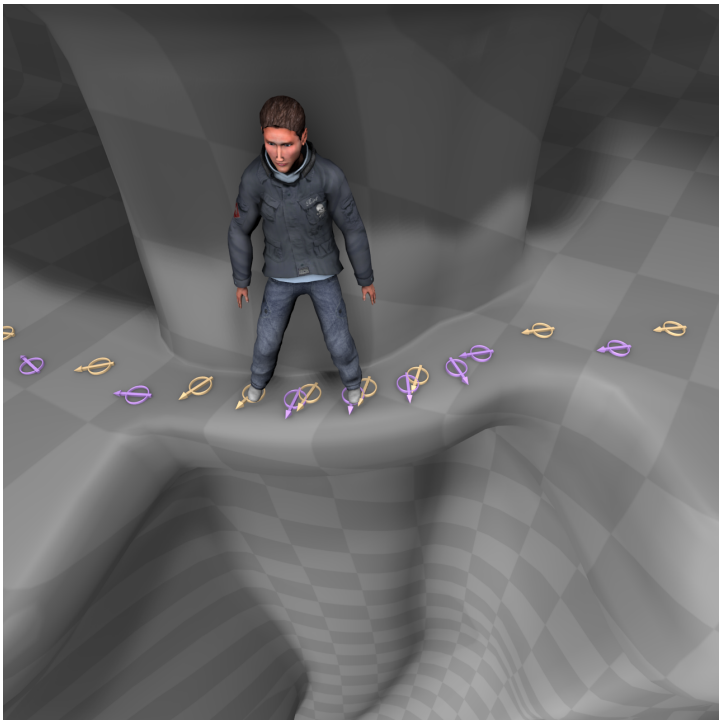
Stride Space: Humanoid walking animation interpolation using 3D Delaunay databases

Sybren A. Stüvel - Utrecht University
3371506



Supervisor: Dr. Ir. Arjan Egges
Thesis number: INF/SCR-09-63
June 2010

Acknowledgements

I would like to thank all the people that have made this research possible. Special thanks go to Dr. Ir. Arjan Egges, who has introduced me to motion capture techniques, given me a scientific basis in the field of computer animation, and provided the concept for this research. I thank MSc. Ben van Basten for his help and interesting discussions, and Drs. Arno Kamphuis for his critical and sometimes different point of view.

Additionally I would like to thank Ton Rosendaal and the Blender community for carefully and energetically fathering and developing Blender, which I have used extensively throughout my research. And Albert Heijn for their excellent Perla Dark Roast beans.

Finally I thank my girlfriend, parents and friends for their inspiration, support, enthusiasm and fondness for elegance.

Typesetting in LaTeX, TeXlive 2009-7

Editing in VIM 7.2.330

Abstract

Precise control over foot placement during character locomotion is crucial to avoid obstacle collision and to produce natural results. We present a new exact parameterization technique for generating humanoid walking animations. Given a database of pre-recorded motion capture data we generate new animations using a spanning neighbours search in a Delaunay database and interpolating those neighbours. Our approach results in exact foot placement while soft constraints such as timing are also taken in account, due to a novel blend candidates selection strategy. We show that this can be done very efficiently as to be compatible with real-time applications.

Keywords: computer animation, generation, interpolation, real-time, walk

Contents

CHAPTER 1

Introduction

Computer animation plays a very important role in contemporary games and simulations. The more realistic the environment, the higher the expectations of the realism of animation. Motion capture provides a way of recording natural motions, but by itself it is not suitable for interactive environments as using the motion capture data in itself can be compared to playing back a recording. To interpret, adjust and merge the data such that it can be used in an interactive setting a different approach is required.

We present a novel method for generating humanoid walking animations based on example motions from a corpus of pre-recorded motion capture data. The motion clips are interpolated in such a way that the result more accurately matches a query than any single one of the original animations. More specifically, we look at a way to create an animation of a walking humanoid based on a list of foot plant positions.

The foot plant positions are assumed to correspond to more or less natural walking motions. They can be manually entered, which is tedious work, or automatically generated based on a desired path and information about the environment. The latter method is being researched in our department at this moment. Based on results from biomechanics, step sizes and foot orientations can be estimated and incorporated in the planner, see for example Boulic et al.[BTT90]

Chapter 2 describes the related work, and establishes a context for our research. Chapter 3 shows the design overview. The offline and online processes are described in chapters 4 and 5. We show our body representation in chapter 6. Results are presented in chapter 7. The conclusion and future work are described in chapters 8 and 8.

Development was done in Visual C++ using both Visual Studio 2005 and Eclipse 3.5. We used the Real-time AGS Game Engine (RAGE) as the animation generation and real-time visualisation system. Blender was used to analyze and render the skeletal animations.

1.1 Notation

For series of variables, say $\{x_A, x_B, \ldots, x_Z\}$, we use a non-standard but quite self-explanatory shorthand notation $x_{A\cdots Z}$. The same is used for numerical indices such as $y_{1\cdots 4} \equiv \{y_1, y_2, y_3, y_4\}$.

CHAPTER 2

Related Work

We humans are very well trained in recognising human motion. From afar we can recognise a friend by the way she moves, even before we can recognise any other identifying traits. This makes it a tough challenge for animators to create realistic and identifiable walking animations. This challenge of course applies to human animators as well as their automated cousins.

2.1 Animation techniques

Three classes of animation techniques can be identified. *Procedural* techniques create locomotion based on biomechanical and empirical concepts. A second approach to generating human walking animations are *physical simulations*. The success of such an approach depends on the correctness of the physical model and the understanding of the human anatomy. Besides having a physical model, the character's motions needs to be governed by a control algorithm.

Hodgins et al.[HWBO95] describe such an algorithm. Their approach is more generic than our approach as they can animate running, jumping and cycling while we only animate walking. However, their approach is based on repetitive motion whereas we assume no repetition. Another important difference is that their method is aimed towards animating athletes, i.e. people that are physically fit and perform the required actions near-perfectly. Our technique is also usable for non-perfect walking animations such as limping or being drunk.

To work around the difficulty of manually creating a suitable control algorithm, it can be learned instead using neural networks and evolutionary programming[AF09]. The resulting algorithms are unfortunately not yet stable enough;

after a limited walking distance of a few meters the character topples.

One of the earliest techniques that use an explicit foot plan is by Van de Panne[vdP97]. He uses a kinematic model of the walking entity to generate the animation. The foot plan is used to create a path for the centre of mass, after which the path and foot plant positions are used to generate a walking motion.

A third class is comprised by *example-based* techniques. Rose et al.[RCB98] and Unuma et al.[UAT95] use interpolation to combine example motions into synthesized motions. Rose et al. use a high-dimensional interpolation space that contains not only dimensions such as "walking" or "running" but also emotional state such as "happy" or "clueless". Unuma et al. also use "emotion-based" animation techniques, but use Fourier analysis on repetitive motions. The focus of both papers is on generating realistic and controllable human motion while requiring little example motions. This is different from our approach, as we focus on positional accuracy of the feet as well as retaining the characteristics of the original motion. We feel that physical correctness is important for a virtual environment, as a character floating with their feet and hands half-way between the steps of a ladder can instantly destroy the feeling of presence and the suspension of disbelief, regardless of the emotional state expressed by the character's motion.

2.2 Manipulating motion capture data

The process of *motion capture* starts by suiting up an actor in a special suit. This suit has highly reflective markers attached to it, which are tracked by an array of cameras. Those cameras are positioned in such a way that ideally every marker can be tracked by at least two cameras at any point in time, regardless of the position and pose of the actor. The markers are recorded at 100 frames per second, and when the system is properly calibrated with sub-millimetre precision. After the markers have been recorded and labeled they are mapped onto a virtual humanoid figure. This figure then drives a skeleton, of which the movements are stored. When referring to "motion capture data" we refer to such motion capture-based skeletal animations.

One of the problems of controlling motion capture data stems from the many degrees of freedom in the human skeleton. An animator may be able to control

all these degrees of freedom separately, but this easily leads to unnatural motion. To aid in this motion graphs[KGP02] are often used to enhance the motion capture data with a graph describing which animations can be blended, and at which frames this blend can happen. This technique allows, for example, to smoothly transition between different walking animations. The downside of motion graphs are that often manual labour is required, modifying the motion capture data to ensure that blends are possible where needed. Another problem is that the blending can only occur at the blending points defined in the graph, introducing artificial movement when interactive control is needed; the current animation will keep playing until a suitable blend can be performed, even if this conflicts with the user's input. To ameliorate this the graph could be enhanced to contain more blend points, but this in itself can result in worsened performance.

Choi et al.[CLS03] sample the collision-free physical space and build a roadmap of possible animations that can let a character walk from one sample point to the other. This method is limited to static environments. Instead of adapting entire walking motions, we manipulate smaller clips of single step animations; every step can be generated for the then-current state of the environment.

A method quite similar to ours is the Step Space method by Van Basten et al.[BPE10]. They too use motion capture data segmented into individual steps. A 10-dimensional parameter space is used to find the step animation that most closely resembles the query step. It is then aligned and fitted onto the previous step animation, producing a walk. The feet are then optionally moved onto the query positions by applying *inverse kinematics*. The animation is time-warped to attain natural pelvis speeds. Instead of finding the step that resembles the query the most, we find four of such steps, in such a way that in our parameter space the query is embedded in the space spanned by those steps.

2.3 Interpolation of animations

In our experiment "StepSpace Interpolation"[Stü09] we have looked at interpolation of humanoid step animations. This paper builds on that experiment and refines the technique. The goal of our algorithm is to generate a walking animation based on foot plant positions and motion capture clips. We generate

new animations by interpolating between up to four existing animations per step, and concatenating the result.

In the aforementioned experiment we keyframed the motion of the human body. An animation consists of a mapping of time codes to keyframes. Each keyframe K_i consists of orientations for each joint, expressed as quaternions. We showed that interpolating between keyframes by interpolating the quaternions, using weights obtained from a linear parameter space, does not yield a desirable animation. The animation does show a walking humanoid, but the feet do not end up in the correct positions.

Park et al.[PSS02] present a method of generating locomotion based on motion blending. Globally their method is very similar to ours, in that they too use motions scattered in parameter space, which they interpolate to produce the final animations. Their parameters are *style*, *speed* and *turning angle*. They blend the joint orientations and root position based on weights in this parameter space. Precise foot placement is not possible. Heck et al.[HG07] generate a variety of animations, such as walking, cartwheeling and punching. They create a highly structured parameterized motion graph to blend between motions. It is possible to use the technique to have a character reach a certain position, such as a square on a grid, but it does not provide foot placement.

Inverse kinematics is a common way to correct the feet. However, care has to be taken by applying such a correction as it can lead to imbalance[Pee09]; this may happen when the feet are both moved in the same direction, for instance (see figure 2.1). Correcting for this imbalance is not a trivial matter, and requires considerable computation. Our technique remains balanced by interpolating between balanced animations.

There are obviously more ways to correct the animation after interpolation. However, we suspect that by storing the motion data in a more linearized representation we can interpolate the animation and ensure the feet end up at the correct positions, without requiring corrections afterwards. Interpolation based on joint positions instead of orientations would guarantee this. It has been used by Guo et al. to interpolate based on a low-dimensional ($D \leq 3$) parameter space[GR96]. Positional interpolation of all joints is known to cause bone stretching.

It may be beneficial to create motions using a simplified representation of the skeleton, rather than the high-DoF human skeleton. Monzani et al.[MBBT00]

Figure 2.1: Findings by P.W.A.M. Peeters[Pee09]. The left image shows the result of an interpolation method. The character is balanced but the feet are not in the correct position. After moving the feet using inverse kinematics the character is no longer balanced (right image).

and Popović et al.[PW99] use a lower-DoF skeleton and recalculate the remaining DoF. Kulpa et al.[KMA03] use a representation of the human body that introduces "limbs of variable length", a way of representing the body in a morphology-independent model, which we use to avert the problem of bone stretching.

2.4 Research goals & motivation

We try to solve the *stepping stone problem* by interpolation of example motions:

> Given a set of query foot placements, called a foot plan, that contains temporal and spatial constraints, generate an animation that adheres to these constraints.

In our problem setting, we consider the feet positions as hard constraints and feet orientation and temporal constraints as soft constraints. As we have described in the previous sections, not many techniques allow exact foot placement. We propose a novel parameterization technique that efficiently generates exact results.

We chose an *example based* method, as such a method allows for subtleties in the motion that are very difficult to obtain using other methods. An actor can be easily instructed by a director to walk in a way that could be difficult to quantify for use in a physical model or procedural technique, such as “airy”, “child-like” or “sneaky”.

CHAPTER 3

Design and Implementation

This chapter describes the design and implementation details of our algorithm. It starts with an explanation of the choice of parameter space and the way we normalize foot steps. We then continue to the selection of blend candidates, calculation of the weights, the interpolation and the concatenation of steps.

A *footstep* is considered as one foot staying on the ground while the other foot moves from one position on the ground to another position on the ground. The foot that remains on the ground is called the *supporting foot*, and the other foot is called the *swing foot*. A step thus starts and ends with both feet on the ground, called a *double stance*.

3.1 Overview

Our system can be divided into an offline and an online phase. In the offline phase we create the data structures that allow for fast motion synthesis. In the online phase these structures are queried and the resulting motion is rendered. An overview is given in figure 3.1.

The offline phase (chapter 4) consists of the following steps:

1. We automatically segment a corpus of motion capture data into clips of individual steps.
2. The clips are normalized by time warping, translating and rotating.
3. Automatic clean-up is applied to the clips.
4. We store these steps in a 3D parameter space S_{low} using a Delaunay

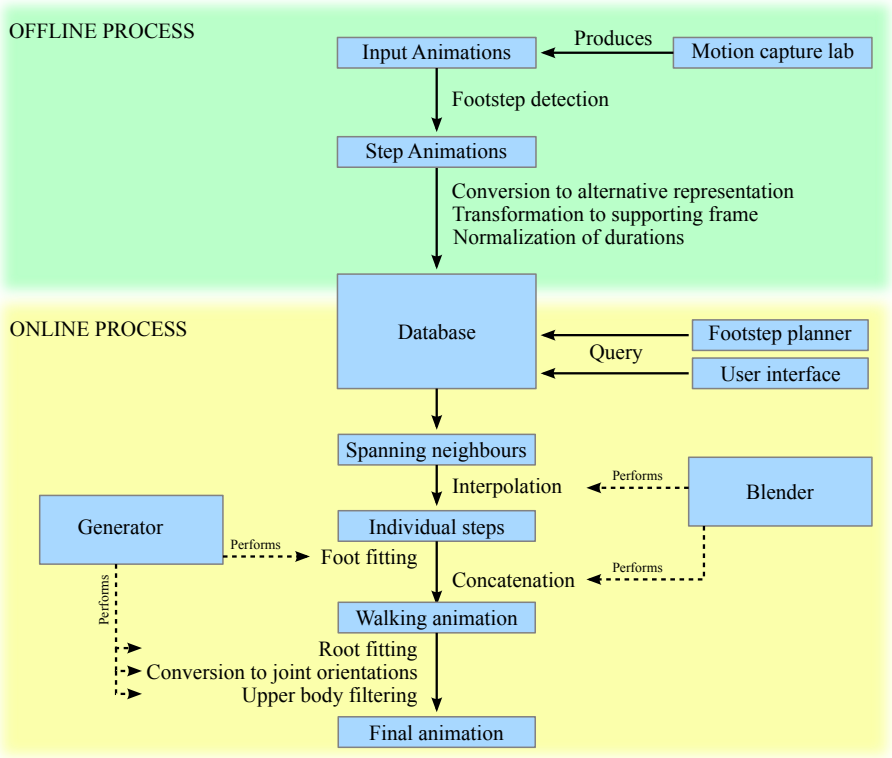


Figure 3.1: Global overview of the StrideSpace method.

tetrahedralization. The steps in S_{low} are represented in our alternative body representation (chapter 6).

The parameter space S_{low} is considered "low-dimensional" as it is an under-parametrization of the footstep. It does not take orientation nor timing into account. We use a higher dimensional distance function in section 5.1.

In the online phase (chapter 5) the user or footstep planner supplies a query foot plan. Then, for each step in this query:

1. The query step is transformed to the lower-dimensional parameter representation.
2. The system determines the blend candidates in S_{low}. These are the vertices of the tetrahedron containing the query step.
3. Based on these blend candidates, the system evaluates nearby blend candidates using a higher-dimensional distance function.
4. The final blend candidates are blended using both a rotational and positional interpolation scheme.
5. The generated step is aligned and fitted to the previous step.

3.2 Choice of parameter space

We discuss positions both in world coordinates and in a local coordinate system called the *supporting frame*. The symbols used to denote the positions of the feet are:

	world	local
Supporting foot	W_{sup}	C_{sup}
Swing foot, initial position	W_{from}	C_{from}
Swing foot, final position	W_{to}	C_{to}

In order to add a footstep animation to the database, its parameters have to be determined. These are determined based on the supporting foot W_{sup}, the initial position of the swing foot W_{from} and the final position of the swing foot W_{to}. By assuming the step is performed on the ground plane, we can describe the step by three parameters.

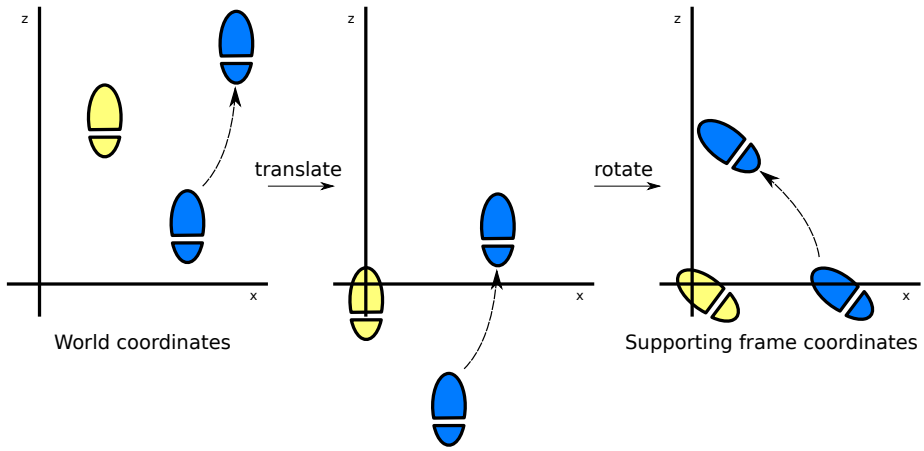


Figure 3.2: The transformation of a single step from world coordinates to supporting frame coordinates

We can create a right-handed *supporting foot coordinate frame*, or *supporting frame* for short, by applying a translation and a rotation to the animation (figure 3.2). The transformation changes W_{xxx} into C_{xxx} . The origin of the coordinate frame is placed at the supporting foot, C_{from} lies on the (positive or negative) x-axis and the y-axis is parallel to the ground plane orthogonal to the x-axis, forming a right-handed coordinate system. C_{from} is placed on the positive x-axis when the right foot swings, and on the negative x-axis otherwise. The parameter vector for the step is then given as:

$$\vec{P}(W_{sup}, W_{from}, W_{to}) = \begin{pmatrix} C_{from,x} \\ C_{to,x} \\ C_{to,z} \end{pmatrix}$$

3.3 The Canonical Step

The *Canonical Step* is a single footstep animation in normalized form. Each step starts and ends with a *double stance*, a posture in which both feet are resting on the ground. There is always one *supporting foot* that remains on the ground for the entire duration of the step; the other foot is called the *swing foot*. Walking is generally distinguished from running in that there is always one foot on the ground, and during a brief period between the swings both feet are.

The swing foot has two relevant positions, at the middle of each stance before and after the swing, C_{from} resp. C_{to}. The supporting foot has only one position C_{sup}, as we assume it does not move during the step. It is determined at the keyframe in the centre of the initial stance of the swing foot, i.e. at the same moment in the animation C_{from} is determined. The centre of the periods of double support is used as we expect it to be the most representative of the start and end location of the step. For deducing C_{from} an earlier keyframe may be influenced too much by the previous step in the original motion capture data, and a later keyframe may already be part of the swing that was not recognised as such by the footstep detector due to noise. A similar argument can be made for C_{to}.

The step's local coordinate frame is positioned with the origin at the supporting foot, the x-axis pointing at C_{from}, the y-axis pointing upward through the foot, and the z-axis orthogonal to both completing the right-handed frame. In other words, the coordinate frame is chosen in the same way as the supporting frame described in section 3.2. This means that C_{sup} is always $(0, 0, 0)$.

The canonical step also includes information about the durations of the swing and the two periods of dual support. This information is used to determine the speed of the blended step. The animation is timewarped in a piecewise linear fashion to predefined durations and resampled before insertion into the database. This ensures that all stored animations have the same duration and the same number of frames, speeding up the blending process.

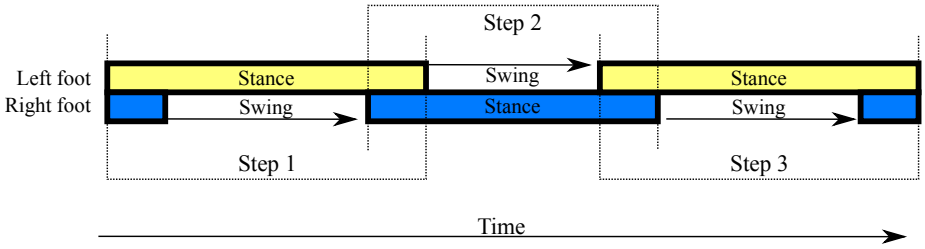


Figure 3.3: Schematic view of a walk. The yellow and blue areas represent those moments in time where the left resp. right foot touches the ground. We blend between steps in the periods of double support.

CHAPTER 4

Creating the Stride Space

In this chapter we elaborate on the offline construction of the parameter space. This three-dimensional parameter space is called the *Stride Space*. We first look at the way the input animations are segmented into separate steps in section 4.1. Section 4.2 describes our three-dimensional database system. Section 4.3 tries to answer questions like "how usable is my database for this technique?" and "what steps are missing?"

4.1 Step segmentation

We extract individual steps from the motion capture data by determining the moments at which the feet are planted. This in general is not trivial due to noise and retargeting errors. The footstep detector needs to be precise in order to get proper segmentation and eventually a decent parameterization. We use a height- and velocity-based footstep detector[BE09].

The individual step animations are now converted to our alternative body representation. All operations are performed in this representation, until the generated animation is handed over to the animation system.

Before determining the step's parameters, every step is cleaned up. Foot skating is removed by moving the root of every frame, such that the supporting foot is placed at the origin. To be able to work with noisy motion capture data we also adjust the feet to rest on the ground during the periods of support. Without this step the feet could end up several centimeters above or below the ground depending on the quality of the motion capture data and the parameters of the footstep detector. The feet are moved only in the vertical axis, so these

adjustments do not change the parameters of the step. The cleanup is done automatically and does not require any manual action.

4.2 The Delaunay Databases

To perform location queries a *Delaunay tetrahedralization* is created from the parameter points. Each parameter is a point in 3D space which makes the Delaunay tetrahedralization particularly useful. The Delaunay tetrahedralization of the parameter points combined with references to the step animations and metadata is called the *Delaunay database*. Performing a location query on the database results in the tetrahedron spanned by four vertices. These vertices are called the *spanning neighbours* and represent the parameters of the four animations that will be blended into the final animation. The properties of the Delaunay tetrahedralization ensure that those are four points that are relatively similar to the query point.

The side of the swing foot determines the sign of the first parameter. Let $P_L \in \mathbb{R}$ be the smallest real number such that first parameter $C_{from,x}$ of all left steps are in the interval $(-\infty, p_L]$. Let $P_R \in \mathbb{R}$ be the largest real number defined similarly for all right steps such that $C_{from,x} \in [p_R, \infty)$. When all blend candidates are inserted into the same database, querying in the interval $[p_L, p_R]$ may result in a mixture of left and right steps being blended. To prevent this, we separate the database in two parts, one for each side of the swing foot. As a result of this separation, extrapolation is necessary to process a query for a left step in the interval (p_L, ∞) or a right step in $(-\infty, P_R)$.

There are many ways in which walking animations can be classified, such as walking forward, turning, side-stepping and walking backwards. However, these classifications are not distinct; walking forward and decreasing the length of the swing results in slowing down and gradually results in walking backward. The same holds for the possibly gradual changes between walking straight and turning sharply. This means that it is impractical to separate the database even further based on these classifications. To decrease the likeliness that steps of different classifications are blended we use a higher-dimensional distance function – see section 5.1.

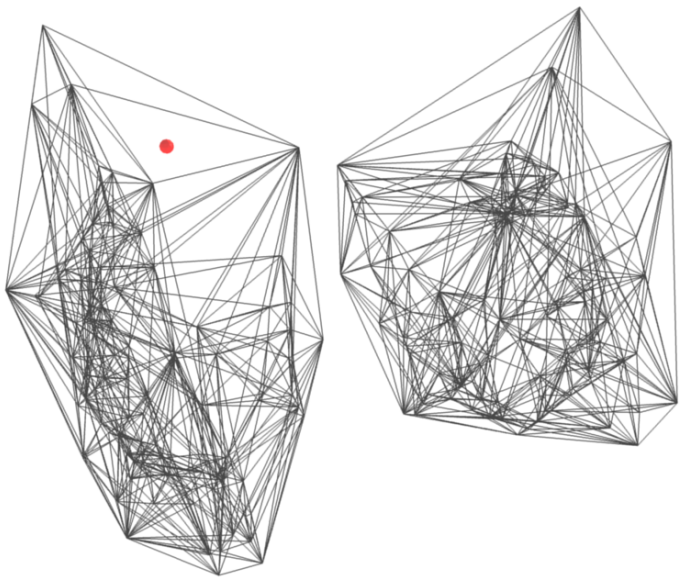


Figure 4.1: Visualization of the left-step and right-step Delaunay databases with a query point.

4.3 Database analysis

As our technique interpolates between steps, the largest steps in the database have to be larger than any of the query steps, and similar for small steps. We have provided an analysis algorithm that detects such potential problems. Previous studies [Hof65][EB79] have shown a relation between stride length (SL) and the trochanterion height (TH). When sprinting at 2.5 m/s the average SL:TH ratio was shown to be 1.10 for males and 1.11 for females. Tripathy [Tri04] measured an average ratio of 0.65 when walking. In our analysis of the database we use a SL:TH ratio of 1.0 to determine the maximum stride length.

We do not only look at the step length, but also regard the width of the stances, i.e. the distance between the feet. It is possible for a step to end smaller or larger than any step starts. This means that it is impossible to continue walking from such a step without resorting to extrapolation. The manually constructed foot plans have been constructed to avoid this situation.

Even though our database did not have a wide enough gamut to allow for a full spectrum of steps, it was nevertheless usable for the examples we have shown. Figure 4.3 shows the projection of the left-side database in the p_2/p_3 plane. The white disc in the centre represents the supporting right foot. The centres of the yellow discs represent the coordinates in the *supporting frame* of the final double stance, i.e. C_{to}. The radius of the yellow discs reflects the area in which this step is considered to be similar to other steps, which we have taken to be 15cm based on the size of a human foot. The blue circle represents the maximum step size as described above. Of course the parameters for the analysis can be adjusted for individual needs.

This is the output of the analysis software. A sidestep is defined as a step where the sideways movement of the swing foot is more than three times the forward movement. There is a chance of a sidestep being blended with a 180^o turn, which is why it is important to have sidesteps that end small in the database.

```
Importing ../final-bvh/L-test.cgal
Loading 94 vertices
Loaded left-step database, inverting XY-parameters.
You have a high enough density of steps that start small.
You have a high enough density of steps that start wide.
You should have steps that start smaller.
    The smallest width in DB is 18.0 cm but should be at most 15.0 cm
You should have steps that start wider.
    The widest width in DB is 77.2 cm but should be at least 95.0 cm
You have a high enough density of steps that end small.
You have a high enough density of steps that end wide.
You should have steps that end smaller.
    The smallest width in DB is 18.4 cm but should be at most 15.0 cm
You should have steps that end wider.
    The widest width in DB is 90.2 cm but should be at least 95.0 cm
Found 36 sidesteps.
You have no sidesteps that start small enough.
    Minimal width in DB is 18.7 cm, should be at most 15.0 cm
You have no sidesteps that end small enough.
    Minimal width in DB is 21.3 cm, should be at most 15.0 cm
You have a step that ends with the feet 90.2 cm apart, and 0 steps
    that start with that width or wider. You need at least 3 steps
    that are wider. The widest so far is 77.2 cm wide.
```

Figure 4.2: Output of the database analysis algorithm.

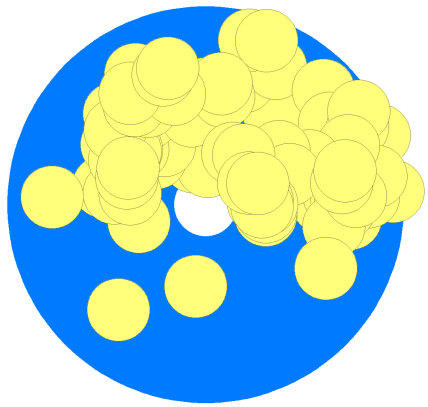


Figure 4.3: Projection of the p_2/p_3 plane of the left-side Delaunay database.

CHAPTER 5

Synthesizing the animation

This chapter describes the online process of the step synthesis. We first look at the determination of the blend candidates. The blend candidates are then interpolated using two strategies. The resulting steps are subsequently adjusted and concatenated into the final animation.

Appendix A contains the algorithm in Python-like pseudo-code. The details and motivations are described in the following sections. The code consists of three major components. The `Database` class contains the example motion clips and can perform location queries and find alternative blend candidates. The `Blender` class concatenates and interpolates animations. The `Generator` class queries the database to determine the weights and animations to use, then passes those to the blender to create the animations.

Section 5.1 shows how we determine the blend candidates from the query. In section 5.2 we describe the way the interpolation weights are determined. Rotational and positional interpolation are explained in sections 5.3 and 5.4. Time scaling is then applied in section 5.5. We describe the foot fitting algorithm in section 5.6, after which the steps are concatenated in section 5.7. Section 5.8 describes the filtering of the upper body motion.

5.1 Determining blend candidates

The parameter vector $\vec{q}$ is extracted from the footplan query. Based on the sign of the first parameter $\vec{q}_1$ we determine whether we need the database for left steps ($\vec{q}_1 < 0$) or right steps ($\vec{q}_1 \geq 0$). A location query on $\vec{q}$ is performed on the appropriate database to find the spanning neighbours $N = n_{1\ldots4}$ with

parameters $P = \vec{p}_{1\cdots4}$. Note that N represents the canonical steps, whereas P represents their respective parameters.

If $\vec{q}$ is found to lie outside the convex hull of the database, the nearest neighbour is used as the only blend candidate n_1, it is given weight 1.0 and further candidate selection and weight determination are skipped.

The four neighbours P are used to span four planes in $\mathbb{R}^3$ – see figure 5.1. The planes are oriented such that $\vec{q}$ lies on the positive side. Let H_i be the intersection of the negative half-spaces defined by the three planes intersecting p_i. All combinations of four points in $H_{1\cdots4}$, such that exactly one point is in each of $H_{1\cdots4}$, are guaranteed to span a tetrahedron around $\vec{q}$.

To determine alternative blend candidates for n_i we use a higher order distance function $D_{high}(\vec{q}, h_i)$ to determine the distance from $\vec{q}$ to all steps $h_i \in H_i$. If there is a step in H_i that is closer than n_i, it will replace n_i as blend candidate. D_{high} takes more information into account than our 3D parameters; it uses the foot orientation and timing information as well. The extra dimensions that we use are:

- orientation of the supporting foot,
- orientation of the swing foot at the initial support,
- orientation of the swing foot at the final support,
- duration of the swing,
- duration of the initial support of the swing foot,
- duration of the final support of the swing foot

The function D_{high} is defined as the weighted sum of the distances in all nine dimensions. The weights can be adjusted by the user of the system, in the three groups *location*, *orientation* and *timing*.

As the current distance function is based on the additional dimensions *orientation* and *timing*, this approach can be considered a simplified form higher dimensional querying. Rather than selecting the span set from a 3D parameter space using an additional 6D heuristic, a full 9D or higher parameter space could be used to achieve higher performance in blend candidate selection. Contrasting the S_{low} introduced in section 3.1 this space could be called S_{high}.

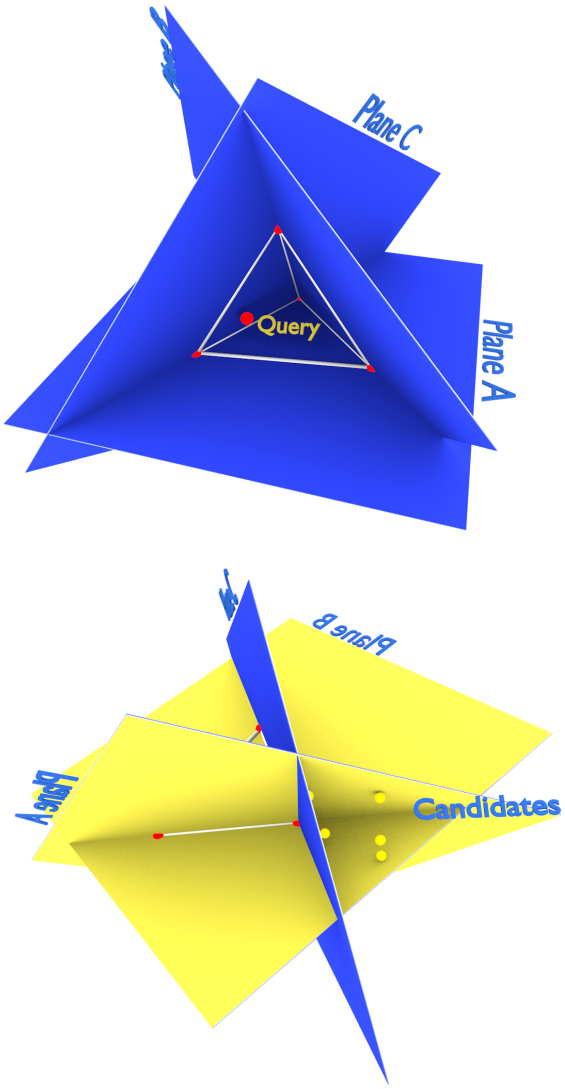


Figure 5.1: Both images show the same four spanning neighbours, from different points of view. The three planes define the frustum for the point furthest (top) and closes (bottom) to the camera.

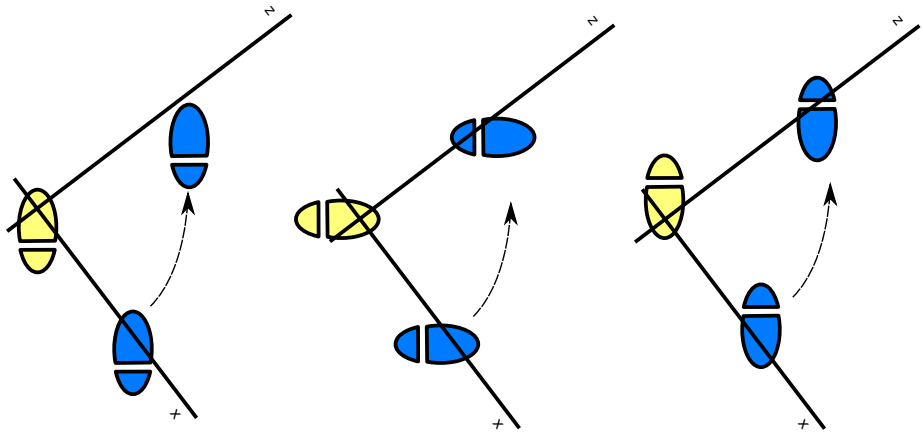


Figure 5.2: These three different steps all have the same parameters. They represent walking forward, cross-legged side-stepping and walking backwards.

The parametrization described in section 3.2 is an under-specification of the step animations. This manifests itself in the identical parameters for the steps shown in figure 5.2. The higher dimensional blend candidate selection can prevent blending different types of steps.

5.2 Determining weights for interpolation

The process of weighted interpolation starts by determining the weights. This is done for each footstep in the query. The goal of the weight determination process is to obtain weights $w_{1\ldots4}$ such that a weighed average of the spanning neighbours equals the query point $\vec{q}$:

$$\vec{q} = \sum_{i \in \{1,\ldots,4\}} w_i \vec{p}_i$$

In order to determine those weights, two *helper points* are used (see figure 5.3). The first helper point $\vec{h}_1$ is defined by intersecting the line $\overline{p_4, q}$ with the plane

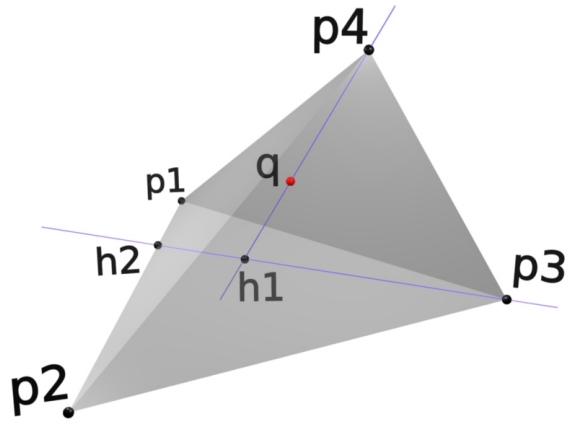


Figure 5.3: A tetrahedralization cell with spanning neighbour parameters $\vec{p}_{1\cdots4}$, query point $\vec{q}$ and helper points $\vec{h}_1$ and $\vec{h}_2$.

$\overline{p_1, p_2, p_3}$. The second helper point $\vec{h}_2$ is defined by intersecting lines $\overline{p_3, h_1}$ and $\overline{p_1, p_2}$. Once these points are obtained we determine weights to express them in linear combinations of $\vec{p}_{1\cdots4}$. This method fails when $\vec{q}$ is on the line $\overline{p_3, p_4}$, as in that case $\vec{h}_1$ is undefined. This is easily worked around by taking $w_1 = w_2 = 0$; determining w_3 and w_4 then does not require $\vec{h}_1$ nor $\vec{h}_2$. Other corner cases, such as when $\vec{q}$ is in the plane $\overline{p_1, p_2, p_3}$, are handled in a similar fashion. This solution is stable in that the resulting footstep will place the feet at the queried positions.

In general, to determine the weights w_1, w_2 such that $X = w_1 P_1 + w_2 P_2$ for points X, P_1 and P_2 we first check that they are collinear. The distances $|X - P_1|$ and $|P_2 - P_1|$ are then used to determine the weights:

$$\begin{aligned} w_1 &= \frac{|X - P_1|}{|P_2 - P_1|} \\ w_2 &= 1 - w_2 \end{aligned}$$

Combined with the fact that $|X - P_1| + |X - P_2| = |P_2 - P_1|$ the above formula gives rise to:

$$\begin{aligned} W_P(P_1, P_2, X) &= \frac{|X - P|}{|P_2 - P_1|} \\ \text{for } P &\in \{P_1, P_2\} \end{aligned}$$

We can now use this general formula to calculate the final weights:

$$\begin{aligned} \vec{h}_2 &\equiv W_{p_1}(\vec{p}_1, \vec{p}_2, \vec{h}_2) \cdot \vec{p}_1 + W_{p_2}(\vec{p}_1, \vec{p}_2, \vec{h}_2) \cdot \vec{p}_2 \\ \vec{h}_1 &\equiv W_{p_3}(\vec{p}_3, \vec{h}_2, \vec{h}_1) \cdot \vec{p}_3 + W_{h_2}(\vec{p}_3, \vec{h}_2, \vec{h}_1) \cdot \vec{h}_2 \\ \vec{q} &\equiv W_{h_1}(\vec{p}_4, \vec{h}_1, \vec{q}) \cdot \vec{h}_1 + W_{p_4}(\vec{p}_4, \vec{h}_1, \vec{q}) \cdot \vec{p}_4 \end{aligned}$$

The weights are not only used to blend the step animations, but also to determine the timing of the resulting motion clip, which we discuss in section 5.5.

5.3 Rotational interpolation

We store an animation A of a walking humanoid as a mapping from time, to the orientations of all joints in the skeleton:

$$A : t \rightarrow \{\textit{root}, \textit{right hip}, \textit{left hip}, \ldots, \textit{left subtalar}\}$$

These orientations are stored in quaternion representation. When represented as vectors $v \in \mathbb{R}^4$, quaternions are only valid when $|v| = 1$, which makes interpolation of more than two orientations rather bothersome. *Generalized quaternion interpolation* is an interpolation method that extends the quaternion SLERP algorithm. This generalized method can interpolate between more than two unit-quaternions, but is neither closed-form nor fixed-time[Wik09].

We transform the joint orientations to the exponential map representation[Gra98] for interpolation. The exponential map has some nice properties that make it suitable for interpolation, the most important of which is that every vector

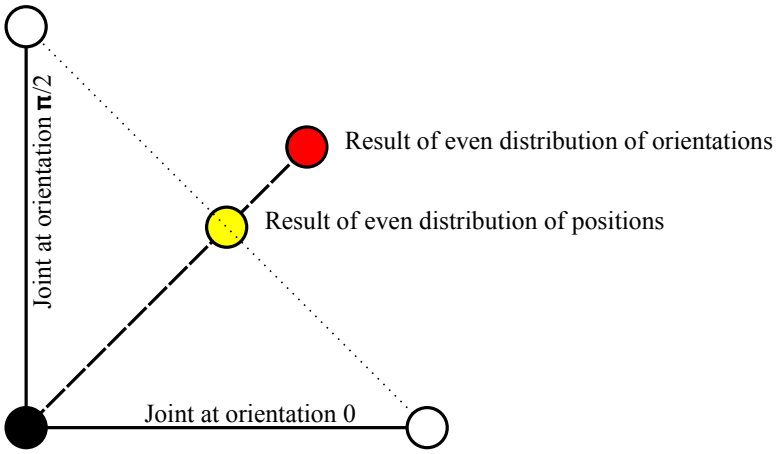


Figure 5.4: Two orientations of a joint, at 0 and $\frac{1}{2}\pi$ radians. The yellow circle shows the desired position. Based on the positions we would choose weights $w_1 = w_2 = 0.5$. The red circle shows the result of interpolating the joint orientations with those weights, demonstrating the difference in desired and realized position.

$v \in \mathbb{R}^3$ represents a valid orientation in this representation. In the exponential map representation a rotation in $\mathbb{R}^3$ is represented by a vector in the same orientation as the rotation axis, with length equal to the rotation angle. This representation introduces no loss of information.

The space of rotations is not linear, but we apply weights from our linear parameter space. The result is that the joints do not end up at the correct position, as shown in figure 5.4, and the parameters of the generated motion are different from the queried parameters. This difference becomes smaller as the interpolated orientations become more similar. By sourcing animations from nearby points of the Delaunay tetrahedralization we are certain we interpolate relatively similar animations. We have shown this in an earlier technical report[Stü09], and the results are visible in figure 5.5. For this specific terrain the rotational interpolation may be precise enough, but this does not hold for the general case.

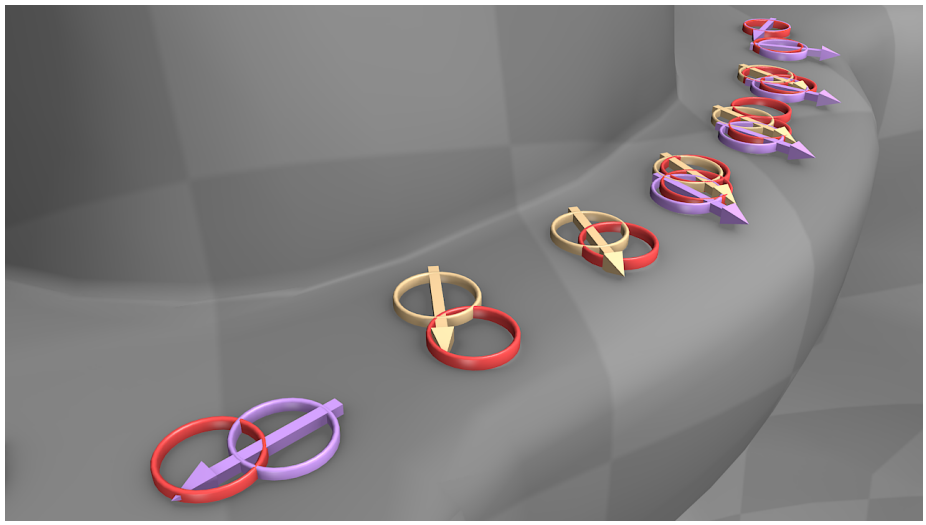


Figure 5.5: Standard parameterization techniques yield a high error. The yellow and purple rings are the query placements, the red rings are the resulting foot positions.

5.4 Positional interpolation

The difference between the interpolated foot position and the desired foot position is relatively small but not zero (see figure 5.5). A correct foot position can be guaranteed by interpolating joint *positions* instead of joint orientations. This introduces bone stretching, which we solve by replacing the thigh, knee and shin with a virtual *limb of variable length*[KMA03]. Both the orientation-based and linearized representations are kept parallel to each other. This allows us to compare the results of rotational and positional interpolation on all joints, utilizing the different outcomes to move parts of the skeleton in a consistent way. The process described below is performed on both legs.

After interpolation has been performed on all joint orientations the result is stored as S_{out}. The weights $w_{1\ldots4}$ are used to determine F_{pos}, the weighted

average of the *positions* of the subtalars in the four source animations[1]:

Let F_{orient}[2] be the position of the ball of the foot in S_{out}. Location constraints are added to the joints in the foot – ankle, subtalar and toe – by taking their location in S_{out} and displacing the joints by $F_{pos} - F_{orient}$. This places the foot at the correct location without affecting the foot orientation.

After the feet have been positioned correctly, the roll of the half-plane containing the knee is interpolated linearly using weights $w_{1\cdots4}$.

5.5 Time scaling

Every step is separated into three parts: the initial double stance, the swing, and the final double stance. Before interpolation starts those parts have to be of equal duration, or one animation's swing will be blended with another animation's stance. This is ensured by applying a variation of *registration curves* by Kovar et al.[KG03]

All step animations are stored in a normalized form such that the corresponding events in different animations occur at the same frame. After interpolation the animation is time-scaled in a piecewise linear fashion. We guess the correct timing by interpolating the duration of every part of the step by taking the weighted average of the source animation durations using $w_{1\cdots4}$. The result is depicted in figure 5.6.

For this process the original timings are used that were stored in the canonical step before the step duration was normalized. As the animation is only stretched in the temporal dimension no foot skating will be introduced.

5.6 Foot fitting

After all the individual steps have been generated they have to be processed before they can be concatenated. Without this processing the foot orientations

[1] The subscript "pos" denotes that it was obtained by interpolating joint positions.
[2] The subscript "orient" denotes that it was obtained by interpolating joint orientations.

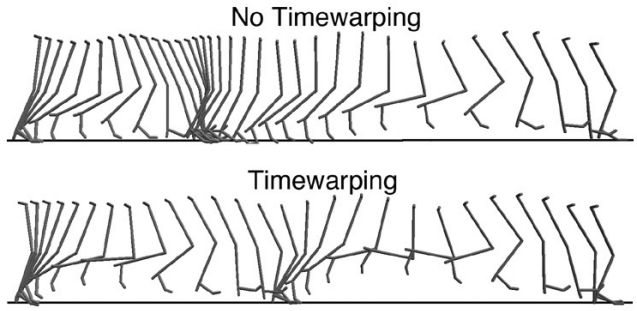


Figure 5.6: A transition between walking and jogging that spans two locomotion cycles. For clarity, only the right leg is shown. Without timewarping, out-of-phase frames are combined and the character floats above the ground with its legs nearly straight. Source: Kovar et al.[KG03]

can differ radically from one step to the next, resulting in rapid rotation of the feet during the brief periods of double support.

The orientation of the foot is determined by the normalized *foot vector* expressed in world coordinates:

$$F(pose) = \frac{\overline{subtalar \rightarrow toe}}{|\overline{subtalar \rightarrow toe}|}$$

The foot fitting consists of two phases, depicted in figure 5.7. The first phase updates the supporting foot of all generated steps $s_{1\cdots n}$. To update step i, the desired supporting foot vector $\vec{f_i}$ is calculated:

$$\vec{f_i} = \frac{F\Big(\text{final support of } s_{i-1}\Big) + F\Big(\text{initial support of } s_{i+1}\Big)}{2}$$

and subsequently normalized. The poses at the centre of the period of double support is used, analogous to the approach in section 3.3. If there is no previous or next step, the appropriate pose of s_i is used instead.

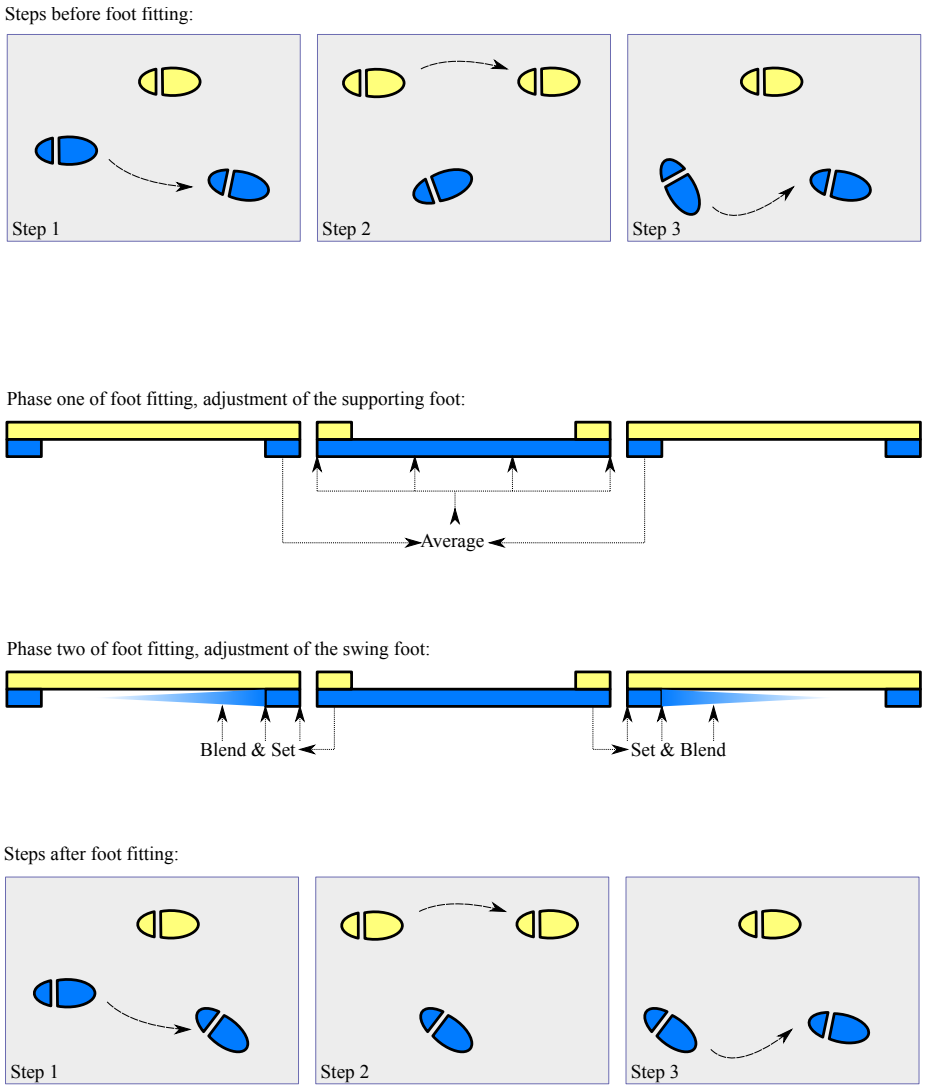


Figure 5.7: The foot fitting process. The three steps at the top have to be concatenated. The three steps at the bottom show the result of the foot fitting algorithm. For clarity only the foot fitting process of the right (blue) foot has been shown.

The second phase of the foot fitting algorithm is applied to the swing foot. The adjacent step (the next resp. previous step when adjusting the period of final resp. initial support) is inspected to obtain the desired foot vector. This foot vector is then applied to the support period, and blended into the swing to produce a smooth motion. We have found that a blend window of 40% to 80% of the swing duration produces a pleasant result.

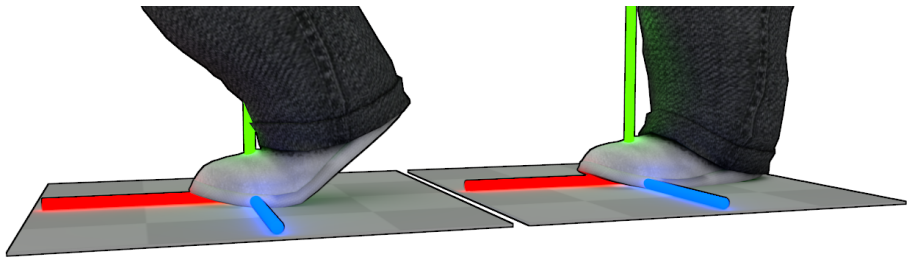


Figure 5.8: Two examples of the sole plane in different foot configurations.

Once $\vec{f_i}$ has been determined it is applied to the foot, identically for the supporting and the swing feet. We calculate the *sole plane* such that the subtalar and toe lie in the plane, and the normal is parallel to the $\overline{ankle, subtalar, toe}$ plane. See figure 5.8. The foot is rotated in the sole plane around the subtalar, such that the foot vector is aligned with the projection of $\vec{f_i}$ on the sole plane. The result is that the pitch of the foot is kept during the swing, maintaining as much of the natural motion as possible, while still producing proper alignment on the ground where the sole planes of the adjacent steps align.

5.7 Concatenation of steps

After the individual steps have been generated they are concatenated. Every step S_{n+1} is aligned to the previous step S_n, in such a way that the feet of S_{n+1} start where S_n ended. The final double stance of S_n is then blended with the initial double stance of S_{n+1} to produce a more or less smooth transition (see figure 3.3). This blending is performed in more or less the same way as the interpolation between the four source animations, including the positional interpolation. Because in both animations the feet are placed in the same posi-

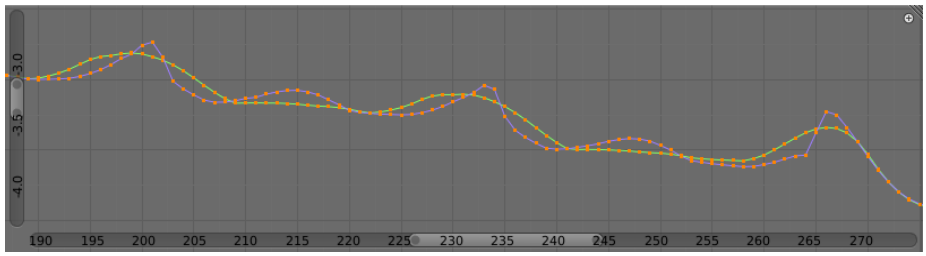


Figure 5.9: Root fitting. The purple and green graphs show the root height before resp. after the root fitting procedure.

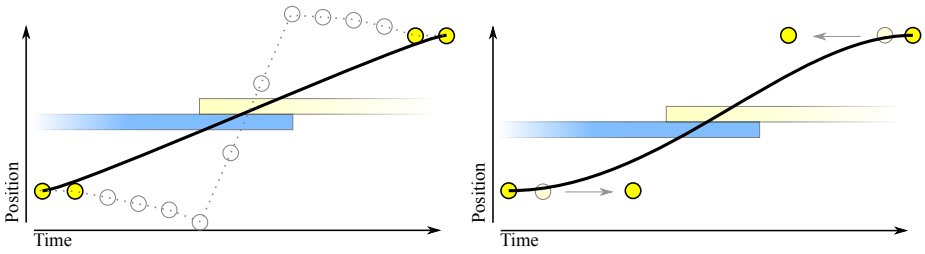


Figure 5.10: Two Bézier curves. The left curve shows the result when the keyframes are used as-is. The right curve shows our corrected curve.

tion, no foot skating will occur. The location constraints on the ankle, subtalar and toe are linearly interpolated. The ankle and toe are then repositioned such that the bones keep their orientation and retain their original length, and the subtalar remains in the same position,

The concatenation is performed during the potentially very short period of double support. A more common approach is to blend during the swing period of the steps before and after the current step. This approach would require that the cadence of the steps in the *input* animations are the same as in the generated walk, posing an undesirable high level of dependence between input and output animations. For example, when all the input animations are in a left-right-left cadence, it would be impossible to generate a left-right-right-left walk.

The positions of the feet of step s_i are guaranteed to align with the final positions of s_{i-1} and the initial positions of s_{i+1}. This guarantee only holds for joints in the feet. Because we generate steps independently we might introduce a discontinuity in the root trajectory between steps. To allow for smoother blending between steps the root joint is fitted using a cubic Bézier spline in a window centred around the overlapping frames – see figure 5.9. A Bézier curve is defined by four *control points*. The first and fourth control point define the start and end of the curve; the second and third point define the slope at the start and end of the curve. We use the root position on the two frames at the start and two frames at the end of the window as control points. These points are not directly usable, as the temporal difference between two consecutive frames is too small – only 1/30 second. The curve would be too linear to allow for smooth interpolation – see the left of figure 5.10. We move the middle two control points such that they are at 1/3rd and 2/3rd of the window while remaining the slope. The adjustment to the root position is also applied to the hips and the upper body. The orientation of the root joint is interpolated using SLERP in quaternion space, also in a window around the overlapping frames. Due to the way we represent the body (see chapter 6) these adjustments can be made independent of the feet, and thus do not result in any foot skating.

5.8 Upper body motions

We concatenate the generated steps during the periods of double support. These periods can be as short as a single keyframe; a blending window this small can cause jerky motion on the upper body. To solve this we filter the orientations of the upper body using an efficient coordinate and time-invariant filtering technique of Lee and Shin [LS02]. This technique determines the response of a smoothing filter in the logarithmic map. The response is then applied in the rotation space. We believe that eventually a more intricate upper body motion planner is needed, for dealing with more complicated tasks such as picking up objects. This basic filter does result in natural upper body motions which is useful when no additional tasks need to be performed by the character.

CHAPTER 6

Body representation

In this chapter we describe the body representation we use throughout the algorithm. The major advantage of this more linearized representation is that we can freely move the feet to any potentially reachable position without having to worry about bone stretching. After we describe the representation itself, we show the method by which we calculate the remaining joint angles so that the animation can be displayed by a traditional animation system.

6.1 A linearized lower body representation

The upper body is represented by a hierarchical system of joints of which the relative orientations are expressed by quaternions. For the upper body rotations, including hip and root, we blend all the rotations in a linear vector space as described in section 5.3. As we do not place any constraints on the hands or head this technique is sufficient for our goal.

Our lower body representation is based on the morphology-independent representation by Kulpa et al. The legs have been replaced by a "limb of variable length". This limb is represented by a half-plane $\mathcal{K}$ that contains the knee. The coordinate frame of $\mathcal{K}$ is determined by the $\overline{hip \rightarrow ankle}$ vector and the roll angle ρ. The latter is defined by the angle between the $\overline{hip \rightarrow root}$ vector and the $\overline{hip \rightarrow knee}$ vector projected onto the plane perpendicular to the $\overline{hip \rightarrow ankle}$ vector, as shown in figure 6.1. The roll angle is normalized to the interval $[-\pi, \pi)$ to ensure that linear interpolation of roll angles will result in a valid roll angle. When the leg is stretched, the $\overline{hip \rightarrow knee}$ vector lies too close to the $\overline{hip \rightarrow ankle}$ vector to be useful. In that case the $\overline{ankle \rightarrow subtalar}$ vector is used instead.

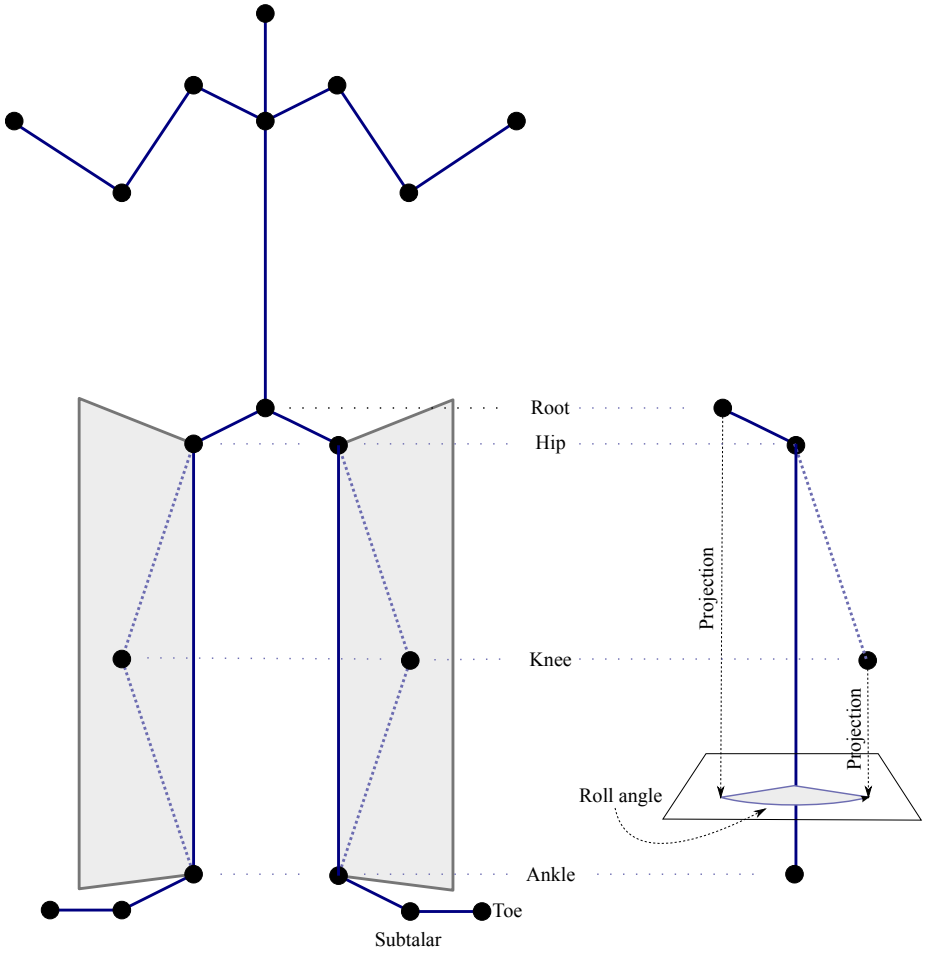


Figure 6.1: The morphology-independent representation of Kulpa et al. (left) and calculation of the roll angle (right)

6.2 Classical skeleton representation

After the entire walking animation has been generated, it has to be converted to a classical skeleton representation in order to be rendered by our framework.

The half-plane $\mathcal{K}$ is defined by three points: the hip and ankle locations, and the point obtained by rotating the $\overline{hip \rightarrow root}$ vector around $\overline{hip \rightarrow ankle}$ by ρ and adding that vector to the hip location. Just as described by Kulpa et al. the origin of $\mathcal{K}$ is at the hip, the z-axis points towards the ankle, the x-axis is perpendicular to $\mathcal{K}$ and the y-axis completes the coordinate frame. The coordinate system is right-handed, such that the knee will have a positive local y-coordinate. The rule of cosines is then used with the known bone lengths to determine the coordinates of the knee.

Before the knee location is calculated we need to ensure that the ankle positions are reachable from the current root position. To avoid knee popping[KSG02] we make sure this can be done without fully stretching the legs. We apply ground adaptation as described in Kulpa et al., and set the allowed length of the $\overline{hip \rightarrow ankle}$ vector to

$$(upper\ leg\ length + lower\ leg\ length) \times damping\ factor$$

We have found a damping factor of 0.995 to give good results. In case damping is not enough to avoid full leg stretch, we lower the root to the highest point at which the feet can be reached. This process is called *ground adaptation*. For both feet we calculate the height the hips can reach, taking the damping factor into account. The minimum of those heights and the current height of the root will become the new height of the root.

The pelvis width is subtracted from the ankle position, so that the adjusted ankle positions can be directly compared to the root. For adjusted ankle position A, root position R and damped leg length L we calculate height difference h:

$$(R_x - A_x)^2 + (R_y + h - A_y)^2 + (R_z - A_z)^2 = L^2$$

$$\Downarrow$$

$$h = A_y - R_y + \sqrt{L^2 - (R_x - A_x)^2 - (R_z - A_z)^2}$$

h is calculated for both feet and taken to be the minimum. It is always a non-positive real number, as the root will never be moved upward by this process.

After h has been determined on a frame-by-frame basis, we perform *ground adaptation filtering*. This adds a degree of temporal consistency to the pelvis height. Gaussian curves are fitted to h, as shown in figure 6.2. This method effectively works like a low-pass filter, with the added advantage that peak values are maintained, ensuring that the foot location constraints remain valid. By only filtering h and not the height of the pelvis, we maintain as much of the original pelvis movement as possible. After filtering of h it is applied by adding h to the root height.

The knee roll angle is filtered using a similar filter as implemented for the upper body motions. A typical result of this filtering is shown in figure 6.3.

After the locations of the lower body joints have been determined the joint orientations are calculated using a direct method that only requires a fixed number of operations. The following steps are performed:

1. The hip is rotated such that the knee is positioned correctly. This defines the hip pitch and yaw.
2. The knee angle k_p (the angle between $\overline{knee \rightarrow hip}$ and $\overline{knee \rightarrow ankle}$) is determined using the local coordinates in $\mathcal{K}$. This defines the knee pitch, its only DoF. The knee is rotated around its local x-axis over k_p.
3. The hip is rotated around the $\overline{hip \rightarrow knee}$ vector such that the ankle is positioned correctly. This defines the hip roll.
4. The ankle is rotated such that the subtalar is positioned correctly. This defines the ankle pitch and yaw.
5. The subtalar angle s_p is calculated from the coordinates of the ankle, subtalar and toe. This defines the subtalar pitch, its only DoF. The subtalar is rotated around its local x-axis over s_p.
6. The ankle is rotated around $\overline{ankle \rightarrow subtalar}$ such that the toe is positioned correctly. This defines the ankle roll.

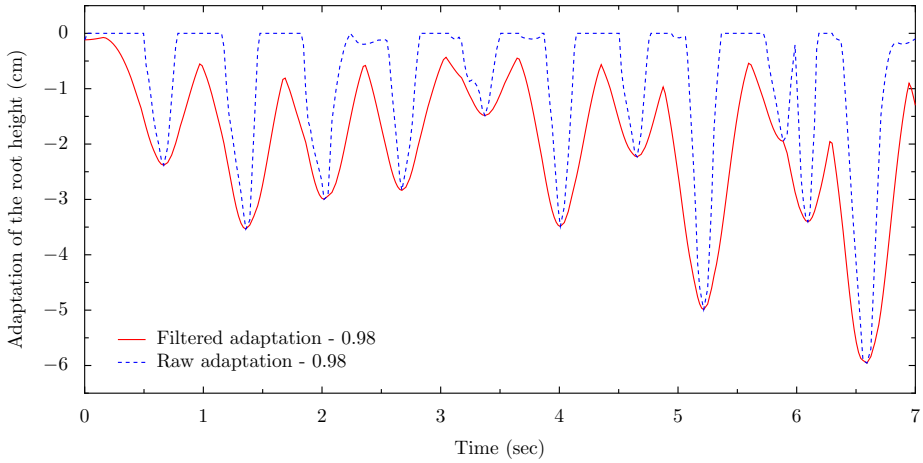


Figure 6.2: Filtering of the ground adaptation when using a 0.98 dampening factor.

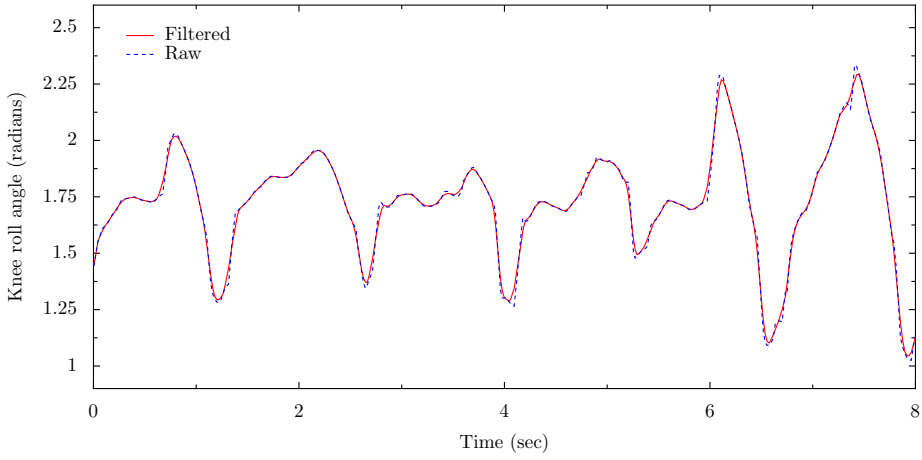


Figure 6.3: The knee roll of a generated walk before and after filtering.

CHAPTER 7

Results

In this section we present some of our results. Our database contained 184 steps (92 each for the left- and right-sided database). All recorded motions are standard walking motions, including walking backward, turning, side stepping, and transitions between those motions and forward walking. The motions were recorded by two subjects of different body physiology, and the joint orientations were mapped onto the same skeleton before insertion into the database. For the upper-body filtering, we use a low-pass filter with mask $\left[\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16}\right]$ as suggested by the authors [LS02]. All experiments were executed on an AMD Athlon 64 X2 3 GHz dual-core processor with 3 GiB RAM.

We primarily tested on three foot plans. In the first test case we guided our character through an in-door environment – see figure 7.1. The query foot plan consisted on 12 steps forming high-curvature locomotion. The duration of the resulting motion was 8.2 seconds. In the second test case we guided the character along a narrow ridge – see figure 7.2. This foot plan consisted of 20 steps with a resulting animation of 14 seconds. Note that the character automatically started sidestepping when required. The third test case consisted of 12 steps comprising a transition from forward-to-backward-to-forward walking – see figure 7.3. This is a difficult and somewhat unnatural foot plan, yet our technique produces reasonable results even for such an example. The duration of the final motion was 9.1 seconds. In the figures showing the examples (figures 7.1, 7.2 and 7.3) the purple rings indicate left query foot placements and the yellow rings indicate right query foot placements. The resulting animations can be seen in the accompanying videos.

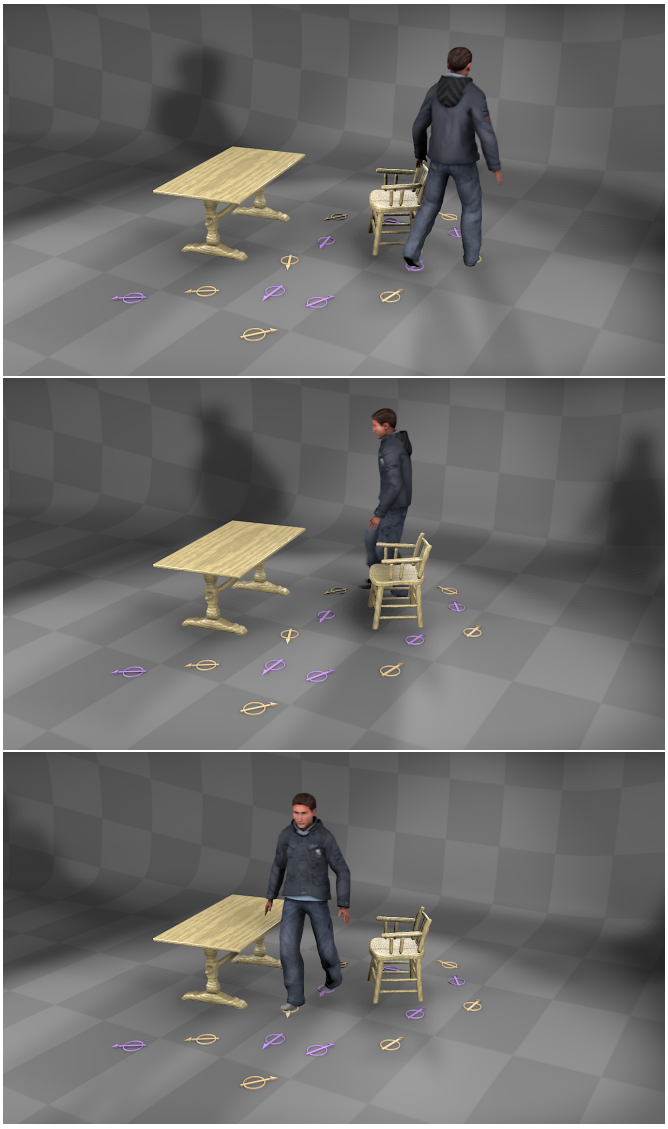


Figure 7.1: A walk in a highly constrained environment.

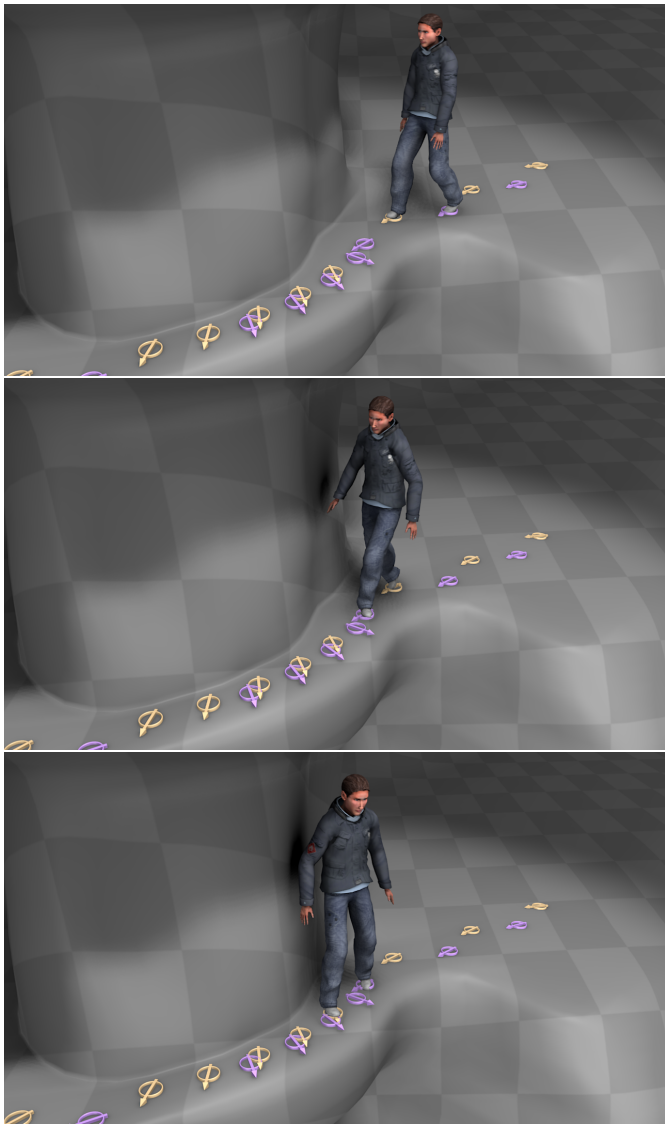


Figure 7.2: A transition from normal walking to sidestepping.

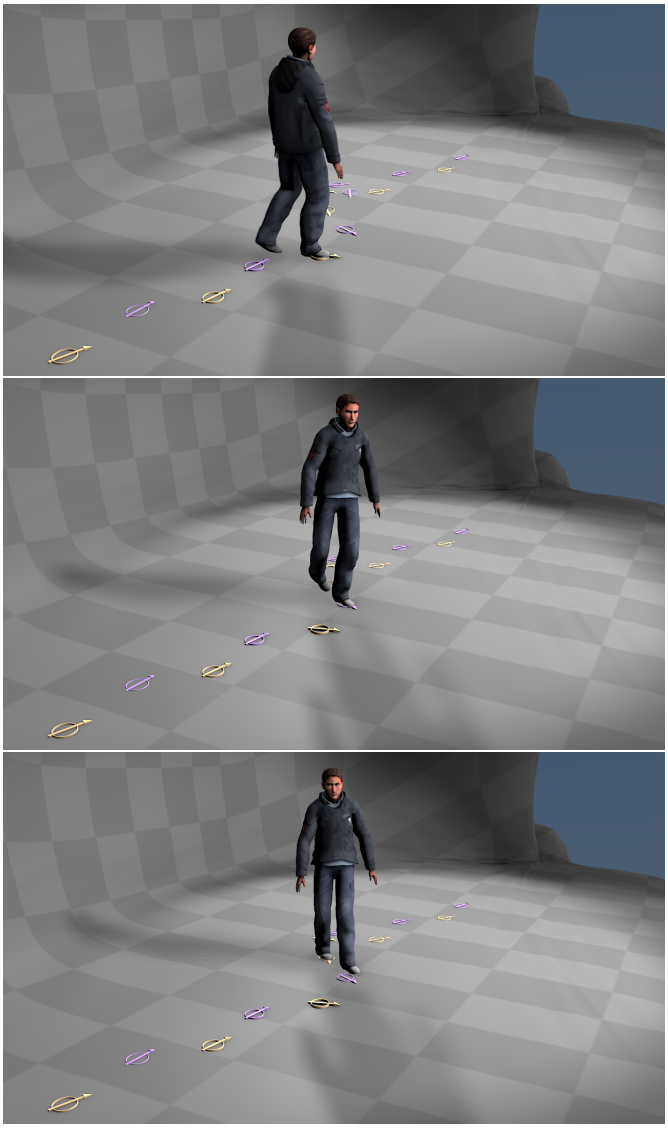


Figure 7.3: A transition from forward to backward walking.

7.1 Performance

On average, online generation of a single step (excluding rendering, including concatenation to previous step and filtering) takes 0.026 seconds ($\sigma = 0.01$ seconds, $N = 44$ steps). Conversion to the classical skeleton representation took 0.005 second per step ($\sigma = 0.001$ seconds per animation, $N = 44$ steps in three animations). The average duration of a generated step was 0.62 seconds. Generation costs only 4.2% of the animation time, making this technique suitable for real-time applications.

7.2 Upper body movement

The arm movement in the accompanying video is quite small. This is partially caused by the input animations which show the same small arm movement. Those animations were recorded in a relatively small space with loose carpet tiles. The motions are by definition natural, but may appear somewhat stiff when shown out of context.

A second cause is the interpolation technique, which acts as a low-pass filter. A possible solution could lie in splicing a different motion onto the upper body; for example the upper body motion of the spanning neighbour with the highest weight could be used.

7.3 Filtering

The filtering and fitting we apply to the skeleton have shown to be effective in smoothing out the blend artefacts. The accompanying video "raw-vs-filtered" provides for a visual comparison.

The filtering of the knee roll resulted in a subjectively "firmer" walk. The graph shown in figure 7.4 shows a real walking animation; this animation was different than the generated animation, but still serves to compare jerkiness between the filtered and real walks. Even though our filtered graph is smoother than that of a real walk, we perceived the filtered motion as more believable walk than the unfiltered motion.

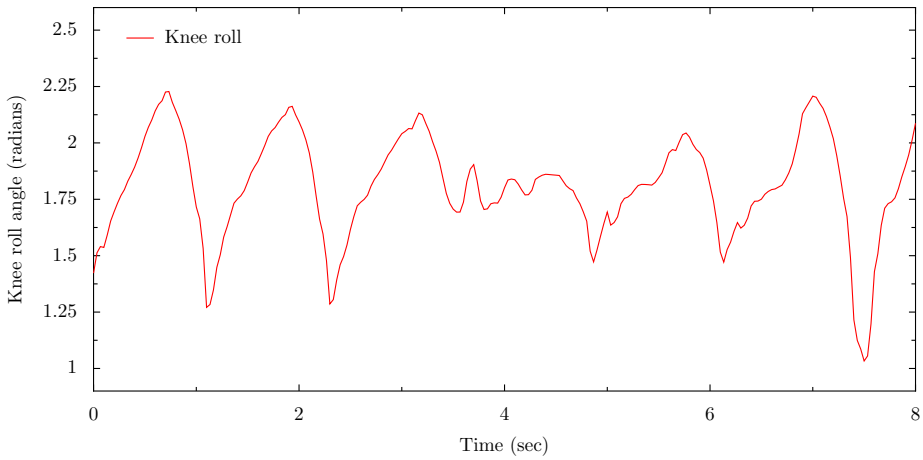


Figure 7.4: The knee roll of a real walking motion.

7.4 Blend candidate selection

We generated several walking animations using blend candidate selection. In total 88 steps were generated. On average 7.39 blend candidates were found for every spanning neighbour. However, these were not equally distributed; for 208 spanning neighbours no blend candidates were found. Of the remaining 144 spanning neighbours only 60 had more than five alternative blend candidates. Of the 144 spanning neighbours for which blend candidates were found, 36.1% were replaced by a better candidate; this makes for 14.7% of all the spanning neighbours.

The alternative blend candidates lie further from the query point in parameter space than the original spanning neighbours. When the blend candidates are replaced by alternatives, the alternatives receive a significantly lower weight to ensure exact foot placement, in certain cases up to a factor 10. Even though the "better" – according to our high-dimensional distance function – candidates get a lower weight, the resulting animations do appear more natural. In some cases it prevents forward and backward steps from being blended – see figure 7.5.

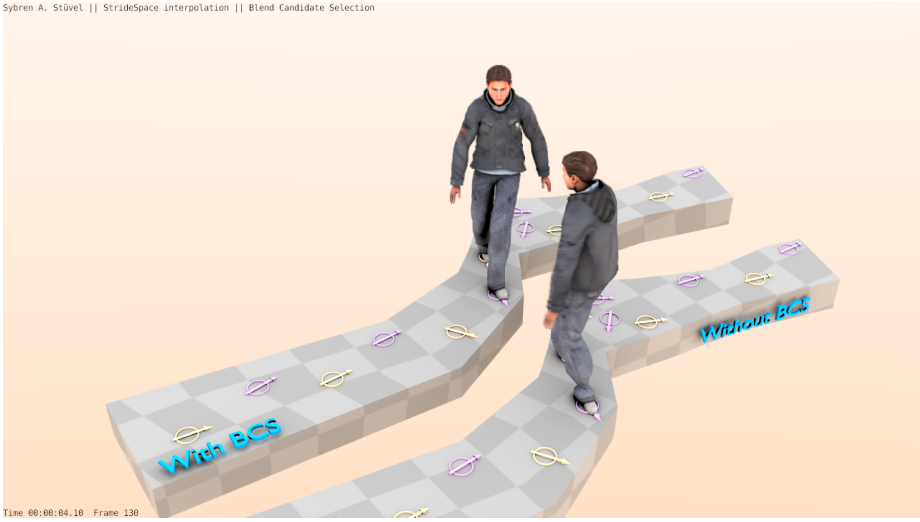


Figure 7.5: The result of blend candidate selection (BCS). The bottom character has BCS disabled, resulting in backward and forward steps blended together.

CHAPTER 8

Conclusion and future work

We have presented an efficient technique that allows for exact parameterization of foot-print driven synthesis. Given a query foot plan, our technique is able to generate an animation that places the feet exactly at the desired positions. It also takes additional soft constraints such as timing into account due to a novel blend candidate selection strategy. The algorithm does not depend on manual annotation or adjustments of the input motion clips.

Due to the linearized representation of the lower body no complex inverse kinematics nor extensive motion modification is needed. This results in a fully automatic generation of highly constrained animation in real-time. Not only does the generation of the step take less time than the duration of the resulting animation, it leaves enough CPU time for other application domains, such as path planning and rendering.

The steps generated by our technique are quite similar to each other, despite the selection of possibly quite different blend candidates. This is a side-effect of the blending process, where the average of four input clips is calculated. This averaging acts like a low-pass filter over the input clips, making the output more uniform and thus more suitable for concatenation.

8.1 Terrain height

There are many possibilities for future research. Our technique is suitable for walking on flat terrain. It could be easily combined with the adaptation technique by Kulpa et al.[KMA03] to allow for limited height differences or moving terrain, by planning the footsteps on a plane then adjusting the resulting ani-

mation to the height of the terrain. Another approach would be to expand S_{low} with more parameters to account for the height differences. However, this would increase the number of required input motions and blend candidates as well as impact the query time on the database.

Our use of a linearized representation of the lower body was very useful. It will be interesting to investigate even simpler representations to reduce and simplify the search space. A simpler search space may also compensate for the added complexity of a non-level terrain.

8.2 Blend candidate selection and database size

At this point the set of blend candidates is underestimated. It might be interesting to preprocess the space such that all possible blend candidates can be efficiently retrieved. This might improve the resulting animation. Also, instead of evaluating each blend candidate individually with respect to the query step, it might be interesting to evaluate the result of blending the set of candidates instead.

A denser database would increase the effectiveness of the blend candidate selection. Not only would there be more candidates; a denser database could make the smallest angle of the tetrahedra larger, thus enlarging the frustums in which blend candidates are found. It would be interesting to subsample the database specifically in those points that maximize the smallest angle. Subsampling has been used by Kovar et al.[KG04] to increase accuracy. Having blend candidates for all spanning neighbours would also reduce the observed "weight loss", so that a better matching animation will have a higher weight.

As can be seen in figure 4.3 it could be that there are more steps in the database than strictly needed. It may be interesting to see the resulting animations after the database has been pruned. This pruning could be performed automatically, guided by the more sophisticated distance metrics mentioned above. One of the problems with a Delaunay tetrahedralization is that they face difficulty in handling large data [DGH01]. Pruning the database could lessen the impact. Even though our algorithm was designed to allow the generation of a wide variety of animations based on a relatively small corpus, a possible solution for the case where a very large database is required could lie in an adaptation of the *Delaunay based shape reconstruction* algorithm by Dey et al.[DGH01]

8.3 Extrapolation outside convex hull

When the query falls outside the convex hull of the database the nearest neighbour is used without blending (see section 5.2). In this case the algorithm can no longer place the feet at the query position, as extrapolation would be required to do so. However, we have observed that limited extrapolation using our technique can result in natural motions. This can be realized by using weights outside the $[0, 1]$ interval. However, it is unclear to what extend extrapolation is possible while maintaining realistic results. The unnaturalness mostly stemmed from implausible to sheer impossible rotations of the feet. A possible solution would be to apply a different interpolation scheme to the orientation of the feet, or to force the feet into the query orientation. Further research is needed to investigate these or other extrapolation techniques.

8.4 Naturalness

The naturalness of the resulting motion is partially determined by the naturalness of the query foot plan. Two out of the three examples in our tests were manually set (figures 7.3 and 7.2). The example in figure 7.1 was based on the foot plan extracted from a motion capture clip, so in a way it could be called "pedestrianally" set. We are currently working on automatic footstep planners that can generate a foot plan without depending on a pre-existing corpus of motion capture data. Next to that, at this point our assumption was that selecting two consecutive steps where the overlapping foot placements resemble each other allow for a good transition. In that sense, selecting blend candidates in S_{high} functions as a posture distance metric. This technique was primarily developed for lower-body motion, for which this assumption seems to hold. To ensure good upper-body transitions, additional information might need to be taken into account, and more sophisticated distance metrics might need to be used. As an alternative the upper body motions could be planned independently of the lower body, and then grafted onto each other. This opens the door for better planning of the upper body, for example producing manipulating or grabbing motions. Future research will have to determine whether this is feasible.

Bibliography

[AF09] Brian F. Allen and P. Faloutsos. Evolved controllers for simulated locomotion. In *MIG 2009: Proceedings of the 2nd international workshop Motion in Games*, pages 219–230. Springer, 2009.

[BE09] B. J. H. van Basten and A. Egges. Evaluating distance metrics for animation blending. In *FDG '09: Proceedings of the 4th International Conference on Foundations of Digital Games*, pages 199–206, New York, NY, USA, 2009. ACM.

[BPE10] B.J.H. van Basten, P.W.A.M. Peeters, and A. Egges. The StepSpace: Example-based footprint-driven motion synthesis. *Computer Animation and Virtual Worlds*, 2010. Special issue of selected papers from the 23rd Annual Conference on Computer Animation and Social Agents.

[BTT90] Ronan Boulic, Nadia Magnenat Thalmann, and Daniel Thalmann. A global human walking model with real-time kinematic personification. *The Visual Computer*, 6(6):344–358, November 1990.

[CLS03] Min Gyu Choi, Jehee Lee, and Sung Yong Shin. Planning biped locomotion using motion capture data and probabilistic roadmaps. *ACM Transactions on Graphics*, 22(2):182–203, April 2003.

[DGH01] Tamal K. Dey, Joachim Giesen, and James Hudson. Delaunay based shape reconstruction from large data. In *PVG '01: Proceedings of the IEEE 2001 symposium on parallel and large-data visualization and graphics*, pages 19–27, Piscataway, NJ, USA, 2001. IEEE Press.

[EB79] B. C. Elliott and B. A. Blanksby. Optimal stride length considerations for male and female recreational runners. *British Journal of Sports Medicine*, 13(1):15–18, 1979.

[GR96] Shang Guo and James Robergé. A high-level control mechanism for human locomotion based on parametric frame space interpolation.

In *Proceedings of the Eurographics workshop on Computer animation and simulation '96*, pages 95–107, New York, NY, USA, 1996. Springer-Verlag New York, Inc.

[Gra98] F. S. Grassia. Practical parameterization of rotations using the exponential map. *The Journal of Graphics Tools*, 3.3, 1998.

[HG07] Rachel Heck and Michael Gleicher. Parametric motion graphs. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 129–136, New York, NY, USA, 2007. ACM.

[Hof65] K. Hoffman. The relationship between the length and frequency of stride, stature and leg length. *Sport (Belgian)*, 8(3), 1965.

[HWBO95] Jessica K. Hodgins, Wayne L. Wooten, David C. Brogan, and James F. O'Brien. Animating human athletics. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 71–78, New York, NY, USA, 1995. ACM.

[KG03] Lucas Kovar and Michael Gleicher. Flexible automatic motion blending with registration curves. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 214–224, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[KG04] Lucas Kovar and Michael Gleicher. Automated extraction and parameterization of motions in large data sets. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 559–568, New York, NY, USA, 2004. ACM.

[KGP02] L. Kovar, M. Gleicher, and F. Pighin. Motion graphs. *Proceedings of ACM SIGGRAPH 2002*, 21(3):473–482, July 2002.

[KMA03] R. Kulpa, F. Multon, and B. Arnaldi. Morphology-independent representation of motions for interactive human-like animation. *Eurographics 2005*, 24(3), 2003.

[KSG02] L. Kovar, J. Schreiner, and M. Gleicher. Footskate cleanup for motion capture editing. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 97–104, New York, NY, USA, 2002. ACM.

[LS02] J. Lee and S. Y. Shin. General construction of time-domain filters for orientation data. *IEEE Transactions on Visualization and Computer Graphics*, 8(2):119–128, 2002.

[MBBT00] Jean-Sébastien Monzani, Paolo Baerlocher, Ronan Boulic, and Daniel Thalmann. Using an intermediate skeleton and inverse kinematics for motion retargeting. *Computer Graphics Forum*, 19(3):11–19, 2000.

[Pee09] P.W.A.M. Peeters. Experimentation project: Footskate. Technical report, Universiteit Utrecht, November 2009.

[PSS02] S. I. Park, H. J. Shin, and S. Y. Shin. On-line locomotion generation based on motion blending. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 105–111, New York, NY, USA, 2002. ACM.

[PW99] Zoran Popović and Andrew Witkin. Physically based motion transformation. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 11–20, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[RCB98] Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications*, 18(5):32–40, 1998.

[Stü09] S. A. Stüvel. Experimentation project - stepspace interpolation. Technical report, Universiteit Utrecht, November 2009.

[Tri04] Bijan Kumar Tripathy. Study on step distance and its relation with some morphometric features in adult male. *Anthropologist*, 6(2):137–139, 2004.

[UAT95] Munetoshi Unuma, Ken Anjyo, and Ryozo Takeuchi. Fourier principles for emotion-based human figure animation. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 91–96, New York, NY, USA, 1995. ACM.

[vdP97] Michiel van de Panne. From Footprints to Animation. *Computer Graphics Forum*, 16(4):211–223, October 1997.

[Wik09] Wikipedia community. Generalized quaternion interpolation. 1 November 2009.

APPENDIX A

The algorithm in pseudocode

A.1 The Database class

```
### the Database class, pseudocode

def add_footstep(step):
    '''Adds a single footstep to the database. The input step is a
    FootStep object obtained from the footstep detector. It already
    contains information about the swing duration, stance durations,
    etc.'''

    # Convert to a canonical step by rotating and translating to the
    # supporting foot coordinate frame, and piecewise linear time
    # warping to the normalized durations.
    canonical_step ← new CanonicalStep(step)

    perform cleanup: move the feet onto the ground during the stance.

    # Insert the step into the proper Delaunay tetrahedralization.
    db ← _delaunay_db(step.parameters)
    vertex ← db.insert(step.parameters)
    vertex.step ← step

def _delaunay_db(parameters):
    '''Returns the Delaunay database for the given parameters.'''

    if parameters.x < 0: return left database
    return right database

def add_animation(animation):
    '''Adds all the footsteps in the animation to the database.'''

    # Run the footstep detector.
    footplan ← create_footplan_from_animation(animation)
```

```
        # Add the individual steps.
        for step in footplan:
            add_footstep(step)

    def weigh(parameters):
        '''Returns the spanning neighbours of the query parameters.

        Each spanning neighbour contains a canonical step and a weight.
        '''

        # Query the correct Delaunay tetrahedralization.
        db ← _delaunay_db(parameters)
        cell ← db.locate(parameters)

        if cell.is_infinite():
            # The parameters were outside the convex hull.
            spanning_neighbour ← db.nearest_point(parameters)
            spanning_neighbour.weight ← 1.0

            # Return a list of only one neighbour, with weight ← 1.0
            return [spanning_neighbour]

        # Determine the weights and return the spanning neighbours.
        neighbours ← find_blend_candidates(query, cell.vertices)
        weights ← determine_weights(neighbours, parameters)

        for index, neighbour in enumerate(neighbours):
            neighbour.weight ← weights[index]

        return neighbours

    def determine_weights(neighbours, q):
        '''Returns the weights such that the neighbours are interpolated
        onto the query parameters 'q'.'''

        (a, b, c, d) ← neighbours

        e ← intersection_point(Plane(a, b, c), Line(d, q))
        f ← intersection_point(Line(a, b), Line(c, e))

        # (wX, wY) ← weigh(X, Y, Z); such that wX * X + wY * Y = Z
        (w_a_abf, w_b_abf) ← weigh(a, b, f)
        (w_c_cfe, w_f_cfe) ← weigh(c, f, e)
        (w_d_deq, w_e_deq) ← weigh(d, e, q)

        # Return the weights
        return [w_e_deq * w_f_cfe * w_a_abf,
                w_e_deq * w_f_cfe * w_b_abf,
```

```
            w_e_deq * w_c_cfe,
            w_d_deq]

def find_blend_candidates(query, neighbours):
    # Define four planes by each combination of three neighbours

    # Find all blend candidates
    for step in database:
        for neighbour in neighbours:
            if step in frustum for neighbour:
                remember step as candidate for this neighbour

    # Select the best four candidates
    for neighbour in neighbours:
        best_distance ← Dhigh(query, neighbour)

        for step in candidates for this neighbour:
            dist ← Dhigh(query, step)
            if dist < best_distance:
                best_distance ← dist
                best_candidate ← step

    return [best candidate for each neighbour]
```

A.2 The Generator class

```
### the Generator class, pseudocode

def generate_chain(FootPlants footplants):
  '''Generates a chain of concatenated steps. Returns the animation.'''

  separate_steps ← create_separate_steps(footplants)
  perform_foot_fitting(separate_steps)
  anim ← concatenate_steps(separate_steps)

  # Convert the animation to the quaternion-based skeletal
  # representation.
  return anim.convert_to_skeleton_anim()

def create_separate_steps(FootPlants footplants):
  '''Returns a vector<CanonicalStep> of separate steps.'''

  # We need at least three foot plants, or there is no walk.
  assert footplants.size > 3

  # This mapping contains the last position of each foot.
  last_position ← {
    footplants[0].side: footplants[0].position
    footplants[1].side: footplants[1].position
  }

  steps ← []

  # The first two foot plants define the initial positions of the feet.
  assert footplants[0].side ≠ footplants[1].side

  # Iterate over the other foot plants to generate steps.
  for plant in footplants[2:]:
    # Determine the 'swing' and 'supporting' sides.
    swing_side ← plant.side
    supporting_side ← plant.side.opposite()

    # Generate the animation based on the previous foot
    # positions, and the next foot position.
    step ← generate_positioned_step(
        last_position[supporting_side], last_position[swing_side],
        plant.position, swing_side)
    steps.append(step)

    # Remember the final positions of the feet.
    last_position[supporting_side] ← step.supporting_position
    last_position[swing_side] ← step.swing_to_position
```

```
  return steps

def generate_positioned_step(supporting_position,
    swing_from_position, swing_to_position, swing_side):
  '''Generates a single step placed on the given positions.'''

  # Determine the query parameters, as well as the rotation that brings
  # the query positions into the supporting foot coordinate frame.
  (rotation, parameters) ← determine_parameters(supporting_position,
      swing_from_position, swing_to_position, swing_side)

  # Determine the spanning neighbours of the query. This includes
  # the interpolation weight for each neighbour.
  spanning_neighbours ← database.weigh(parameters)

  # Let the blender class generate an animation based on the
  # interpolation weights and animations from the database.
  canonical_step ← blender.generate(spanning_neighbours)

  # Rotate and translate the step animation into the correct orientation
  # and position.
  canonical_step.anim.rotate(rotation.conjugate())
  canonical_step.anim.translate(supporting_position)

  return step

def perform_foot_fitting(vector<CanonicalStep> separate_steps):
  '''Performs foot fitting, updating the steps.

  This method uses the foot fitter, described in the appendix.
  '''

  # Phase 1, update the supporting foot
  for step_nr, step in enumerate(separate_steps):
      # See section 5.6: Foot fitting
    target_vector ← find_support_average(step_nr, separate_steps)

    # Update each frame in the animation with the new
    # orientation for the supporting foot.
    for frame in step.anim:
      foot_fitter.update_foot(frame, target_vector, 1.0,
              step.supporting_leg_joints)

  # Phase 2, update the swing foot.
  for step in separate_steps:

    # Update the initial support based on the previous step.
    # Skipped if this is the first step.
```

```
96      target_vector ← prev_step.foot_vector(step.swing_side,
                mid_of_final_support)
        foot_fitter.fit_swing_from_initial_stance(step, target_vector)

        # Update the final support based on the next step.
        # Skipped if this is the final step.
        target_vector ← next_step.foot_vector(step.swing_side,
                mid_of_initial_support)
        foot_fitter.fit_swing_to_final_stance(step, target_vector)

106 def concatenate_steps(vector<CanonicalStep> separate_steps):
      '''Concatenates the steps, returns the new animation.'''

      walk_anim ← new empty animation

      for step in separate_steps:
        # Determine double-stance durations (dsd). Assumes prev_step and
        # next_step exist, otherwise just uses the timings from the step
        # itself.
        dsd_start ← (step.dsd_start + prev_step.dsd_end) / 2.0
116     dsd_end   ← (step.dsd_end + next_step.dsd_start) / 2.0

        step_anim ← step.timewarp(dsd_start, dsd_end)

        # Let the blender concatenate the step animation to the walk
        # animation we have so far, using an overlap of 'dsd_start' seconds.
        duration_before_concatenation ← walk_anim.duration()
        blender.concatenate(walk_anim, step_anim, dsd_start)

        # Fit the root location using Bezier spline, and root orientation
126     # using SLERP
        fit_root(walk_anim, duration_before_concatenation,
                 options.root_fit_window_size)

      return walk_anim
```

A.3 The FootFitter class

```
### Foot fitter class, pseudocode

def update_foot(frame, target_vctr, weight, leg_joints):
    '''Updates the foot orientation by rotating it in the sole plane.'''

    subtlr_pos ← frame.get_location_constraint(leg_joints.subtalar)
    ankle_pos  ← frame.get_location_constraint(leg_joints.ankle)
    toe_pos    ← frame.get_location_constraint(leg_joints.foot)
    current_vctr ← toe_pos - subtlr_pos

    # First get the plane through the joints, then calculate the sole
    # plane.
    foot_plane ← Plane(ankle_pos, subtlr_pos, toe_pos)
    sole_plane ← Plane(subtlr_pos, toe_pos, subtlr_pos +
                                            foot_plane.normal())

    # Project the current and target vectors onto the sole plane
    normal ← sole_plane.normal()
    current_on_sole ← current_vctr - normal.dot(current_vctr) * normal
    target_on_sole  ← target_vctr  - normal.dot(target_vctr ) * normal

    # Calculate the roll angle, and normalize to [-pi, pi)
    roll_angle ← angle_between(current_on_sole, target_on_sole)
    while roll_angle < pi: roll_angle +← 2 * pi
    while roll_angle >← pi: roll_angle -← 2 * pi

    rotation ← Quaternion(roll_angle * weight, normal)

    # Rotate around the subtalar
    new_ankle_pos ← rotation.rotate(ankle_pos - subtlr_pos) + subtlr_pos
    new_toe_pos   ← rotation.rotate(toe_pos   - subtlr_pos) + subtlr_pos

    # Set the new constraints.
    frame.set_location_constraint(leg_joints.ankle, new_ankle_pos)
    frame.set_location_constraint(leg_joints.foot,  new_toe_pos)

    # The rotation to the leg as well
    leg ← left or right leg of the frame, depending on leg_joints
    leg.roll +← roll_angle

def fit_swing_to_final_stance(step, target_vector):
    '''Fits the final stance to the target vector.'''

    # Calculate the three relevant timekeys. The NORMAL_XXX constants
    # refer to the predefined timekeys of normalized animations.
    start_swing_blend ← NORMAL_SWING_END
```

```
                        - NORMAL_SWING_DURATION * options.foot_fit_width
    start_dblstance ← NORMAL_SWING_END
    end_dblstance ← NORMAL_DURATION

    # Rotate the foot in the double stance to the target vector
    for timekey ∈ [start_dblstance, end_dblstance]:
        frame ← step.anim.get_keyframe(timekey)
        update_foot(frame, target_vector, 1.0, step.swing_leg_joints)

    # Blend during the swing
    blend_duration ← start_dblstance - start_swing_blend
    for timekey ∈ [start_swing_blend, start_dblstance):
        frame ← step.anim.get_keyframe(timekey)
        weight ← (timekey - start_swing_blend) / blend_duration
        update_foot(frame, target_vector, weight, step.swing_leg_joints)

# fit_swing_from_initial_stance is similar to fit_swing_to_final_stance
# and has not been included here.
```

A.4 The Blender class

```
### the Blender class, pseudocode

# We use 30 frames per second.
FRAME_RATE ← 30

def generate(vector<WeightedStep> spanning_neighbours):
    '''Generates a canonical step by interpolating the spanning neighbours.'''

    # Get a list of weights and a list of input steps.
    weights ← [neighbour.weight for neighbour in spanning_neighbours]
    steps   ← [neighbour.step   for neighbour in spanning_neighbours]

    output_step ← empty step object

    # Iterate over all frames. We know how many those are, as this is
    # done on the normalized steps. Allow the timekey to be slightly
    # larger than the actual duration, for numerical stability.
    frame_duration ← 1/FRAME_RATE
    for(timekey←0 ; timekey < NORMAL_DURATION + 0.1 * frame_duration ;
            timekey +← frame_duration):

        # Get a list of frames to blend, one frame of each input step.
        to_blend ← [step.get_keyframe(timekey) for step in steps]

        # Interpolate the frames and store the result in the output step.
        blended ← interpolate(to_blend, weights)
        output_step.set_keyframe(timekey, blended)

    return output_step

def concatenate(anim_a, anim_b, overlap_duration):
    '''Concatenates anim_b onto anim_a using 'overlap_duration' seconds
    of overlap.'''

    start_of_blend_timekey ← anim_a.last_timekey - overlap_duration

    overlap ← empty animation

    # Iterate over all frames in the overlap. Allow the timekey to be
    # slightly larger than the actual duration, for numerical stability.
    frame_duration ← 1/FRAME_RATE
    for(timekey←0 ; timekey < NORMAL_DURATION + 0.1 * frame_duration ;
            timekey +← frame_duration):

        # Build a list of weights and frames to perform interpolation.
        weights ← [1 - timekey / overlap_duration,
```

```
                        timekey / overlap_duration]

        frames ← [anim_a.get_keyframe(timekey + start_of_blend_timekey),
                  anim_b.get_keyframe(timekey)]

        # Interpolate the frames and store the result in the overlap
        # animation.
        blended ← interpolate(to_blend, weights)
        overlap.set_keyframe(timekey, blended)

    # Append the pieces
    b_after_overlap ← anim_b.crop(overlap_duration, anim_b.last_timekey)
    anim_a.remove_keyframes_after(start_of_blend_timekey)
    anim_a.append(overlap)
    anim_a.append(b_after_overlap)


def interpolate(frames, weights):
    '''Returns the interpolated keyframe.'''

    result ← empty keyframe
    result.root_translation ←
                    average(frame.root_translation for frame in frames)

    # The implementations of blend_rotations and
    # blend_location_constraints have not been included, as they
    # implement trivial weighted interpolations of 3D vectors.
    blend_rotations(result, frames, weights)
    blend_location_constraints(result, frames, weights)

    blend_legs(result, frames, weights)

    return result

def blend_legs(result, frames, weights):
    '''Blends the legs, i.e. the foot positions.'''

    average ← function that takes the weighted average over the input frames

    avg_l_subtalar ← average(left subtalar location)
    avg_r_subtalar ← average(right subtalar location)

    update_foot(result, left leg joints, avg_l_subtalar)
    update_foot(result, right leg joints, avg_r_subtalar)

    result.left_leg_roll ← average(left leg roll)
    result.right_leg_roll ← average(right leg roll)

def update_foot(frame, joints, subtalar_position):
```

```
    '''Moves the foot such that the subtalar is positioned correctly.'''

    if the frame already has constraints for ankle, subtalar and toe:
        return

    # Get the position based on the result of rotational interpolation.
    current_position ← frame.get_location(joints.subtalar)

    offset ← subtalar_position − current_position
105
    # Apply the offset to the entire foot.
    frame.set_location_constraint(joints.subtalar, subtalar_position)
    frame.set_location_constraint(joints.ankle,
                                  frame.get_location(ankle) + offset)
    frame.set_location_constraint(joints.toe,
                                  frame.get_location(toe)   + offset)
```