human motion. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '08, pages 29–38, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[13] A. Menache. *Understanding Motion Capture for Computer Animation and Video Games*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[14] R. Parent. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[15] A. Safonova and J. K. Hodgins. Construction and optimal search of interpolated motion graphs. *ACM Trans. Graph.*, 26(3), July 2007.

[16] K. Shoemake. Animating rotation with quaternion curves. *SIGGRAPH Comput. Graph.*, 19(3):245–254, July 1985.

[17] R. Tanawongsuwan and A. Bobick. A study of human gaits across different speeds, 2003.

[18] D. Tolani, A. Goswami, and N. I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graph. Models*, 62(5):353–388, Sept. 2000.

[19] A. Treuille, Y. Lee, and Z. Popović. Near-optimal character animation with continuous control. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[20] S. Yeung. Inverse kinematics (two joints) for foot placement. `http://gamasutra.com/view/news/129168/Inverse_Kinematics_two_joints_for_foot_placement.php`, Jan 2012.

Appendix A

A.1 Catmull-Rom spline

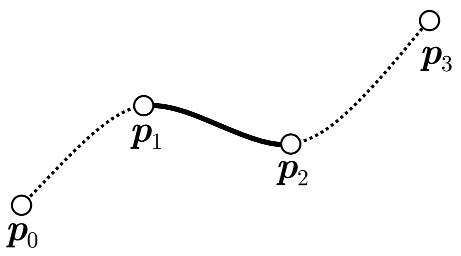


Figure A.1: Catmull-Rom spline. Note that the spline only passes between $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$.

The Catmull-Rom spline [1] is defined for $t \in [0, 1]$ as

$$\boldsymbol{p}_{\boldsymbol{c},\boldsymbol{r}}(\boldsymbol{p}_0, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, t) = \begin{bmatrix} \boldsymbol{p}_0 & \boldsymbol{p}_1 & \boldsymbol{p}_2 & \boldsymbol{p}_3 \end{bmatrix} \begin{bmatrix} 0 & -\tau & 2\tau & -\tau \\ 1 & 0 & \tau - 3 & 2 - \tau \\ 0 & \tau & 3 - 2\tau & \tau - 2 \\ 0 & 0 & -\tau & \tau \end{bmatrix} \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

where $\tau = \frac{1}{2}$ for the centripetal Catmull-Rom spline.

A.2 Quaternions

Quaternions are an extension of complex numbers. In the same way as an imaginary number can represent a rotation in 2D space, quaternions can represent rotations in 3D space [5].

A.2.1 Logarithm

Quaternions can be decomposed into a unit vector part $\hat{\boldsymbol{v}}$ and a scalar part θ which are to be interpreted as a rotation around the axis $\hat{\boldsymbol{v}}$ by θ radians. We can define the logarithm of a quaternion as

$$\mathbf{ln}(q) = \mathbf{ln}((\theta, \hat{\boldsymbol{v}})) = \theta\hat{\boldsymbol{v}}.$$

As any vector be uniquely be decomposed into its length multiplied by a unit vector we can analogously define the exponential of an arbitrary vector $\theta\hat{\boldsymbol{v}}$ as

$$\exp(\theta\hat{\boldsymbol{v}}) = q.$$

One advantage of transforming quaternions to vectors is that weighted sums of vectors make sense mathematically. In contrast weighted sums of quaternions in general do not make sense unless they lie very close to each other. Two quaternions can be interpolated between using *spherical linear interpolation* however it does not generalize to more than two quaternions.