UNIVERSITA' DEGLI STUDI DI MILANO
Facoltà di Scienze Matematiche, Fisiche e Naturali

Near Optimal Synthesis of Digital Animation from Multiple Motion Capture Clips

Corso di Laurea Magistrale in:

Tecnologie dell'Informazione e della Comunicazione

Marco Alamia

Matricola n. 736779

Relatore: Prof. Ing. Alberto N. BORGHESE

Correlatore: Dott. Iuri FROSIO

Anno Accademico 2008/2009

ABSTRACT

The present work discusses theory and practice of a powerful animation method designed to generate walk animations for digital characters. Our system permits to interact with the animation, allowing the user to change at run-time several animation's parameters, such as the motion direction or the gait style, influencing the final animation. Our work starts presenting the skeleton animation system and the motion capture system; then we explain how we can, thanks to these two techniques, generate a database of walk animation clips. The so obtained animations are provided to the animation system which links them together in a sequence. Linking is obtained generating a smooth transition from one clip to the next one through the use of the animation blending technique. Since we want to interact with the animation at run-time, the clip's sequence is not given a priori, instead it is generated in real-time in respect to the user's desires. To this aim we create a controller in charge of choosing the next clip in respect to the task that it is given, which can be simulating a walk along a line, or simulating a walk where motion direction and character's orientation are required by the user. The controller leans on a selection policy in order to choose the next clip; in our work we propose two possible policies, a greedy one and a near-optimal one. The former goes through every animation contained in the database and evaluates the direct cost of adding the clip in exam to the sequence. The latter policy, instead, chooses the next clip evaluating an approximation of the optimal choice, which is obtained through the implementation of a reinforcement learning algorithm. The optimal choice estimates both the direct cost of choosing a clip as well as all the future costs that the system will pay for that choice. Unfortunately we can't effectively implement the optimal policy, therefore we content ourselves with an approximation that leads to the near-optimal policy. We lastly show how both these policies produce controllers capable of responding, in real-time, to the change of several animation parameters due to the user's interaction as well as the environmental constraints.

ACKNOWLEDGMENTS

I want to thank my thesis supervisor associate professor Alberto N. Borghese for his contribution during the development of this thesis. I am grateful to him for passing on to me appreciation for accurate work and method. This thesis is the fruit of a close collaboration and would certainly not have been the same without his contribution. In addition I thank my co-supervisor dott. Iuri Frosio for his help with the motion capture system.

I also wish to thank my family, who always supported me and encouraged me. I am grateful to you because you have always supported me with no hesitation. Thank you.

A mio padre per aver posto le basi di tutto ciò che so

A mia madre per avermi sempre cresciuto con infinito amore, nel bene e nel male ed in ogni momento della mia vita

A mio fratello, inestimabile compagno di viaggio verso Itaca

A Flora per aver saputo recuperare

CONTENTS

Part I

Introduction

1: Introduction

1.1 Introduction

Despite years of research on the topic, generating *interactive* realistic *character animation* remains one of the greatest challenges in digital animation. Capturing the smoothness and the nuance of a person that reacts to external stimulations requires tracking a high amount of variables that often change in unpredictable ways.

Our goal is to tackle a sub-problem of interactive character animation that is achieving *walk animations with interactive control*. We will approach this problem through the synthesis of a kinematic controller that blends some clips, read from an animation database, to achieve one (or eventually even more than one) specific task in real-time. For our technique we have defined, in the following order, an *animation framework*, a *blending engine* and a *control system*. This thesis discusses how we implemented the three mentioned systems and what results we achieved.

1.2 Animation framework

To animate our character we need a motion model for walk animations. We choose to use the *skeletal animation* system for representing movements since this is currently the most qualified way to animate a human character. In the first section of chapter *2:Animation Framework* we present what skeletal animation is and how it is implemented. This animation system has several well known drawbacks related to linear interpolation of orthonormal matrices. A well known solution to this problem is to use one quaternion and one vector to store transformations instead of one matrix or to use a dual-quaternion instead of the pair quaternion-vector (Kavan, et al. 2008). Because of time-limit problems the matrix approach was used since its code was already present inside the engine. The skeleton animation movement can be achieved in several ways so we had to decide which one could work better for us.

After having evaluated several animation methods we opted for *motion capture* techniques to develop the database of motions. In the second part of chapter one we analyze what motion capture is and which issues arise in using this technique, for example how we do deal with noise and missing data. Some problems are related to the fact that raw data, as it comes from motion capture, is acquired in an uncomfortable format; we analyze why this happens and how a conversion to a better form can be done.

Once we have acquired in a proper format the animation data, and we have mapped it onto our skeleton, we cut the so obtained animations into smaller clips ready to be mixed together into a single longer animation. To perform this mix we will use *animation blending*.

1.3 Blending system

1.3.1 Animation blending

In the past, long character's motions were obtained creating a single linear stream of animations where the entire motion was planned in advance and computed off-line. This approach did not fit well with interactive 3D character animation because of the uncertainty that interaction presents. One solution for this problem is to generate a set of high quality motions and then create transitions between these motions so they can be strung together into animations of unlimited length and great variety (Charles, et al. 1996). Transitions can be achieved in several ways, but the most widely used is animation blending.

To mix acquired data into a single long walk animation we blend a new skeleton animation and the current one every time the latter is going to end. In the first part of chapter *3:Blending System* we discuss what animation blending is and how we can use it to achieve our goal.

1.3.2 Linear interpolation

When blending two animations together we use automatic *interpolation* techniques based on mathematical frameworks. There are several ways to perform interpolation; the simplest, and probably the most widely used, is *linear interpolation* that in computer's jargon is often abbreviated as *lerp* (Raymond 2003). Linear interpolation is a method of curve fitting using linear polynomials. The simplest formula used to lerp two values is the following:

$$x = x_1(1 - t) + x_2 t$$

Equation 1.3-1

where x, x_1 and x_2 can be scalar values as well as matrices or vectors, while t is a scalar in $\mathbb{R}$ and must respect $0 \leq t \leq 1$. If one wants to blend a first skeleton's pose into another skeleton's pose he can use Equation 1.3-1 on every bone's matrix. This means that we can compute the in-between poses using the following equation:

$$\mathbf{B} = \mathbf{B_1}\,(1 - t) + \mathbf{B_2} t \quad \forall\, \mathbf{B_1} \in p, \mathbf{B_2} \in p'$$

Equation 1.3-2

where $\mathbf{B}$, $\mathbf{B_1}$ and $\mathbf{B_2}$ are bone's matrices while p and p' are two different skeleton poses.

To blend two animations the pose from both the animation curves are modified at each of the frames in the blending interval. Using this approach, when a new animation is required, the current movement can be blended into the new one in real-time, without having to wait the animation to end.

Actually, blending is used to solve a wider range of problems then the only animation mixing. For example it can be used to create new poses from existing ones, like the *blend shapes* method implemented inside *Maya*. Ideally, using this method, we can compute all the possible in-between positions of vertices from two (eventually even more) starting mesh configurations. One of the earliest innovations that led to motion blending was the real-time procedural animation system of Perlin (Perlin 1995). Blending operations were used to create new motions, and to change from one motion to another, presuming to have a manually constructed set of base animations. Blending operations can be used to get multi-target interpolation aiming to create parameterized motions as explored in several works (Wiley and Hahn 1997) (Rose, Sloan and Cohen 2001). Blending is a powerful way of synthesizing new data and is no surprise to see its application in continuous control of locomotion, for example (Park, Shin and Shin 2002).

1.3.3 Clip blending system in the motion model

To properly blend skeleton animations we need them to be divided and cut in a convenient way. We call *motion model clip* every short animation sequence that we obtain cutting the animation data in shorter animations; in the second half of chapter three we formally define this entity and the proprieties we want it to have.

The blending technique we developed is not only used for mixing clips together, it is also capable to prevent foot-skating during locomotion without inverse kinematics. To do this every motion clip is provided with constraints that supply, to the blending algorithm, information about the character's configuration. The way we define this constraints system is explained in detail in the chapter.

We now have all the elements we need to blend several clips into a single walk animation. Our blending method allows us to produce a valid animation mixing any sequence of clips, with no restrictions on which clip to use and, differently from other works, no explicit motion graph is required in this model. In other works, for example Kovar's et al. work, a direct graph is generated from motion capture data, and then, using a branch and bound algorithm, characters are made follow some sketched paths (Kovar, Gleicher and Phigin, Motion graphs 2002). Other works lean on a user-guided process, like Snap-Together Motion system which constructs well-connected graphs and where the searching procedure is made more efficient by using a simple cyclic graph structure (Kovar, Gleicher and Shin, et al. 2003). Other works studied efficiency for graph search by composing the whole from smaller environment related graphs; every animation block contains a graph which informs what actions are available for animated characters

within the block, and the final result is obtained combining these "motion patches" (Lee, Choi and Lee 2006), by pre-computing searching trees (Lau and Kuffner 2006), and using groups of similar motions to build better hierarchical motion graphs (Kwon and Shin 2005). Our motion model instead uses a graph structure that admits blending between any clips from the database and automatically prevents foot-skating during the mix. The only constraint we must impose is to choose clips from the set of those that starts with the appropriate foot; in the detail if the clips ends with a left footstep the set must contain clips that starts with the right footstep and vice-versa.

Now that we have defined in general the blending process as well as all the elements involved, the last step is to provide a formal definition to the process. To implement the blending algorithm we use a "re-rooting" system that continually swaps the skeleton's root from one foot to the other. To do this we need to have every clip recorded two times, one per starting foot. To simplify the mocap session all the clips were recorded with the left foot starting, and then were mirrored to get the right foot starting clips. Mirroring the animation without affecting the model's geometry is not straightforward, therefore we analyze a possible solution to this problem.

When every clip in the database has its mirror copy we are ready to define a control system that selects the next clip for the sequence at run-time.

1.4 Control System

The *control system* allows us to handle user's inputs in real-time and to control the character computing the next clip for the blending sequence. To this aim we define a *controller* which maintains an internal representation of the current system's state in order to choose the next clip. In chapter *4: Control system* we present how to define a controller and what we put inside the controller's state.

Controllers are capable to simulate the evolution of the system; once they know the current state they can decide how well is to add a clip to the sequence for a given *task*. Each controller is then correlated to a task, and each task can have several goals, like walking towards a given direction while maintaining the torso oriented to another direction. For different tasks, we obtain very different controllers; only one controller at a time can be active which means that only one task at a time can be pursued. For instance, the task we have implemented is navigation with user control which goals are to allow the user to control the character motion direction, gait and torso orientation. Another task may be to require the character to walk along a straight line while avoiding obstacles on it and we can even train a controller to move the character while avoiding some moving obstacles.

Task's goals are expressed as numbers using a *cost system*. Once we have defined tasks and costs we are capable to find the best clip in every situation, but we still leak a searching strategy. To define this strategy we introduce the concept of *policy*, which is a function that given the system's state decides

which is the best clip. In the last section of chapter four we introduce two possible policies, the straightforward *greedy policy*, and the more complex *near-optimal policy*.

The greedy policy searches for the clip which state's and transition's cost are the smallest. This approach can lead to unrealistic results since motion usually requires planning, like paying higher costs in the first step in order to pay less in the following steps. To tackle this problem we generate the near-optimal policy using an artificial intelligent technique called reinforcement learning. This method allows us to explore the decision steps in the future with an infinite horizon obtaining an optimal decision. Unfortunately, optimality is out of our reach since we are moving inside a continuous domain; we will need to approximate the optimum using a finite domain and this approximation leads to the near-optimal policy we implemented.

Once the controller is defined the character control system can be said completed. Our controller will choose, in real-time, the next clip after having queried the selected policy. The policy correctness grants us that the avatar will behave in the correct way achieving his tasks.

1.5 Software implementation

To obtain our results we developed two software tools, a first tool for clips creation and a program that runs the controller and shows the avatar in real-time. The software tool, called `ClipBlender`, is written in C++ and it takes advantage of a graphic engine developed in the spare time by the author of this thesis. The engine is called *Psyche*; it is cross-platform (Windows and GNU/Linux compatible) and uses OpenGL for rendering. The engine also provides an exporter for 3DStudio Max 9.0 that can export both geometry and skeleton animation data in a proprietary file format called `EXO` (from the word `EXOdus`). These files can be directly exported from 3DStudio Max 9.0 or eventually they are generated by the tool we developed, `Clip Editor`. `EXO` files can contain both rendering and/or animation information. The engine allows to load the rendering data from one file and to link the animation data, read from another file, to it.

The `Clip Editor` tool allows to generate a proprietary file, identified with extension `ECB` (`EXO Clip Blender`), that is used to contain information like clips list, skeleton's bones correlations, feet bone's index, constraints and so on. This file can be created using the `Clip Editor` software and can be read from *Psyche*. `EXO` files are grouped using a list stored inside `ECB` files.

The controller program, called *Character Controller*, is able to read the list from `ECB` files, and then it can load the animations data and the mesh accordingly. In this program we implemented all the algorithms discussed in this thesis and we used it to compute the results exposed in chapter five

Part II

MOTION MODEL

2: Animation Framework

2.1 Introduction

First, we introduce here the model that we have adopted to animate our character and how we have built it. We then introduce motion capture and the SMART system, used in this thesis, which is based on passive markers. We also describe the motion capture sessions that we have performed to obtain the motion data required to animate our character. Lastly we describe the procedure to convert the motion capture linear data in the matrix representation required for animation. To this aim, we have developed a script that is optimized for our task.

2.2 Skeletal Animation

Skeletal animation is the most widely used method for characters animation. In skeletal animation a character is split into two parts, a graphic representation, constituted of a textured mesh, and a hierarchical set of links (roughly corresponding to bones) connected through joints (roughly corresponding to the body joints). This set resembles the human skeleton and it is therefore named generically skeleton. The mesh is treated like a skin which covers the skeleton. The underlying idea of this subdivision is to use the skeleton information, of lower dimensionality, to interact with the graphical representation of higher dimensionality, in a simple and intuitive way. The strength of this model is the possibility, for the animator, to control few objects to achieve complex movements; its weakness is that it does not provide any surface deformation due to muscle contraction or wrinkle formation; moreover, a detailed representation of the joint's motion is still beyond reach, especially for complex bony structures like ankle, shoulder and vertebrae; in fact, their true motion is often complex as it requires sliding and rotations that cannot be capture by the simplified model of the skeleton. A simplified representation of these joints is usually accepted, in the form of simple spherical joints.

2.3 The avatar model

The "dress" model is a collection of polygons, usually triangles, namely a mesh. To define a mesh, first a list of 3D points is defined, followed by the definition of all the triangles that constitute the

mesh: for each triangle the index of its vertices in the list of 3D points is given. Attributes associated to each vertex are also defined, typically the normal direction that allows smoothing the color field when final rendering is applied. Additional properties can be added to the different triangles, called faces, like degree of transparency and texture; the latter, in particular, is widely used to give realistic appearance to the mesh. A texture is a 2D (or 3D) image that is given along its mapping over the mesh that is realized assigning a correspondence between the vertexes and the pixels in the mesh. Adequate hardware provides assigning and interpolating the pixels to the mesh faces.

Our avatar model is composed of a single textured mesh, created by myself manually from scratch using *Autodesk 3ds Max 9.0*. The mesh counts 1908 vertices and 3783 faces and it is covered with a texture of 2048x2048 pixels. The character is a simplified digital representation of the author (Figure 2.3-1) and both the texture and the model were developed using his anatomical structure.

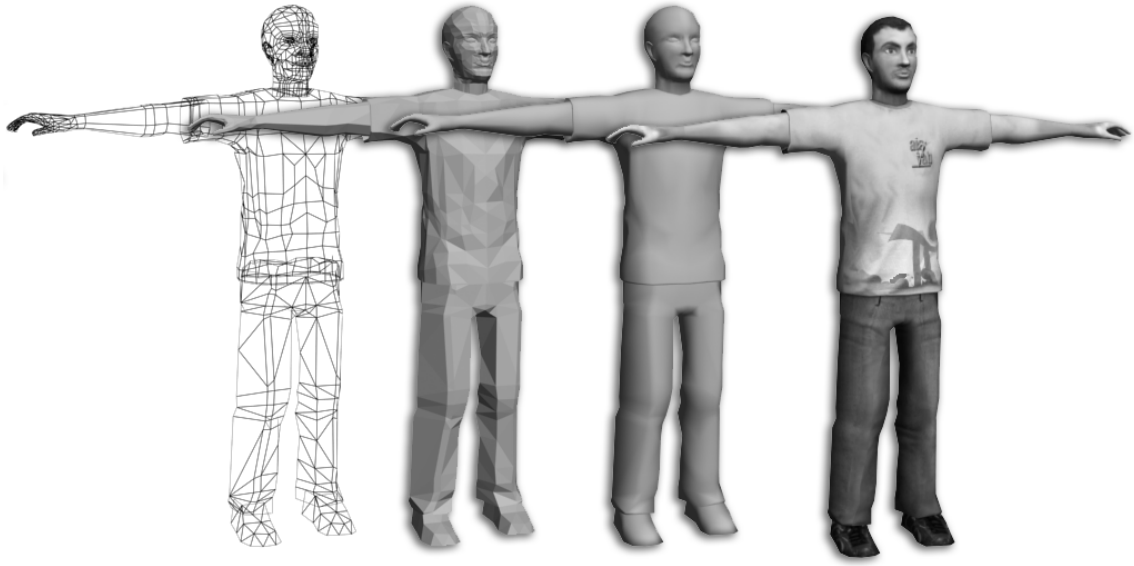

Figure 2.3-1: From left to right, mesh in wireframe rendering, mesh in flat shading, mesh Gouraud shading and mesh in textured rendering

The animation of the entire mesh is not feasible for two reasons. The first one is that the position of all the vertexes should be defined for each frame.

Since the mesh is composed of thousands of vertices it would be impossible to directly edit every vertex's information for every animation's frame. To simplify the definition of the motion, we need a method for reducing the volume of data to manipulate. An effective technique for this goal is *skeletal animation*.

2.3.1 Skeleton

Formally, the topology of a skeleton is an *open directed graph*, or *tree* (sometimes called *hierarchy*). One joint is selected as the *root* and the other joints are connected in a hierarchy (Figure 2.3-1). Differently from the other joints, the root has no restrictions on its motion. This joint is particularly critical as it identifies the global position of the whole skeleton; moreover, skeleton motion is computed starting from this root joint.

The nodes in the tree represent the joints of the skeleton therefore a node directly above another in the tree is that node's *parent*. All nodes will have exactly one parent except for the root node, which has none. A node directly below another in the tree is that node's *child*. Any node can have zero or more children. Child nodes inherit transformation from their parent, so if a leg is rotated the foot will follow the same rotation. Because of this, any node will implicitly inherit transformations from all the parent nodes found in the path from the node itself to the root. The inheritance is handled through forward kinematics and relies on matrix concatenation. Below the tree representation of our skeleton is shown.

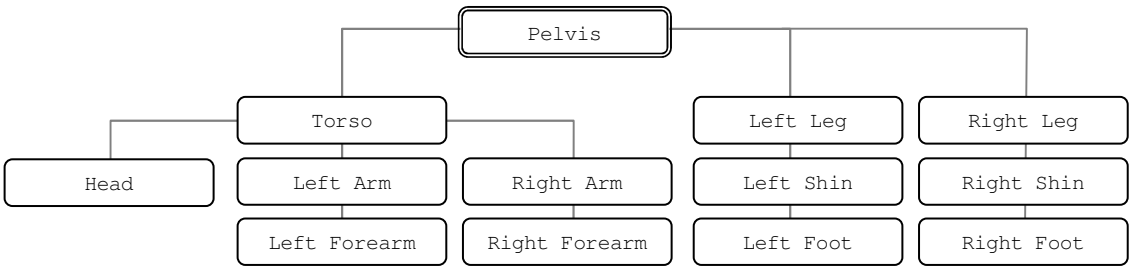

Figure 2.3-2: Our skeleton's tree representation

Information at every node can be saved in several ways; one can use Euler angles, quaternions or matrices and the transformations can be relative to the parent node or can be absolute. In our animation model a skeleton is a collection of matrices where every matrix $\mathbf{B} \in \mathrm{M}(4,4,\mathbb{R})$ and the transformation is expressed in absolute coordinates. It is well known that 4-by-4 matrices can represent the rigid transformations (roto-translations and deformations), within a given reference system, thus each joint of the skeleton is fully described by a transformation matrix, which includes all position, rotation and eventually scale information.

In particular, for each joint, the matrix **B**, according to classical notation in robotics and computer animation, contains the position of the joint higher in the hierarchy with respect to the absolute reference frame. As far as orientation is concerned, the X axis is usually oriented as the bone axis, exiting from the higher joint, while the orientation of the other two orthogonal axes has to be determined. We will set the orientation of these two axes according to geometrical considerations as described later.

A generic bone's matrix can be represented as follows:

$$\begin{pmatrix} n & n_y & n_z & 0 \\ t_x & t_y & t_z & 0 \\ b_x & b_y & b_z & 0 \\ x & y & z & 1 \end{pmatrix}$$

Equation 2.3-1

where $n =< n_x, n_y, n_z >$, $t =< t_x, t_y, t_z >$, $b =< b_x, b_y, b_z >$ forms the orthonormal basis that represent the bone's orientation in space while vector $< x, y, z >$ is the position of the joint higher in the hierarchy. Once we defined all the bones of a skeleton we can refer to this set as to a pose. A pose $p \in \mathrm{M}(4,4,\mathbb{R})^n$, where n is the number of bones in the skeleton, is a collection of matrices which specifies every joint position in the kinematic skeleton. We define a skeletal animation to be fully represented by a finite sequence of poses.

The skeleton considered for the present work is constituted of thirteen bones as shown in the right side of Figure 2.3-3. Every segment can rotate around the joint higher in the hierarchy, with three degrees of freedom that express three independent rotations of the bone with respect to this joint. Every bone in the model is stored using absolute coordinates therefore there is no concatenation to the root bone which acts like any other bone. For roto-translating the whole skeleton we defined a specific root-matrix that, by default, is the identity matrix. The root-matrix allows rotations around all the three axe and is obtained from:

$$\mathbf{R} = R_x(\alpha)\, R_y(\beta)\, R_z(\gamma)\, \mathbf{P}$$

Equation 2.3-2

where functions α, β and γ are the rotations angles around respectively x, y and z axis. Functions R_x, R_y and R_z provide the rotation matrix around the respective subscript axis and are described in detail in Appendix A while matrix **P** describes the root position.

Once the definition of the skeleton has been completed we must set a correspondence between the mesh's vertices and the skeleton's joints such that moving the bones result in deforming the overlying mesh. This operation is often called *skinning*.

2.3.2 Skinning

To bind the graphical representation through the mesh to the skeleton every mesh's vertex is tied to at least one bone segment through an *influence* or weight(Parent 2001). Every vertex can be tied to

several bones, and each weight specifies how much influence every bone has over the vertex itself. For every bone linked to the vertex, one and only one weight is defined, and the sum of all the weights for any of the vertices must be exactly *one*. As we already said, the process of linking the mesh to the skeleton is often called *skinning*, because the mesh can be seen as a virtual skin covering the bones. This process is usually performed by an animator using modeling/animation software, such as 3D Studio or Maya, and it often requires some manual editing.

The posture of the mesh (and of the skeleton), when skinning is performed, is called bind pose (Figure 2.3-3). In the present case, the avatar is standing with legs slightly apart and arm unfolded. Differently from the figure, when the animator performs the skinning process, the skeleton and the mesh are overlapped. When the skinning starts the program tries to automatically link any vertex to the closest bone. The animator can then manually edit the influence of the bones for achieving better results.

The bind pose is very important for animating the mesh. In fact, when vertices are saved, they are

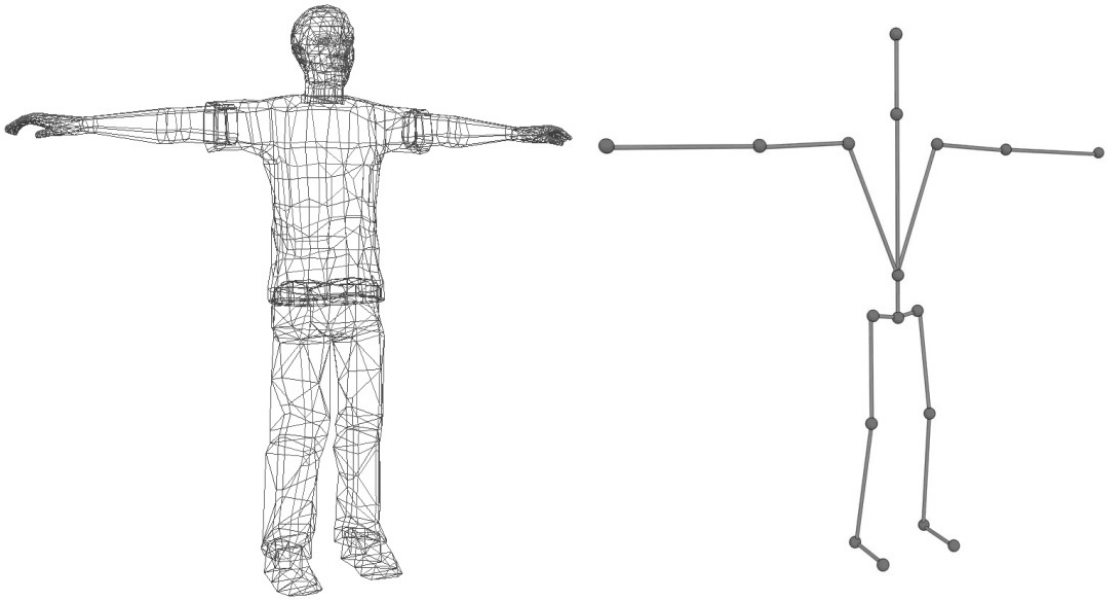

Figure 2.3-3: On the left the mesh, on the right the skeleton

stored in the position of the bind pose. Since we are saving the transformations as absolute movements, we must use the bone's matrix inverse to move the vertices "away" from the bind pose before applying the appropriate transformation that represents the motion for the next frame. This is described in the following formula:

$$v' = v \sum_{i=0}^{n} \left(\mathbf{B}_{(\mathrm{i,bp})}^{-1}\, \mathbf{B}_{(\mathrm{i,f})}\right) w_i$$

Equation 2.3-3

where v is the vertex at the current frame and v' at the next frame. $\mathbf{B}^{-1}_{(\mathrm{i,bp})}$ is the bind pose inverse matrix for the bone 'i' that is the matrix that defines the position of all the bones with respect to the absolute reference frame in the bind pose, $\mathbf{B}_{(\mathrm{i,f})}$ is the current frame bone matrix for bone 'i', that is the matrix that the defines the position of all the bones with respect to the absolute reference frame in the actual pose, and w_i is the weight of each bone for the vertex v.

Using Equation 2.3-3, at a given frame f, for every vertex in the mesh, results in moving the mesh's vertices in the new position obtaining a new attitude for our avatar. Therefore we need to define all the bones' matrices for all the frames of the animation. Setting every bone's position at every single frame is a very time consuming task. For one animation that last 30 seconds, playing at 25 frames per seconds, which uses a skeleton composed of 13 bones, we are going to define 9750 bones' transformations. These are too much for any animator to handle. This problem leads us to *key framing*.

2.4 Key frames

A *key frame,* in computer graphics skeleton animation, is a specific frame where we manually set the position and attitude of the skeleton. This frame defines the starting and the ending points of a smooth transition. Key frames are defined in some specific moments over the timeline and provide both spatial and time information. The remaining frames are filled using *in-betweens* techniques. Inbetweening (or just *tweening*), in skeletal animation, is the process of generating intermediate frames, between two skeleton attitudes, to give the appearance that the first bones' configuration evolves smoothly into the second one.

The main advantage of this technique is that the animator is not forced to set the skeleton at every single frame while the biggest disadvantage is a lower control over the "in-betweened" frames.

In characters animation the in-betweening techniques have another big problem. The human movement is highly coordinated and respects several movement spatiotemporal constraints. Any alteration of these constraints, even a small one, leads to unnatural character movements. Thus is very difficult to achieve any realistic animation using key frames and inbetweening techniques. Although few attempts to synthesize human motion from scratch(Faloutsos, van de Panne and Terzopoulos 2001), this task is still unfeasible for complex human motion.

If we cannot rely on key frames animations we need another technique to achieve the animation data for every frame. This technique should provide the high number of frames we are looking for without having to manually define every bone. This technique is called *motion capture*.

2.5 Motion Capture

2.5.1 Motion Capture

The term motion capture refers to an animation technique used for recording a movement and reproducing it on a digital model. Scott Dyer, Jeff Martin, and John Zulauf, in their paper on the subject, explain that motion capture "involves measuring an object's position and orientation in physical space, then recording that information in a computer-usable form. Objects of interest include human and non-human bodies, facial expressions, camera or light positions, and other elements in a scene." (Dyer, Martin and Zulauf 1995). Since in this work motion capture was used solely to acquire human motion, we will refer to this kind of acquisition with this term. We will also use the generic term "*mocap*" as abbreviation for motion capture. We explicitly remark that motion capture is aimed to acquire the motion disregarding all the appearance aspects of the actor: color, lighting and texture.

To perform a motion capture session movements from the actor are sampled at a frequency sufficiently high to get the impression of continues motion when motion is replied: typically capture sessions sample the motion at 30Hz or 60Hz. The animation data acquired are then mapped onto a 3D model so that the model performs the same actions recorded from the actor.

Mocap techniques have several applications in the most disparate fields, from medicine to video-games, and can be achieved using several technologies, one of the most diffused is the *optical* one which is based on video cameras that acquire images as a video stream. Using the images taken from several cameras the 3D position of some repere points, can be computed by triangulation from the different video streams. In optical systems the repere points onto the actor are identified by markers. *Passive optical systems* use marker coated with a retro-reflective material to retro reflect the light rays generated by a ring of flashing lights, positioned around every camera and synchronized with the camera shutter. *Active optical systems* employ, instead of retro-reflective markers, a set of LEDs rapidly illuminated one at a time and then relying on software capable to identify their position. An improvement of active optical system employs *time modulated active markers* which takes advantage of amplitude modulation or pulse width to transmit marker's ID information. Emerging techniques and research in motion capture is aiming to a markerless approach to mocap. The visual hull of an object can be defined as the locally convex over approximation of the volume occupied; its reconstruction consists in the projection of the object's silhouette from each of the cameras' plane back to the 3D volume. Having the hull is then possible to reconstruct the motion of the subject. (Corazza, et al. 2006). Other then optical methods there are several motion capture technologies that lean on different principles. *Inertial systems* for example are based on inertial sensors, biomechanical models and sensor fusion algorithms. Most inertial systems use gyroscopes to measure rotations that are then translated into skeleton data from software algorithms. Other systems may use *mechanical motion* capture devices that track joint angles using an exoskeleton to measure joint

angles in real-time without occlusion problems, or *magnetic systems*, which compute position and orientation using orthogonal coils.

The interest in using motion capture for character animation increased a lot in the last years because this technique can provide motion data for all degrees of freedom at a very high level of detail. Mocap allows obtaining data of good quality in a short time and it does reduce the cost of classical key-frame based animation; moreover, complex and physical realistic interactions such as weights and exchange of forces can be easily recreated in an accurate manner. On the other hand we have a lack of flexibility since it is difficult to modify captured motion tracks and repeating the acquisitions can be too costly or even impossible. A general approach to provide better ways of editing motion capture is to adapt the motion to different constraints while preserving the style of the original motion. Witkin and Popović studied a method to blend clips at some specified key frames that an animator manually set (Witkin and Popović 1995). Wiley and Hahn developed a method to mix samples from a motion capture database to create new animation that matches required specification (Wiley and Hahn 1997). Rose et al. developed a method which leans on radial basis functions and polynomials to interpolate between example motions while maintaining inverse kinematics constraints (Rose, Cohen and Bodenheimer 1998). However all these methods do not allow creating movements that are too different from the original ones; techniques aimed to reuse captured data linking them to form long animation sequences have become of large interest.

2.5.2 SMART – Motion Capture System

Performing a mocap session is a complex task composed of a number of phases that Lisa Marie Naugle identifies in her paper (Naugle 1990) as *studio set-up*, *calibration of capture area*, *capture of movement*, *clean-up of data* and *post-processing of data*. To accomplish every phase described by Naugle, and obtain the animation data, we had to rearrange the whole lab for the acquisition sessions and to deploy the SMART mocap system to acquire the bigger volume available.

The system we used is a six cameras configuration from BTS-Bioengineering (B&W cameras, 640 x 480 pixel at 8bpp, 30-60Hz) provided with three software, a capture program, a tracker and an analyzer.

Goal of the acquisition was to record gait clips at different speeds and turning angle, each of them containing four walking steps.

The choice of the repere points on which attach the markers is the result of a compromise between the number of markers and the cleanness of the motion tracks: the more are the markers, the more detailed is the acquisition of the motion; however, increases the probability of markers hiding, swapping or missing.

As a result, we have analyzed gait motion and we have identified the following repere points: neck base, pelvis front, pelvis back, left shoulder, left elbow, left wrist, right shoulder, right elbow, right

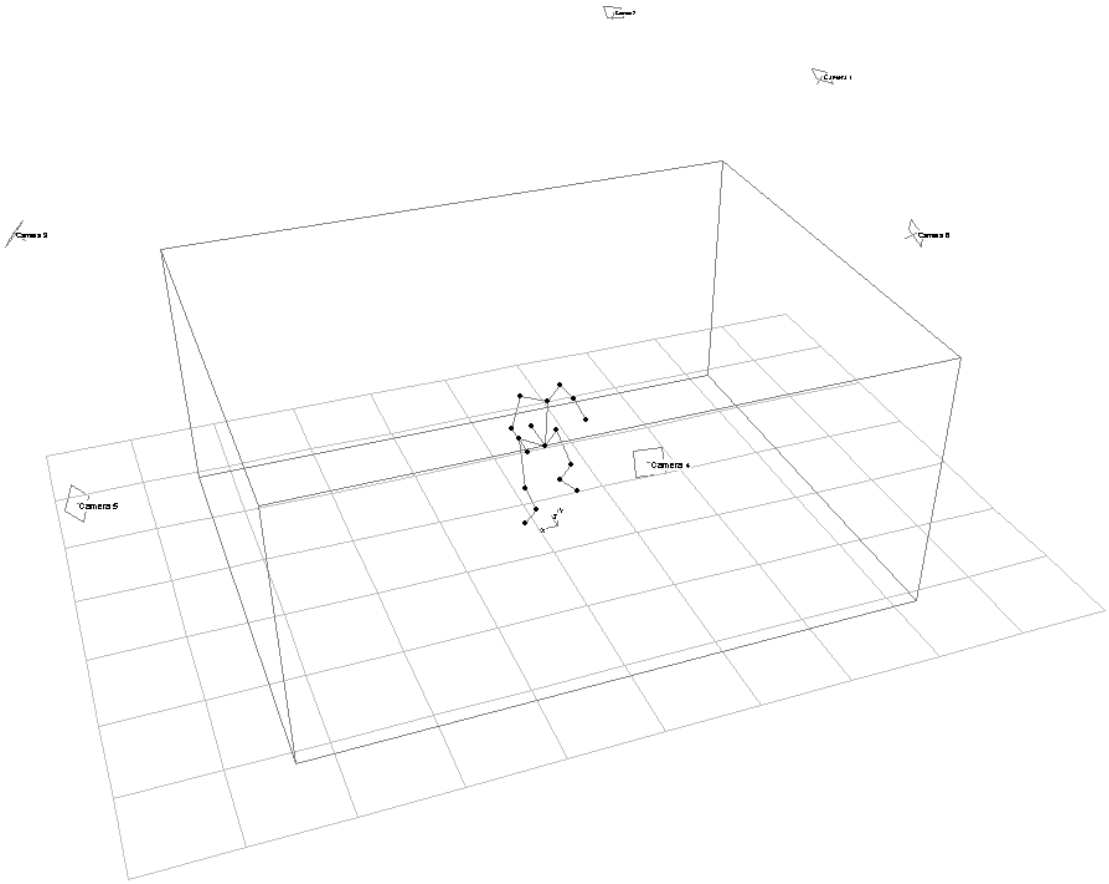

Figure 2.5-1: Work volume and cameras positions.

wrist, left greater trochanter, left knee, left malleolus, left foot metacarpus, right greater trochanter, right knee, right malleolus and right foot metacarpus. We explicitly take into account that the elbow and the knee have only one degree of freedom and therefore the bi-normal of arm and forearm, and thigh and shank, is the same and it is equivalent to that of the entire limb (a deeper explanation about mapping can be found in section 2.5.4).

To set-up the laboratory for acquisition we first disposed the six cameras in a hexagon, as close as possible to the lab walls to guarantee the largest working volume: in the final configuration we were able to capture motion inside a parallelepiped five meters long, four meters width and two meters high. It took two days to find the optimal configuration. Since we needed to acquire four steps, we measured the space required for a fast walk of four steps that turned out to be about four meters long. The only way to

successfully capture such length was to walk along the diagonal line of the lab (acquisitions tend to have more missing data when the actor is near to the volume boundaries). We had therefore to reorganize the laboratory furniture: we moved some desks and rolled the carpet of the AIBO robots soccer field to make enough space for performing the different motions. When the cameras were in position and correctly oriented, we moved to the second phase: calibration.

We spent a lot of time in setting up our system because a poor configuration can lead to lot of issues, in fact motion capture optical systems often suffer of *noise* and *missing data* problems. Even with good cameras configuration it is still possible to run into such problems, mainly because of bad calibration.

The SMART system calibration consists in two steps; in the first one a reference system, composed of three axes with markers mounted on each, is placed in the middle of the acquisition area. Capturing a dozen of seconds is usually enough for the system to identify the axes position and orientation. In this phase the absolute reference system is set, and it is usually set in the center of the acquisition volume with the Z axis orthogonal to the laboratory floor.

The second step requires walking around in the working area slowly moving a wand carrying three markers. In this step the system refines the estimate of the calibration parameters and identifies also the working volume. This step could be done in a couple of minutes but calibration would turn out really poor. After several attempts we discovered that about ten minutes of calibration were needed to have a well calibrated system.

As the system was ready, markers were attached to the actor body in the repere points and his motion was acquired several times: roughly 130 gait clips, of four steps each, have been captured overall, at different speeds and for different turning angles. Every acquisition lasted from about 20 to 40 seconds.

Having followed all the phases but two (we postpone the *clean-up of data* and the *post-processing* phases) the motion capture session can be said done.

2.5.3 Noise and Missing Data

The captured raw data usually include errors and noise. These issues often stem from a bad calibration and/or from generic system error measurement as well as unintentional marker movements related to cloths or skin motion and of wrong labeling of the markers. It may also happen that some markers disappear for several frames from the view of most cameras and therefore the system is not able any more to reconstruct their 3D position by triangulation. When they reappear some time later these markers have to be correctly classified. Missing data problems are due to marker occlusions that may occur while the actor is performing. To be able to use the acquired data we need to solve all this problems. This lead us the first of the two phases we had not analyzed before. The "*clean-up of data"* phase.

Clean-up is made during the *tracking process*. Since, in passive optical systems, markers are not automatically identified during the acquisition, there is no information on how to discern markers among

them. The tracking process is therefore a semi-automatic procedure that requires identifying one frame to manually assign to each point 3D track an identification label. This procedure is carried out once for each trial. During the process we are forced to deal with noise and missing data problems.

When one starts tracking the software automatically tries to reconstruct the motion of each marker and tries to follow it over the whole acquisition. This is not always possible, therefore the first output of tracking is a set of strings, each associated to one marker and associated to part of the motion time interval. If no data are lost for a marker, it is sufficient to give a label to the marker track as it spans the whole motion time interval. If the marker track is lost due to noise, wrong marker automatic labeling or disappearing of the marker, the track will result interrupted and it may restart again later in time. In this case the animator must edit the track before being able to assign the label properly. The tracker software provides some manual tools to deal with these problems, in particular it allows manually labeling markers in some frames or assigning the same label to multiple partial tracks. When tracking is done it most of the noise from the raw data should have been removed.

Missing data issues cannot be directly handled during tracking. Obviously these problems lead to interrupted tracks, but while the tracks can be edited to incorporate or eliminate markers not correctly labeled, not all the hole can be fitted just by tracking. Some other tool is required. The SMART Analyzer tool permits to fill data gaps employing interpolation techniques based on complex data analysis. Any moving point is analyzed by the tool which computes several information such as its velocity and its acceleration; then the analyzer tries to guess where the point could have been if it had not get lost. This covers the last phase, *post-processing*.

To speed up the acquisition process we did not use the SMART Analyzer tool; instead, we developed a less elaborated but efficient interpolation algorithm that was added into the tool that we wrote to blend the different clips, the `ClipBlender` tool (see *2.6 Software implementation*). Aim of this tool is to complete automatically the tracks when imported avoiding to manual check every recorded sample. The algorithm is based on computing a linear interpolation between the last known 3D positions of a marker before and after the gap; since we are not dealing with any complex motion the simple space interpolation results to work very well without any needs of managing speed or acceleration information that allows higher order interpolation that cannot always be applied.

The “hole filling” algorithm scans all the markers at every frame, looking for any missing position data. When it founds a missing marker it looks backward and forward, within a defined frame window, for any valid data to use for interpolation. We discovered that the algorithm works better with smaller windows and multiple iterations instead of a single run with a larger window.

Once we finally have the data correctly tracked and cleaned up, we needed to convert them into skeleton’s animation bones matrices. Currently one of the most difficult problems in motion capture is mapping the acquired linear raw data into animation angles suitable to the 3D animation models; therefore it is important to understand why these issues arise and which solutions are available.

2.5.4 Mapping

One difficult problem is the linear motion of the markers into rotations of the joints angles. We have seen that markers motion is acquired as a set of 3D moving points but the motion has to be represented as rotations of the skeleton bones.

There are several ways to perform this task and a lot of research has been done on the argument, for example see Choi, et al. (Choi, Park and Ko 1999), which presents an algorithm that processes the original joint angle data to produce a new motion in which the end-effector error is reduced to zero at keyframes. Other works propose the use of physical models (Zordan and Van Der Horst 2003) or the use of least-squares fitting technique as recently presented by Wen, et al. (Wen, et al. 2006).

Our approach is related to the particular model we used and turns out to be particularly efficient. It was implemented directly into the `ClipBlender` tool. The approach is specific for the bone hierarchy we decided to use; this choice allowed us to speed up the mapping's phase but prevent the algorithm from being used on different configurations.

2.5.5 Skeleton's Frenet frames

The idea beyond the mapping algorithm is to get a *Frenet frame* for each segment of the skeleton. Once we have the Frenet frame is straightforward to compute the bone transformation matrix $\mathbf{B}_{(i,f)}$ for that segment. A Frenet frame is made of three orthonormal vectors, normal, tangent and bi-normal, which define a basis for the 3D space. We want to extract these vectors from the point representation. Since, as we've seen before, our acquisitions were sampled using the number of markers as small as possible, some information was missing, forcing us to estimate some of the Frenet components.

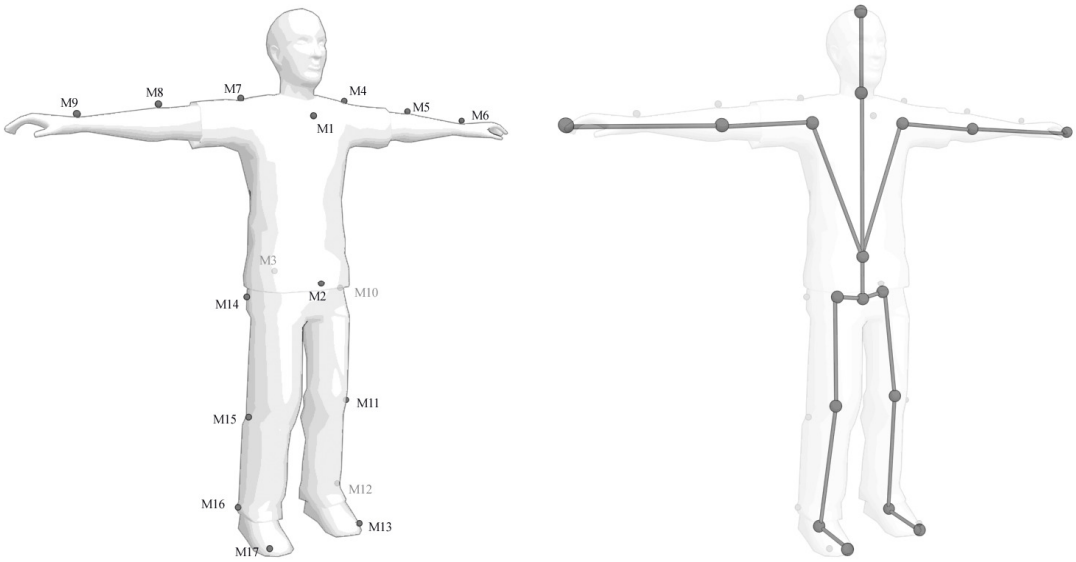

Figure 2.5-2: On the left the marker disposal over the actor, on the right the skeleton representation

The torso has enough markers (`M1`, `M2` and `M3` as visible in Figure 2.5-2) and we were able to compute its Frenet frame without any problem. For torso here we intend the whole trunk from the pelvis to the neck, notice that we do not allow the spine to twist. The pelvis bone was computed in a very similar way using the same markers.

Arms and forearms had three markers overall (`M4`, `M5` and `M6` for the left arm, `M7`, `M8` and `M9` for the right arm): one on the wrist, another on the elbow and the last on the shoulder. When these three points are not aligned they lay on a plane which can be used to determinate the whole arm orientation. Let us analyze the details for the derivation of the Frenet frame for the forearm segment (Figure 2.5-3). Having the three joint points we can compute forearm and arm vectors from pairs of markers. Then, with two cross products, we can derivate the other vectors:

$$v_2 = \mathrm{M7} - \mathrm{M8}$$
$$v_1 = M9 - M8$$
$$v_3 = v_1 \times v_2$$
$$v_4 = v_3 \times v_1$$

Vectors v_1, v_3 and v_4 together form the Frenet frame for the forearm segment. When the three points are aligned there are infinite planes passing from them so it is not possible to compute the exact

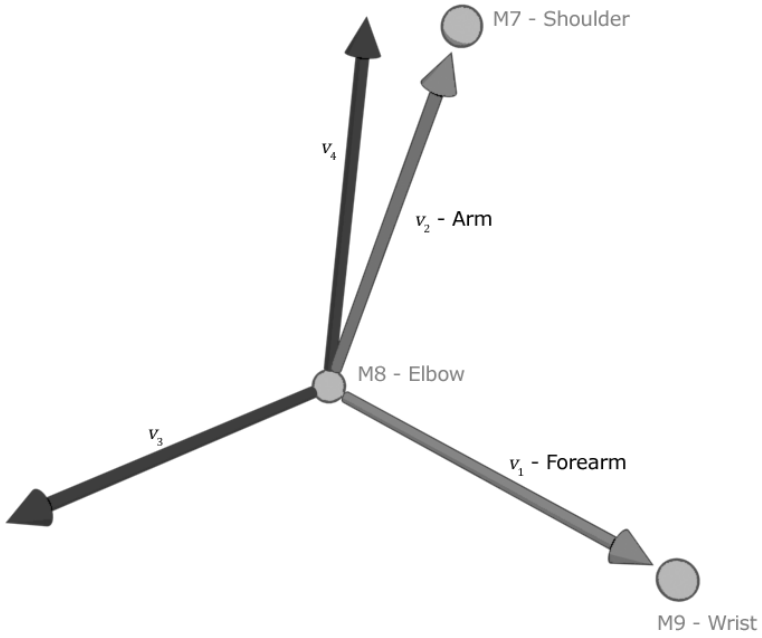

Figure 2.5-3: Forearm's Frenet frame construction

orientation. In this case an estimate has to be made. In detail we derive a vector v_3', combining the

shoulders' markers M7 and M4. Vector v_3' defines where the character's side points to. The derivation is done as follows:

$$v_3' = M7 - M4$$
$$v_4 = v_3' \times v_1$$
$$v_3 = v_1 \times v_4$$

The Frenet frames for the thigh and the leg were computed in a similar way since they have also three markers attached (greater trochanter, knee and ankle). The foot has been considered as a rigid body and approximated to a single segment. This is a strong assumption for gain, in which, during the push off phase the forefoot forms an angle with the rest of the foot to propel the body forward. Nevertheless this assumption was made to simplify both the capture sessions and the subsequent processing. Anyway, the most difficult problem was to compute the Frenet frame of the head.

2.5.6 Head's Frenet frame

To reduce the number of markers to track, no markers were attached to the actor's head. This would have required making the working volume higher introducing problems in capturing accurately feet motion. Therefore, to obtain a reference system for the head, we decided to use the data from the torso's Frenet frame to estimate the position of the skull.

In the first attempt we copied the Frenet frame from the torso to the head, keeping the same orientation and taking as origin of the Frenet frame the neck joint, identified by the marker attached to the neck onto the actor. This approach led to unrealistic movements because the head looked like stuck to the torso.

To tackle this problem, we decided to discard both *pitch* and *roll* motions, that are the motions around the *normal* and the *bi-normal* axis (the vectors marked as **n** and **b** in Figure 2.5-4), when copying the Frenet frame. We imagined that the character would try to maintain his sight parallel to the floor even if the torso is bent forward or backwards. This is inspired to human motion description work that hypothesize that the eyes and the head are like a stable platform from which reconstruct the body motion(Pozzo, Berthoz and Lefort 1992). To achieve this we force the tangent vector to be orthogonal to the floor, which means that we impose $\mathbf{t}' = < 0,1,0 >$. We can then use cross product operation to derive the other two axes as follows:

$$\mathbf{b}' = \mathbf{t}' \times \mathbf{n}$$
$$\mathbf{n}' = \mathbf{b}' \times \mathbf{t}'$$

Equation 2.5-1

where vectors $\mathbf{b}'$, $\mathbf{t}'$ and $\mathbf{n}'$ form the new Frenet frame for the head, while $\mathbf{b}$, $\mathbf{t}$ and $\mathbf{n}$ are read from the original head's Frenet frame (Figure 2.5-4).

Moreover, to get a more realistic feeling for the head motion, we observe that the head has to look forward when walking straight and turns some instants before the body turns. For simulating this motion knowledge we needed to know when and how the character will turn. To do this we read the torso

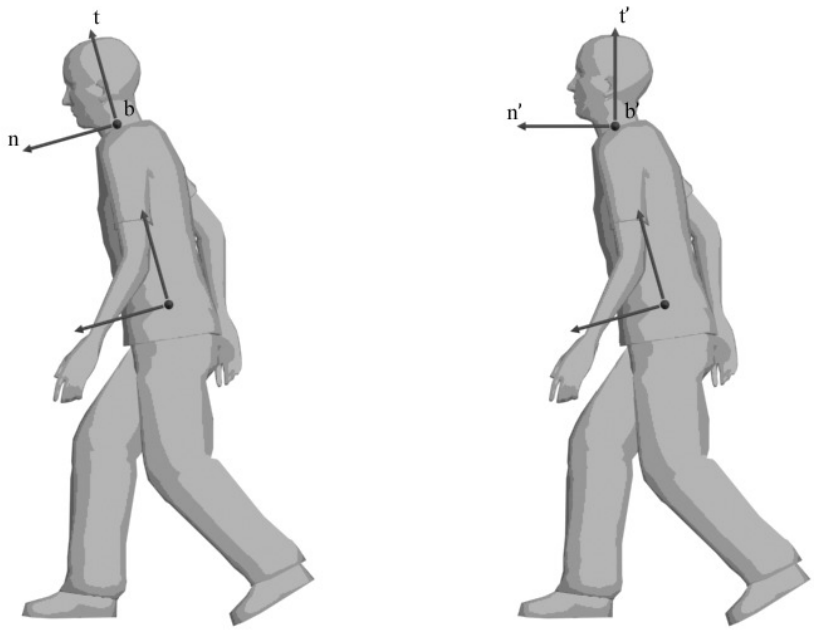

Figure 2.5-4: On the left, the Frenet frame for the head is copied from the torso, on the right the Frenet frame modified

information ten frames forward, with respect to the current frame, and we computed, using these data, the Frenet frame for the skull. Thus, if in the near future the torso will be turned, the head starts to spin in that direction anticipating the movement of the rest of the body. If, from the current frame, we are not able to read ten frames forward, because there are no so many frames left in the animation, we use the current torso's frame information. This solution worked quite well since the head pointed to the motion direction just before turning, giving a realistic motion feeling

The last problem we faced was related to the marker-joint correspondence. Markers position is not coincident with the position of the skeleton corresponding joint since markers are applied over the actor skin and not inside his bone's joints. Because of those problems a little post-processing of the markers position was needed. Re-targeting is a well known solution for adapting motion capture data (Gleicher 1998), that works well especially when trajectory changes are small. In our case retargeting involved some adjustments in markers' position in order to make them fit segments' length and joints' position.

In details, after having computed the bi-normal vector for the arm and the forearm (vector v_3 in Figure 2.5-3) we moved the markers of the limb along the bi-normal direction to pull them as close to the

joints as possible. Since we are trying to estimate the joint's position we have no information of how far we must move the marker to reach the joint itself; the amount depends on the skeleton's configuration therefore we have to factor it out by trial and error. To this aim we defined a basic amount for translating the arm's markers and we used a one-frame animation to test the mapping's results. If the arm and the forearm were still not in place we modified the translation amount and we tried again until the character's limb looked fine. Exactly the same operation was done for the leg and the foreleg.

The neck marker required slight adjustments too: we inverted the normal vector from the torso's Frenet frame (that is the vector that points straight out from the chest) and then we moved the marker's point as close as possible to the neck's joint position. We do not have the joint's position for this case either, so we have performed these operations by trial and error too; we knew that the neck's marker is a few centimeters down, along the normal vector, from the related joint, therefore we determine the neck joint's position starting from the marker's position and moving back of some centimeters. Again, we played the test animation and we adjusted the translation amount until the head looked in position.

Besides the arm and the forearm segments of the skeleton were slightly longer if compared to the segments obtained from markers' position. Therefore, after having moved the markers closer to joints as seen before, the elbow and the wrist were moved further from the shoulder point along the shoulder-to-elbow vector, and then the wrist was moved further from the elbow using the elbow-to-wrist vector. Both the movements' amount was computed by trial and error.

Once the mapping is completed the skeleton can be said to be fully animated.

2.6 Software implementation

All the discussed algorithms are implemented inside `ClipBlender`. Clips may be added from an `EXO` file or from a `C3D` file. `C3D` files store raw 3D coordinate and analog sample data and are directly exported by the SMART3D Tracker tool. When a clip is added from a `C3D` file, the tool loads automatically the data from this file and converts them into bone's animation matrices saving the result to an `EXO` file; the list of the available animation clips is then updated with this file.

The conversion task is committed to a *class* named `C3DEXOConverter`. This class receives a pointer to the `C3D` file itself, a frame index from which starting the conversion and a second frame index in order to know where to stop the conversion. Any object instantiated from `C3DEXOConverter` allows converting the data by invoking the appropriate method. The object automatically loads the data from the `C3D` file, but it requires the explicit invocation of the conversion method before starting the procedure; the conversion process will complete the tracks and converts the motion curve into bones expressed in Frenet frames using absolute coordinates. The output of this conversion task is a file in `EXO` format that contains the bone animation.

2.7 Conclusion summary

Animation is carried out defining an adequate skeleton, a mesh and skinning the mesh over the skeleton. To animate the skeleton, we resorted to motion capture and we have developed an efficient technique to complete the motion capture track and convert them into sequences of joint angles that can be used to animate the skeleton inside the animation tools, typically 3D Studio Max.

The skeleton, in turns, propagates its motion to the mesh and a realistic animation of the defined avatar is obtained. The motion clips acquired for this work have all the same number of markers and are constituted of walks at different speed and turning angles to produce a dense library of gait clips.

3: Blending System

3.1 Introduction

In this chapter we describe the technique implemented to blend different animation clips acquired by motion capture to obtain a single continuous set of animation curves.

3.2 Animation blending

Animation blending will not produce realistic results unless the input motion clips are chosen with some care. Blending process depends heavily on how much information about animations is given to the algorithm, if nothing is known about the input other than the raw motion parameters, linear blending algorithms are reliable only if the two motions are quite similar. Defining the blending algorithm can be challenging; therefore, to tackle the problems that blending presents, we require that the clips to blend have some specific features.

First, we impose that *clips overlap using the same frame amount*. Since we are working on transitions we typically blend the last set of frames from the first animation with the first set of frames from the second animation. If we decide to use, for example, ten frames, we must specify which frames are read from the end of the first animation and which are read from the start of the second animation, but anyway we are forced to use only ten frames.

The second property we want, in order to have a pleasant and smooth blend, is that *logically related events must occur simultaneously and in the same absolute position* in the two motion clips. For example we will incur in blending anomalies if we try to blend animations that totally have no spatial correlations (i.e. different absolute positions); the result will be some kind of quick distortion that moves the character from one position to the other, deforming the whole mesh. Linear blending can fail when motions have different timing, that is, when corresponding events occur at different absolute times(Kovar and Gleicher, Flexible automatic motion blending with registration curves 2003). This point is very important and we can imagine a lot of situations where blending is going to show strange behaviors if animations have no correlations. The correlation we want for the two animations must be provided on two levels: on a timing level, that is specifying when corresponding events occur at the same absolute time, and on a frame alignment level, that is a spatial correspondence for the two animations. For example if we have two walking motions that are in phase, but one curving 90 degrees to the left and one curving 90

degrees to the right, when blended the result will be a root path that collapse and eventually flips around(Kovar and Gleicher, Flexible automatic motion blending with registration curves 2003).

There are several solutions to these issues. For having clips that have the same frame amount used for blending we must work out a method for identifying which frames to use in every animation curve. We will see how this can be done further in this chapter.

A lot more troublesome is providing a logical correlation between the two clips. Two skeletal animations, with different joint speeds and root position, can be very difficult to blend in a single realistic motion animation if we have no information on what the animations are representing, how they are oriented and where the skeletons are placed in space.

We will solve these problems marking all the frames that contain logically correlated events and using this information to overlap events and count how many frames are involved in the overlap. We are going to explain in detail the solutions we adopted, but, before being able to do it, we need to agree on a suitable terminology. Since we are dealing with walk animations we will borrow the bioengineer's definition of gait cycle.

3.3 Gait cycles

In normal walks a gait cycle beings when one foot contacts the ground and ends when that foot hits the ground again. Each cycle contains two main phases, *stance phase* and *swing phase*. Stance phase accounts for approximately sixty percent, and swing phase for approximately forty percent, of a single gait

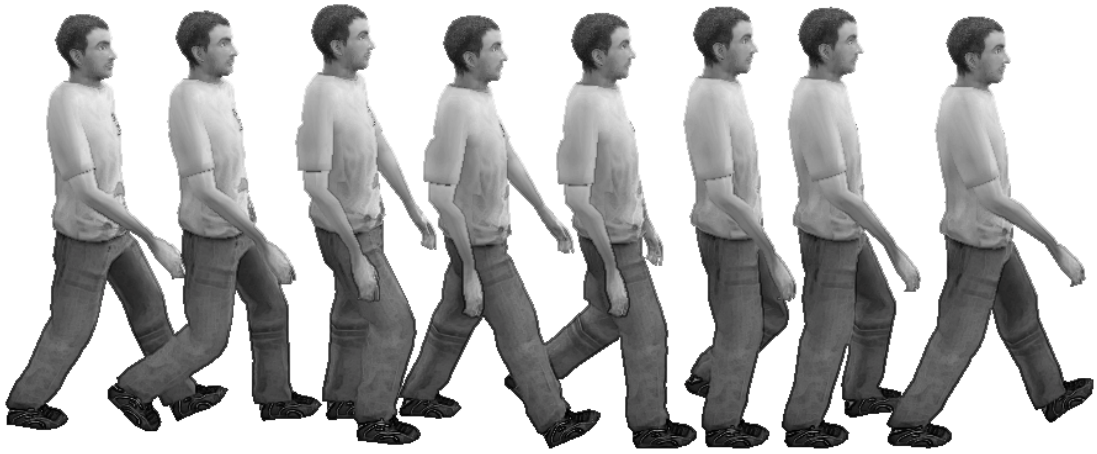

Figure 3.3-1: Gait cycle showing every begin/end pair

cycle. Stance phase of gait is divided into four periods: *loading response*, *midstance, terminal stance* and *pre-swing*. Swing phase is divided into three periods: *initial swing, midswing* and *terminal swing*. The beginning and the ending of each period is identified by specific events:

Loading response
- ⇒ Begins with *initial contact* (also known as heel strike)
- ⇒ Ends with *contra-lateral toe off*, when the opposite extremity leaves the ground

Midstance
- ⇒ Begins with *contra-lateral toe off*
- ⇒ Ends with *heel rise* when the center of gravity is directly over the reference foot

Terminal stance
- ⇒ Begins with *heel rise*
- ⇒ Ends with *opposite initial contact*, when contra-lateral foot contacts the ground

Pre-swing
- ⇒ Begins with *opposite initial contact*
- ⇒ Ends at *toe off*

Initial swing
- ⇒ Begins with *toe off*
- ⇒ Ends with maximum knee flexion when feet are adjacent

Midswing
- ⇒ Begins with *feet adjacent*
- ⇒ Ends when the tibia is perpendicular to the ground

Terminal swing
- ⇒ Begins when the tibia is perpendicular to the ground
- ⇒ Ends with *initial contact*

Now that we have defined a common terminology about gait cycles we have all the elements to face the blending problems discussed before. In details, we impose that the animation clips that we want to blend will respect some proprieties, and this will allow us to solve both the logical correlation problem and the length problem. The proprieties we want are obtained cutting the acquired animations so that they cover a single gait cycle and then defining a constraints system. We will call these shorter animations *motion model clips* or just *clips*. From here we will use the term *clip* solely to refer to these motion model clips.

3.4 Motion model clips

3.4.1 Clips definition

As said before, we define one clip to cover exactly one gait cycle, which is an animation that starts in the initial contact position and ends after a full gait cycle. Notice that a clip, defined this way, necessarily has two periods where only one of the two feet is in ground contact phase. The first one is at the beginning of the terminal stance while the second one is at the beginning of the mid-swing. We can easily cut the motion capture animations so that they start and end in the proper way.

Then we will have our motion capture animations adapted into a set of clips $\mathcal{C}$ where each element $C \in \mathcal{C}$ consists in a sequence of poses $C = (p_1, \dots, p_n)$ where n is the number of poses. Since we want to blend one clip into another we split each sequence of poses into two subsequences, paying attention so that each one includes at least one ground contact frame, and we can then use the second subsequence from the first clip over the first subsequence of the second clip. Since we defined that every

clip covers one single walk cycle we will call the two subsequences *walk cycle in* and *walk cycle out*. The former begins at heel strike and ends at opposite initial contact. The latter begins with pre-swing period

Figure 3.4-1: Clip struture. Notice the poses inside the clip and the subsequence subdivision

and ends when the terminal swing ends. This particular division allows us to provide a logical correlation between every clip in the database because of the following consideration.

If the first subsequence contains a step with the left foot, then the second subsequence must contain the next step with the right foot, and vice-versa. Therefore we know which foot is carrying the weight in every subsequence and this provides us both a position and an orientation correlation. For example, let say that we want to blend two clips, C^1 and C^2. If we have that the skeleton for clip C^1 has its weight on the right foot in its second subsequence, we want the skeleton in C^2 to have the right foot stepping forward in the first half of the animation, and then we can overlap the two subsequences. With this splitting method we can be sure to have a walk that never steps forward with the same foot two times, and we can assure that the second clip is positioned and oriented exactly in the same way of the first one just caring to overlap their weight feet. Summarizing, the two clips are playing the footstep with the same foot (logical correlation), they can be oriented in the same direction using the foot orientation and they can be repositioned to be in the same place overlapping their feet positions. Once we have formally defined a way to overlap the clips we have solved the logical correlation problem for blending.

We still have to face the length problem; as we said, blending requires the two clips to have the same length. Currently we are not providing any information about the starting nor the ending point for the blend, therefore we need to define how *walk cycle in* and *walk cycle out* are obtained from the clip and how the blending system should use them for blending. Moreover, the two subdivided sets usually will not match and we need to decide which of the overlapped frames we will discard.

We can solve both the overlapping problem and the length problem defining a constraint system for our motion model clips' set.

3.4.2 Clips Constraints System

To overlap the clips we need to know which frame in each subsequence contains the pose that represent the full ground contact instant. It is difficult to automatically identify this frame, even if we can try to find it analyzing the segments' vertical velocities the so obtained results are unreliable, therefore, to

have a more accurate estimate, we will require an animator to manually mark it. We call the marked frame from the first subsequence C_{in} and the one from the second subsequence C_{out}. Once we have identified the two frames in every animation we can overlap two clips so that C_{out}^1 fit together with C_{in}^2 as shown in Figure 3.4-2. This provides the logical correlation and allows us compute the frames involved in blending for both the clips. Clip one uses frames from $C_{out}^1 - C_{in}^2$ to C_{out}^1 , while clip two uses frames from 0 to $C_{in}^2 + n^1 - C_{out}^1$. Therefore we can define the interpolation value t_b, for blending, as follows:

$$t_b = clamp_{(0,1)}\left(\frac{f^1 - (C_{out}^1 - C_{in}^2)}{C_{in}^2 + n^1 - C_{out}^1}\right)$$

Equation 3.4-1

where f^1 is the current frame read from clip one and n^1 is the number of frames from the same clip. Now that we have the interpolation value we can use it in Equation 1.3-2, as t, to compute the new blended

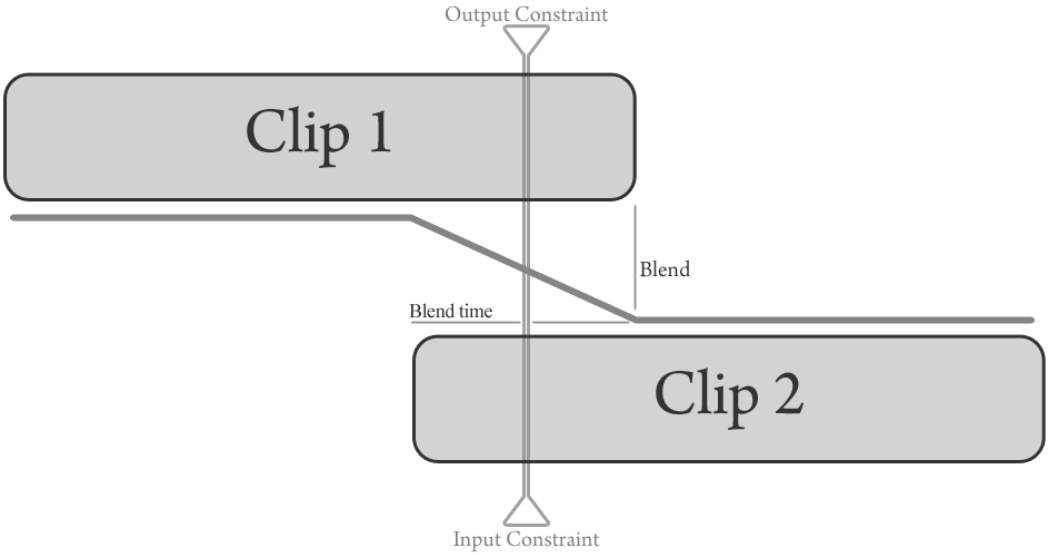

Figure 3.4-2: Two overlapped clips

skeleton pose. We have now solved both our problems, we have isolated the frames we want to overlap, and we have a correlation given by the constraint system we set.

Having defined the clip constraint system in this way has another big advantage; it allows us to prevent foot-skating, during locomotion, without using any inverse kinematics method. *Inverse kinematics* (abbreviate *IK*) is a type of motion planning used for determining the parameters of a kinematic chain in order to achieve the goal pose (Zhao and Badler 1989). This motion planning method allows the animator to define the position of the "end-effector" charging the software to factor out all the values for the involved joint angles. Therefore the animator can specify the foot position during the walk and the algorithm will do everything else. Inverse kinematics has its own disadvantages too; as it is based on minimizing a geometric cost function, it doesn't always choose the most normal pose, can lock under

certain conditions and doesn't avoid self collisions. These drawbacks in IK methods are difficult to tackle or solve, often undermining this technique's efficiency. The constraint system we use can achieve the same results of an inverse kinematics algorithm, but paying smaller costs, although it is thought to work only for legs and during a walk animation.

Foot-skate prevention in our method is made possible because of two pre-processing steps. First, before blending two clips we can arbitrarily re-orient and re-position the root of the incoming clip preserving continuity. This operation is done when we overlap the clips using their constrain frame. Second, kinematic blending is linear in the root of the skeleton, although nonlinear in all other skeleton joints. Therefore if we properly transform the foot position and orientation while "re-rooting" the skeleton at its foot, we can satisfy a foot position constraint. As we saw, the second animation is placed over the first one so that the foot is overlapped with the same foot from the first animation. If we re-root the skeleton at is foot and if the foot is fixed during interval $[t_a, t_b]$ in subsequence *walk cycle out* and during $[t'_a, t'_b]$ in subsequence *walk cycle in*, then, by the linearity of blending at the root, there cannot be foot-skate on the interval $[t_a, t_b] \cap [t'_a, t'_b]$ of the blended animation. Since we have manually marked the frames where foot is not moving, we are sure that in the intersection of intervals no foot-skating can happen for the constrained foot.

There is a last issue, not directly related with blending, that we have to solve. All the clips from the database are recorded with the left foot stepping forward, but we also need clips to start with the right foot for the overlaying system to work. To avoid capturing all the clips a second time we decided to *mirror* every animation previously acquired.

3.4.3 Mirroring animations

Mirroring a mesh is usually a simple task, but what we want to do here is not the plain mesh reflection, instead we want to reflect the sole animation's motion without any change to the mesh's geometry; this goal can be quite challenging. Note that if we simply reflect the skeleton we will also reflect all the mesh's vertices that are binded to the bones, moving, for example, the right arm's vertices to the left side of the mesh. To reflect the sole animation we need to reflect the skeleton paying attention to preserve the reference system of every bone and to re-bind the mesh to the new mirrored skeleton. Let first describe what a regular mirroring process involve.

In mathematics, a reflection is a map that transforms an object into its mirror image. In order to reflect a three-dimensional object one should use a plane for mirroring. In detail, to find the reflection of a point, a perpendicular line is drawn from the point to the plane used for reflection, and continues for the same distance on the other side. To find the reflection of an object, for example a polygonal mesh, one reflects each vertex in the mesh. Since the reflection preserve distances between points the reflection operation is an isometry.

In three-dimensional geometry mirroring can be done using matrix multiplication. Mirroring operation can be seen as a change of reference system where one of the axes is reflected towards the other side. The following mirror matrix $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$ reflects any point multiplied against it using the x-y plane as the mirror plane.

Here we meet the first problem. Unfortunately we cannot just mirror all the bones of the skeleton using an x-y mirror matrix since this will flip the z axis in every bones' matrix (obviously the same problem happens using any other mirror plane, it just change the axis that flips). If this happens we will have that the whole model reverted along the z axis, and this is a problem since we do not want to have a right arm that blends to a left arm (Figure 3.4-3).

To avoid this issue we need to preserve the Frenet frame composition avoiding that any axis flips when the system is mirrored. Since we cannot directly prevent the axis to be reverted, we manipulate the Frenet frame locally before the mirroring happens. Therefore, if we flip only the axis that we know will

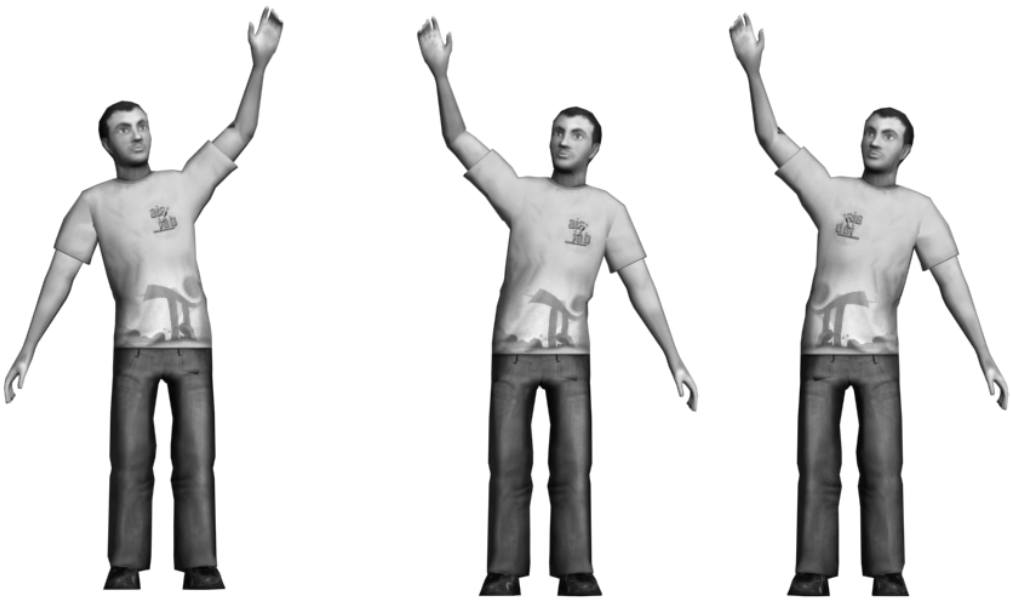

Figure 3.4-3: On the left the character's pose, in the middle the correctly mirrored pose and on the right a wrong mirrored pose.

revert, at the very beginning of the mirroring, we will have the Frenet frame locally inverted. Then we can use the transformation matrix as usual, but the frame now will have the z axis flipped and, if we mirror the bone this time, we get the frame correctly oriented (Figure 3.4-3 in the middle. Notice the logo on the shirt that is still on the right).

This approach solves our first issue but we still have to deal with the second problem. Let recall that our goal is to have, for example, the right arm that acts like the left arm and vice-versa; therefore any

bone on the left side must be swapped to the right side and vice-versa. Notice that this also means that we need to re-bind the whole mesh after having mirrored the skeleton.

That being so we divide mirroring equations in two different categories; the ones that reflect bones that lies along the character symmetry mirror plane, and those that reflect bones far from the plane. The bones far from the plane are the bones that must be swapped to grant, for example, the left arm to act as the right arm and vice-versa.

Let define the mirroring formula for the first category of bones, those that have no correspondences (for instance in our skeleton they are only `Pelivs`, `Torso` and `Head` bones). Recalling Equation 2.3-3 we see that every bone $\mathbf{B}$ is pre-multiplied for its bind pose inverse

$$\mathbf{B} = \mathbf{B}_{(\mathrm{i,bp})}^{-1}\,\mathbf{B}_{(\mathrm{i,f})}$$

Equation 3.4-2

so, assuming that the model is facing along the x positive axis, we can mirror the z axis, after the bind pose multiplication, using a local mirror matrix; then we act normally using the current frame matrix, and then we multiply for the world mirror matrix.

$$\mathbf{B} = \mathbf{B}_{(\mathrm{i,bp})}^{-1}\,\mathbf{M_{local}}\mathbf{B}_{(\mathrm{i,f})}\,\mathbf{M_{world}}$$

Equation 3.4-3

where $\mathbf{M_{local}}$ should mirror the character along the symmetry plane and must be computed in relation to the bind pose, that is, if the model is facing the positive z axis $\mathbf{M_{local}}$ will be an x-y mirror matrix. $\mathbf{M_{world}}$ mirrors the animation reverting the same axis flipped using $\mathbf{M_{local}}$.

We handle the second category of bones now. If the bone has a correspondence on the other side, then it is far from the symmetry plane and we must use the bind pose from the correlated bone, instead of its own bind pose, as follows:

$$\mathbf{B} = \mathbf{C}_{(\mathrm{i,bp})}^{-1}\,\mathbf{M_{local}}\mathbf{B}_{(\mathrm{i,f})}\,\mathbf{M_{world}}$$

Equation 3.4-4

where $\mathbf{C}_{(\mathrm{i,bp})}^{-1}$ is the bind pose inverse of the correlated bone. Thus we need to save, for every bone that uses this formula, the correlated first frame matrix for mirroring. If we imagine the first category of bones to be self-correlated, for example we correlate the head to itself, we can use the sole Equation 3.4-4 for mirroring. Mirroring the whole set of clips doubles their number and provides to our blending engine the missing animation needed. Finally, since now we have all the animations, we can discuss the blending system itself.

3.5 Clip blending

3.5.1 Blending process

Blending between two clips C^1 and C^2 is a three step process. In the first step we overlap the two clips at their constraint frames, respectively C^1_{out} and C^2_{in} (as shown in Figure 3.4-2). Therefore we will work, in the following steps, using the constraint frames we defined.

In the second step we re-orient C^2 so that its ground-contact foot coincides with the same foot from C^1. We are sure that feet are in ground-contact because they are taken from the constraint frames. In detail, when the foot will be in the position described at C^2_{in}, it will be overlapped onto the same foot, from clip one, read at frame C^1_{out}. We don't want a "perfect" overlap, we require their feet to match in positions and in orientation only on the x-z plane (that is the rotation around the Y axis and the position on the X-Z axis). We discard pitch and roll motions (around X and Z axis) because we are supposing that the character is walking on a horizontal plane.

Re-orienting and re-positioning are both represented by a single roto-translation matrix, therefore we need to factor out this matrix to accomplish step two. We remember from section 3.4.2 that we must re-root the second clip, before overlapping, and that this will also assure foot-skate prevention. We also recall that our bones are expressed in absolute coordinates thus we defined a root-matrix that transform the skeleton in the space as the root bone should have done if we were using relative coordinates.

During the animation of clip two we update the root-matrix using the inverse transformation of the foot defined at frame C^2_{in}. Since we want a partial overlap we need to change the foot's transformation matrix, in order to discard the unused components, and then we invert it. In detail we break down the matrix, let call it $\mathbf{B^2}$, and use its parts as follows:

$$
\begin{aligned}
n &= normal\left(\mathbf{B^2}_{(\mathrm{i},C^2_{in})}\right) \\
x &= (1 \quad 0 \quad 0 \quad 0)^{\mathrm{T}} \\
\theta &= angle(n, x) \\
\mathbf{P} &= positionXZ\left(\mathbf{B^2}_{(\mathrm{i},C^2_{in})}\right) \\
\mathbf{R_2} &= \left(R_y(\theta) \cdot \mathbf{P}\right)^{-1} \cdot \mathbf{T_2}
\end{aligned}
$$

Equation 3.5-1

where i (in the matrix subscript) is the index for the foot bone in the skeleton, $\mathbf{R_2}$ is the root-matrix for clip two (functions $normal$, $angle$, $positionXZ$ and R_y are defined in *Appendix A:Mathematical notations*) and $\mathbf{T_2}$ is the transformation matrix for the skeleton in clip one, that is the position and orientation of the skeleton measured at the constrained foot in respect to the absolute reference frame.

The first part of $\mathbf{R_2}$, that is $\left(R_y(\theta)\ \cdot \mathbf{P}\right)^{-1}$, will move the whole skeleton so that the constrained foot will be in the origin when the animation plays frame C_{in}^2. If we now apply a transformation that moves the skeleton from the origin to the position where is placed the constrained foot from clip one, we have effectively re-positioned clip two onto clip one. This is exactly the role of matrix $\mathbf{T_2}$. We can compute this matrix as follows:

$$\mathbf{B}^{\mathbf{1}}_{(\mathrm{j},C^1_{out})} = \mathbf{B}^{\mathbf{1}}_{(\mathrm{j},C^1_{out})}\,\mathbf{R_1}$$
$$n = normal\left(\mathbf{B}^{\mathbf{1}}_{(\mathrm{j},C^1_{out})}\right)$$
$$x = (1 \quad 0 \quad 0 \quad 0)^{\mathrm{T}}$$
$$\theta = angle(n, x)$$
$$\mathbf{P} = positionXZ\left(\mathbf{B}^{\mathbf{1}}_{(\mathrm{j},C^1_{out})}\right)$$
$$\mathbf{T_2} = R_y(\theta)\ \cdot \mathbf{P}$$

Equation 3.5-2

where $\mathbf{R_1}$ is the root-matrix for clip one and it is used for positioning the animation so that $\mathbf{B}^{\mathbf{1}}_{(\mathrm{j},C^1_{out})}$ contains the foot position in respect to the absolute reference frame. Notice that here index j in $\mathbf{B}^1$ represent the same foot of Equation 3.5-1 indexed using i in $\mathbf{B}^2$; this happens because one of the two clips is coming from the mirrored set, so the right foot uses the index of the left foot and vice-versa.

Multiplying the skeleton in clip two and the root-matrix we can complete step two overlapping the clips. Notice that clip one has its own root-matrix; when a new clip is added to the sequence the clip named two will be re-labeled as clip one and the newcomer will be labeled clip two, the same happens to the root matrix so $\mathbf{R_1}$ is the old $\mathbf{R_2}$. At the very first step instead, $\mathbf{R_1}$ is just the identity matrix.

Finally, we face the last step, which is blending from C^1 to C^2 with the ground contact foot treated as the root of the kinematic skeleton. The blend time is computed as shown in Equation 3.4-1. If we iterate this method for a sequence of clips we obtain a long seamless animation as shown in Figure 3.5-1.

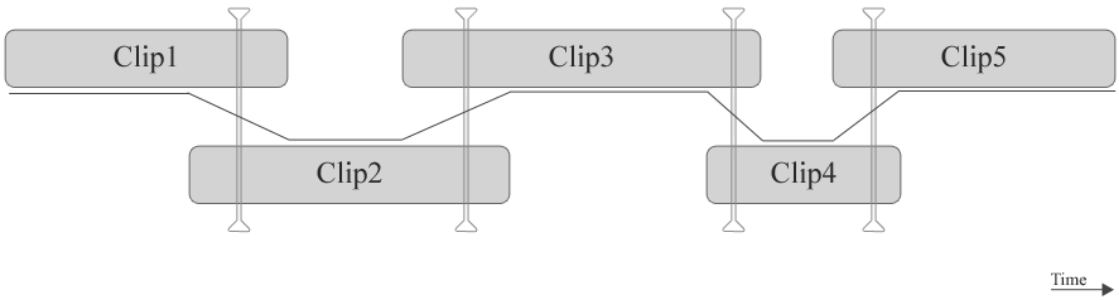

Figure 3.5-1: A sequence of blended clips

The sequence of clips is mixed in a seamless walk animation composed of the clips we used. Notice that, thank to the constraint system, the maximum blending occurs during ground contact, when forces are used for changing direction and both linear and angular momentum vary the most. Minimal blending occurs during initial contact and opposite initial contact periods.

The major drawback is that we use constraints one at a time, when both are on the ground only one of the two is constrained to don't skate. For the large class of motion that don't involve double stance, such as walking and running, this model works well. Because the motion model avoids non-smooth blends it doesn't need any graph structure of valid motions. Every transition is allowed and valid therefore we are granting a very high branching factor that allows quick changes of direction.

3.6 Conclusion summary

Blending allows mixing a sequence of animations, in particular we blend clips captured using motion capture, to obtain a single long walk animation. We prevent foot-skating using a constraint system and "re-rooting" the clip.

We have not defined yet how the sequence of clips is chosen and how we can generate it on the fly satisfying the user inputs. To achieve this goal we want to define a controller system.

Part III

CONTROL SYSTEM

4: Control system

4.1 Introduction

The sequence of clips to be blended is not defined a priori, instead it is dynamically generated in order to accomplish a specific task, which is defined a priori and gives the goal of the whole animation. To this aim we define the controller, which selects the next best clip for the sequence on the basis of the high level information of the current task.

We have implemented only one controller whose task is to navigate the character over a plane allowing the user to specify a desired motion direction, torso orientation and type of gait. These desires are encoded as parameters inside the controller itself, therefore the controller selects the next clip in function of the user's requirements besides of the overall goal. This allows the user to interact with the avatar as specified by the task during the animation itself at interactive rates.

In the first part, we propose a definition of a meaningful state valid for the controller and the task considered here and the parametric structure of the controller. We also define a way to move from the controller's current state to the next state that is obtained adding the selected clip to the sequence. We then introduce more formally the concept of task and in the second part we present some cost functions that allow the controller to choose each next clip; the clip is passed to a blender system that will mix the two clips smoothly. This chapter ends analyzing the policy system that we developed to choose each next clip and presents two possible policies, the greedy and the near-optimal one.

4.2 Controller and states

4.2.1 State

The character's controller allows the user controlling the avatar while it is accomplishing a given task, for example moving over a plane. To this aim a controller has the main function of supplying a way for changing from one state into the next one. A controller, therefore, must capture and handle the system state; however, these states cannot be defined a priori since they depend on what the controller's tasks are and on what the controller must keep track of. Therefore, the controller depends on the chosen task and then, to define a state, we must identify those variables that are important for our task. Once we have identified all the meaningful variables for every task we can save them inside the *state vector*.

State is represented using a vector $X \in \mathcal{S}$ (where $\mathcal{S}$ is the collection of all the possible states) containing all the variables that better describe the current character situation. For the task considered here, namely navigation with explicitly control of torso orientation, gait style and gait direction, we have chosen the following state variables. The first variable included in the state vector is the current clip $C \in \mathcal{C}$, where $\mathcal{C}$ is the set of all the available motion model clips; including clip C is required since we cannot define a state without knowing which clip we are playing. The next three state variables are the character's position and orientation on the x-z plane. These variables are named $x,\ z$ and θ and represent respectively the position along the X-axis, the position along Z-axis and the angle around the Y-axis, taken clockwise. The next variable is named τ and it is the angle between the torso's normal vector and the desired torso's orientation read in the very moment when the state is computed. The current torso orientation is read from the clip data at the current frame, while the desired torso orientation is an angle defined by user around the Y-axis taken clockwise. Finally, the last state variable represents the user's desires, in particular, in this case it represents the gait style that is coded as an integer $\bar{G}$. These last two variables, τ and $\bar{G}$, represent intentions and can be directly manipulated in real-time while the animation is created. As we said, all the named variables are inserted in a single state vector:

$$X = \begin{pmatrix} C \\ x \\ z \\ \theta \\ \tau \\ \bar{G} \end{pmatrix}$$

Equation 4.2-1

More explanations are required on the reference frame in which the position and orientation variables x, z and θ are expressed. Position and orientation variables are not expressed in the absolute reference frame, instead, every time a new clip is required, the reference system is changed an repositioned in a convenient way as explained in the next paragraph and shown in Figure 4.2-1; therefore position and orientation variables refer to a specific coordinate system that is updated at every clip switch. This means that every time a new clip is added to the sequence the state vector is no longer valid and it has to be computed again.

When blending of two clips occurs, we have two clips to blend, C^1, that is the last played clip (it can even being still playing for blending purposes), and C^2 that is the current clip; we then also introduce a third clip, C^3, that represents a possible next clip choice. We now reposition the reference system in the mentioned convenient way, that is, as shown in Figure 4.2-1, we place the reference system's origin at the position where feet overlap for blending, that is where the constraints C^2_{in} and C^1_{out} overlap (as specified in the first step of the blending algorithm, section 3.5.1). The reference frame is then re-oriented so that the X-axis is coincident with the desired direction specified by the user. Once the reference system has been set, we can finally define what is represented by the position and orientation state variables. In this scenario x and z represent the third footstep's position, that is where we have the constraint frame C^2_{out}, on

the x-z plane. Notice that foot one is the C_{in}^{1} position of the last played clip, while the fourth foot represented in the diagram is the possible C_{out}^{3} position computed using a generic third clip. The diagram represents three possible foot-four's position because this footstep is yet not defined at this level, so those steps are the possible positions obtained using three different clips overlapped on the third foot.

Summarizing, this diagram involves three clips: one that was played (or it is still playing while blended), one that is currently being played (eventually blending with the old clip) and another one that can be used for computing the next potential step. This last clip is not defined at this level, it is shown in the graph to give the idea of what would happen if a third clip is used and how the direction changes using different clips. Notice how the controller automatically knows where foot-two is, since it is always in the origin, and saves only the foot-three position explicitly. The first footstep does not interest the system then we just discard the information about it. As shown in the diagram θ is the angle embraced by the foot's "look" vector and the desired direction therefore angle θ represents the orientation of the second footstep in respect to the x-axis.

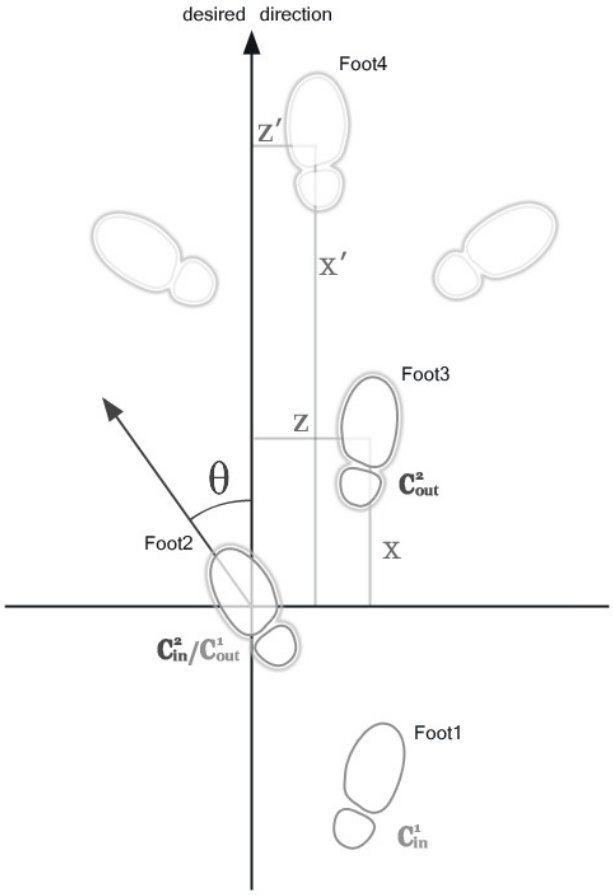

Figure 4.2-1: State coordinate system

The state vector captures everything that is relevant from the system so that the task can be achieved correctly. Now we want to define a transition function that computes the new state given any new clip. Once we define this function we will be able to simulate where the character will move after having performed a step with any of the clips from the database.

4.2.2 Transition from state to state

We define a *transition function* for our controller in order to change from a state to the next one. The transition function requires a new clip to be specified; choosing a clip for transitioning means fixing both the third clip and the fourth footstep we discussed before. Changing from a state X to another state X' is done according to the transition function $f: \mathcal{S} \times \mathcal{C} \rightarrow \mathcal{S}$ as follows:

$$f(X, C') = X' = \begin{pmatrix} C' \\ \mathrm{x}' \\ \mathrm{z}' \\ \theta' \\ \tau' \\ \bar{G} \end{pmatrix} = \begin{pmatrix} C' \\ (\cos(\theta)\Delta\mathrm{x} - \sin(\theta)\Delta\mathrm{z}) - \mathrm{x} \\ (\cos(\theta)\Delta\mathrm{z} + \sin(\theta)\Delta\mathrm{x}) - \mathrm{z} \\ \theta + \Delta\theta \\ \tau' \\ \bar{G} \end{pmatrix}$$

Equation 4.2-2

where C' is the new clip, x', z' and θ' represent the new position and orientation, τ' the new angle between torso's normal and torso desired orientation while $\bar{G}$ is copied as it is. Points x' and z' are computed in function of theta using two distances, $\Delta\mathrm{x}$ and $\Delta\mathrm{z}$. These distances are measured from the fourth to the second foot after having aligned the reference system so that the normal vector of the second foot coincides with the x-axis (Figure 4.2-2). Notice how these distances can be computed a priori after having chosen any two clips, they do not depend on the character's position nor on its orientation. In Figure 4.2-1 and Figure 4.2-2 there are three possible positions for foot-four obtained using three different clips; the middle one, labeled foot-four, is obtained using C' so it shows $x'$and z' coordinates. Angle difference $\Delta\theta$ is computed measuring the angle embraced between the normal vector from foot-two and the normal vector from foot-three:

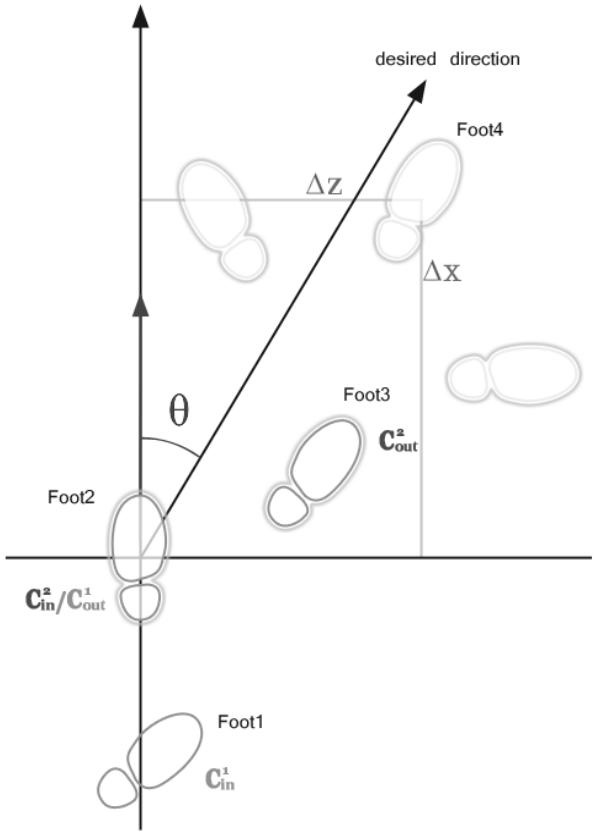

Figure 4.2-2: How Δx and Δz are measured

$$\Delta\theta = angle\left(normal\left(\mathbf{B}^{\mathbf{2}}_{(\mathrm{i},C^2_{out})}\right), normal\left(\mathbf{B}^{\mathbf{2}}_{(\mathrm{j},C^2_{in})}\right)\right)$$

Equation 4.2-3

where i is the index of foot-three's bone and j the index of foot-two's bone. Angle τ is computed measuring the angle embraced between the normal vector from the torso in the first clip and the normal vector $\hat{\tau}$ that represent the desired torso orientation:

$$\tau' = angle\left(normal\left(\mathbf{B}^{\mathbf{2}}_{(\mathrm{k},C^2_{out})}\right), \hat{\tau}\right)$$

Equation 4.2-4

where the index k refers to the torso's bone.

Once we have defined the state representation system, and a way to make it evolve in time using the transition function, we have to define an adequate cost function. We need to define a cost associated to each task because this allows to assign a value to a state with respect to the specific goals provided to the task itself. States cannot have any absolute value by themselves, they are just representing the current character's situation, but we can evaluate them with respect to a task, and then we can decide which transition will better fit in our goals.

4.3 Task

As said, a task, in our system, requires to specify the goals that the controller must achieve when choosing a clip; after having casted these goals into numerical values we will be able to define some strategy to accomplish the task optimally.

The task that we have considered is navigation. This, for example, requires the avatar to move on a plane giving to the user the ability to control motion direction, gait style and torso's orientation. We want that the character can move with different gaits; therefore, to do this, we captured different clips through motion capture, in each of which that subject walked at different speeds, eventually staying steady in place. These last type of clips are needed for starting and stopping the character's motion; for example it starts steady on place when the simulation begins. We divided the clips in different set based on three gait styles: steady, normal walking and fast walking. When asked to choose the next clip the controller will exclude those clips that are not in the set of the gait style required by the user.

We allow the user to specify a direction along which to move the character. The task will then require the character to walk along the linear path which is oriented as specified by the direction and which starts from the foot that is currently supporting the body weight. We also require the character to move forward as close as possible to the linear motion path, avoiding to have the character that walks with the legs apart.

The last goal is to allow the user to specify the torso orientation that the character must maintain while moving along the desired direction. This allows obtaining different kinds of walk, for example we can revert the torso direction with respect to motion direction, to achieve a backward walk.

Given the current task, we need a way to specify how good is a clip in achieving all the goals specified for the task. To this aim we use a cost system as this allows us to use reinforcement learning techniques to learn the optimal policy for clips selection.

4.4 Costs

4.4.1 Representation of the task goals using costs

We express the task's goals using a cost system that assigns a cost to every state and every transition. The underlying idea is that a more costly state, or transition, is less attractive then a cheaper one; therefore if we properly assign the cost to every new state and to all the possible transitions, we can estimate which transition will lead to a new state paying the lower cost. Costs are obtained from two functions:

$$c_s: \mathcal{S} \to \mathbb{R}$$
$$c_t: \mathcal{S} \times \mathcal{S} \to \mathbb{R}$$

Equation 4.4-1

where c_s is the *state cost function* and c_t is the *transition cost function*. This functions compute costs that are inversely proportional to how well the input state (or states for transition) fulfill the task's goals.

4.4.2 State cost

Every state has an inherent cost. We define the cost function as follows:

$$c_s(X) = \begin{cases} \gamma_z|z| + \gamma_\tau|\tau| & C \in \bar{G} \\ \infty & C \notin \bar{G} \end{cases}$$

Equation 4.4-2

In this function the first part tries to minimize the distance of the new step from the desired direction motion path. Parameter $\gamma_z$is used to weight this factor. The second part evaluates how far is the clip from the desired orientation. Again parameter γ_τ is used to weight this torso orientation factor. In our results we have set the weights $\gamma_z = 0.70$ and $\gamma_\tau = 2.10$; these values have been obtained by trial and error.

4.4.3 Transition cost

We impose a cost for changing from one state to another using the following function:

$$c_t(X, X') = \gamma_\Psi \Psi(C, C') + \gamma_\mu|\mu|$$

Equation 4.4-3

where the physical cost $\Psi: \mathcal{C} \times \mathcal{C} \to \mathbb{R}$ measures the physical error in blending from C to C' using a weighted sum of squared difference on position and velocity computed over all the bones. This function allows us to get the most natural transition possible from our library of clips. Ψ computes the squared difference of joints' position and velocity during the blending frames, that are those frames where clips are overlapped. These differences are weighted and summed together and the result will be in inverse proportion to the naturalness of the transition. This value is used as a cost, which allows us to choose the clip that produces the most natural transition. Lastly, μ is the angle between the vector from the origin to foot-three and the x-axis.

$$\mu = angle\left(\begin{pmatrix}1\\0\\0\end{pmatrix}, \begin{pmatrix}x'\\0\\z'\end{pmatrix}\right)$$

Equation 4.4-4

In our results we have set the weight of these constraints to $\gamma_\Psi = 0.40$ and $\gamma_\mu = 1.00$ (like the previous values, these parameters have been obtained by trial and error). The presented cost system provides a way to give a numerical value to every clip with respect to the task, but it is not saying anything about how to choose the next clip for the sequence. To this aim we need to define a *policy* that takes advantage of the cost system to choose the next clip.

4.5 Policies

4.5.1 Greedy policy

Differently from the standard definition used in reinforcement learning (where $\Pi: \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and $\Pi_t(s, a)$ is the probability that $a_t = a$ if $s_t = s$, see Appendix B.1 for more details) we define a policy as a function $\Pi: \mathcal{S} \to \mathcal{C}$ that, given a state, chooses the next clip to add to the sequence. The way the policy chooses the next clip depends on how we implement the policy's decision process. The most obvious policy is greedy, that is a policy whose strategy is to minimize the immediate cost of both the new clip and the transition to get to it:

$$\Pi_g(X) = \min_{C' \in \mathcal{C}} \big(c_t(X, X') + c_s(X')\big)$$

Equation 4.5-1

where Π_g denotes the greedy policy and X' is given by the transition function $X' = f(X, C')$ as shown in Equation 4.2-2. In our results we have seen that the greedy policy produces realistic movements.

In general, motion usually requires to be planned in advance in time in order to look natural, that is we can choose action in the immediate that are costly but that produce a global lower cost in the long run(Treuille, Lee and Popović 2007). Therefore, even if the greedy policy behaved well we have chosen to implement a more complex planning policy that is not based on short-term cost but to a long term one to achieve a better result in the long run. To this aim we designed a cost function capable to consider all the costs that a choice will impose to the system in the present and in the future. This approach is typical in reinforcement learning techniques and will help our system to choose the clip that will produce the most natural motion. The best of all the policies in reinforcement learning is called *optimal policy* (Appendix B.1.3).

4.5.2 Optimal policy

The optimal policy evaluates both the immediate cost of selecting the new clip and all the following costs that the system will incur in the future if that clip is chosen. Such a policy is optimal because it can consider an infinite number of steps in the future. To define the optimal policy let's suppose that we have a policy Π which produces the sequence of states $(X_0, X_1, X_2, \dots)$, where $X_t = f(X_{t-1}, \Pi(\mathrm{X_{t-1}}))$ and X_0 is known, then we will measure the cost of the whole sequence instead of the sole first step measured by the greedy policy. To do this we define a cost function that measures the cost of the first element of the sequence as:

$$c_\Pi(X_0) = \sum_{t=0}^{\infty} \alpha^t \left(c_t(X_t, X_{t+1}) + c_s(X_{t+1})\right)$$

Equation 4.5-2

where α is the *discount rate* and it is in $[0,1)$ to ensure the series to converge since we are setting no limits on the step number. The equation can be easily rewritten in a recursive and more general form:

$$c_\Pi(X_t) = c_t(X_t, X_{t+1}) + c_s(X_{t+1}) + \alpha c_\Pi(X_{t+1})$$

Equation 4.5-3

Under general condition, minimizing Equation 4.5-2 for each initial state results in obtaining the optimal policy Π_* (Bertsekas 2001). This allows us to define the optimal value function $V_{\Pi_*}: \mathcal{S} \to \mathbb{R}$ for

this policy that can be written as $V_{\Pi_*}(X) = c_{\Pi}(X)$, therefore, for all the X we can rewrite the previous equation as:

$$V_{\Pi_*}(X) = c_t(X, X') + c_s(X') + \alpha V_{\Pi_*}(X')$$

Equation 4.5-4

where $X' = f(X, C')$ and $C' = \Pi_*(\mathrm{X})$ that is the optimal next clip. Now that we have defined the value function for the optimal policy we can define the policy itself in a proper way.

$$\Pi_*(\mathrm{X}) = \min_{C' \in \mathcal{C}} \left(c_t(X, X') + \mathrm{c_s}(X') + \alpha V_{\Pi_*}(X') \right)$$

Equation 4.5-5

Therefore the optimal value function $V_{\Pi_*}(X)$ completely specifies the optimal controller. Unfortunately we cannot implement the optimal controller as it is; the problem is that we cannot compute the exact value of the optimal value function in a continuous states space, because this would mean to handle infinite possible states. Therefore the only way to pursue optimality is to achieve a near-optimal result through approximation of the value function; this is why we are going to introduce *basis functions approximation*.

4.5.3 Basis Function Approximation

We achieve approximation using a method based on linear programming proposed in the work "The linear programming approach to approximate dynamic programming" (de Farias and Roy 2003). The idea is to work on the cost system; the domain of this system is the states space whose size typically grows exponentially in the number of state variables (this problem is known as *curse of dimensionality*). To deal with this difficult problem one can try to approximate a value function V mapping it to an approximated function $\tilde{V}: \mathcal{S} \times \mathbb{R}^{\mathrm{n}} \to \mathbb{R}$, and then compute a vector $r \in \mathbb{R}^{\mathrm{n}}$ that "fit" with the original value function so that $\tilde{V} \approx V$. Vector r can be quite difficult to be computed because it heavily depends on what shape the value function has. In de Farias and Roy's work, to define $\tilde{V}$, they introduce a basis vector $\Phi(X) = (\varphi_1(X), \varphi_2(X), \dots, \varphi_n(X))$ where each basis $\varphi_i: \mathcal{S} \to \mathbb{R}$ can be evaluated in closed form, such as polynomials, and this leads to the approximation equation:

$$\tilde{V}(X) = \sum_{i=1}^{n} r_i \, \varphi_i(X)$$

Equation 4.5-6

This lead us to approximate the optimal value function defined previously as follows:

$$V_{\Pi_*}(X) \approx \sum_{i=1}^{n} r_i \, \varphi_i(X) = \Phi(X) r$$

Equation 4.5-7

Therefore we have reduced the problem of solving for the complete value function to the lower dimensional problem of solving for vector r that approximates it.

We now want to define basis vector Φ in order to be able to compute a vector r that approximates V_{Π_*}. These basis depends on the desired degree of approximation. We started our tries from the work done in the Treuille, Lee and Popović's paper and we defined the basis as polynomials of second degree in function of θ and z. In this paper the authors suggested to use for continuous basis function $\mathcal{P}_2(z)\mathcal{P}_2(\theta)$ where $\mathcal{P}_2$ is defined as $\mathcal{P}_2(x) = \{f(x) = 1, f(x) = x, f(x) = x^2\}$ while the outer product is defined as $\mathcal{Q}(x)\mathcal{R}(y) = \{f(x,y) = q(x)r(y) \mid q \in \mathcal{Q},\ r \in \mathcal{R}\}$. This produced the following nine bases:

$$\varphi_1(X) = 1$$
$$\varphi_2(X) = z$$
$$\varphi_3(X) = z^2$$
$$\varphi_4(X) = \theta$$
$$\varphi_5(X) = \theta z$$
$$\varphi_6(X) = \theta z^2$$
$$\varphi_7(X) = \theta^2$$
$$\varphi_8(X) = \theta^2 z$$
$$\varphi_9(X) = \theta^2 z^2$$

graphically represented in Figure 4.5-1.

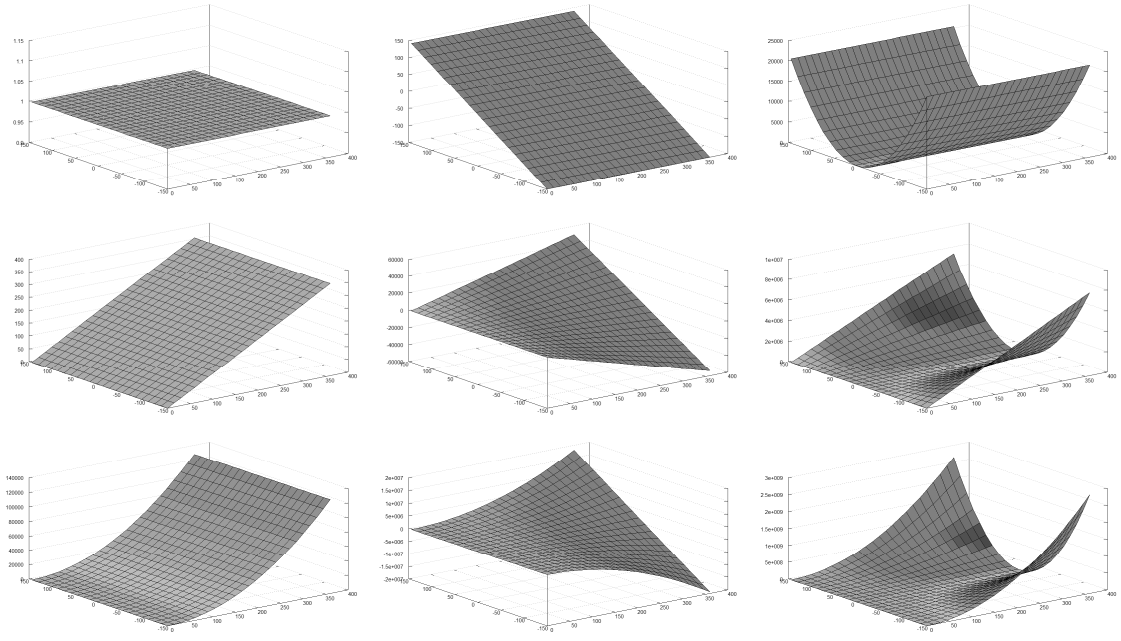

Figure 4.5-1: The nine basis graphically represented

Once we defined the basis we must define an algorithm to compute the parameters vector r. Approximation will remove the problem of having infinite states and will allow us to define the new function $\tilde{V}_{\Pi_*} \approx V_{\Pi_*}$ and to use it for computing a *near-optimal policy*.

4.5.4 Near-optimal policy

Near-optimal policy is defined using the approximated value function we derived in the previous section:

$$\tilde{\Pi}_*(\mathrm{X}) = \min_{C' \in \mathcal{C}} \left(c_t(X, X') + \mathrm{c_s}(X') + \alpha \tilde{V}_{\Pi_*}(X') \right) = \min_{C' \in \mathcal{C}} (c_t(X, X') + \mathrm{c_s}(X') + \alpha \Phi(X') r)$$

Equation 4.5-8

The last problem we have to face now is to compute vector r. To compute it we generate a set of samples for the state set $\bar{S} \subset S$ and then, for every state sample, we evaluate the value function. Once we have estimated the value function over the samples state set we can develop an algorithm that works in the same way the iterative policy evaluation algorithm used in reinforcement learning do.

We define a set $\mathcal{L} \subseteq \bar{\mathcal{S}} \times \mathcal{S}$ of state transition pairs, each one starting from a sample state and ending in the corresponding next state computed using the policy $\widetilde{\Pi}_*$ as defined in Equation 4.5-8. Therefore $\mathcal{L}$ contains an array of pairs that describes what state the controller will choose using the current policy and starting from one of the sample state, but to be able to use Equation 4.5-8 we must define a default value for vector r. When the algorithm starts we set all elements of this vector to zero. The idea is that, after every iteration, we refine vector r and we use it to re-compute state transitions inside $\mathcal{L}$. The algorithm iterates through the following:

1. Empty out $\mathcal{L}$ and then fill it again computing a new set of states starting from every sample and according to the current value function approximation

$$\mathcal{L} \leftarrow \{\,\}$$
$$\mathcal{L} \leftarrow \{(X, X') | X \in \bar{\mathcal{S}}\}$$

   where X' is the next state obtained from X computed using policy $\widetilde{\Pi}_*(\mathrm{X})$ and using the resulting clip as a parameter for the transition function so that $X' = f(X, \widetilde{\Pi}_*(\mathrm{X}))$
2. We then solve a linear program that maximizes vector r using the pairs read from $\mathcal{L}$ as constraints

$$\max_r \sum_{X \in \bar{\mathcal{S}}} \tilde{V}_{\Pi_*}(X)$$
$$\text{s.t. } \tilde{V}_{\Pi_*}(X) \leq c_t(X, X') + \mathrm{c_s}(X') + \alpha \tilde{V}_{\Pi_*}(X') \quad \forall (X, X') \in \mathcal{L}$$

   In this step we essentially inflate the value function approximation as much as possible but having it still subjected to the bounds. The inequality constraint forces the approximated value function to respect the cost system and in particular Equation 4.5-4.
3. Test if the resulting vector is not too different from the one obtained in the last iteration; if it is then exit, else go back to step one.

This algorithm can or cannot converge depending on what value the parameter α is set, depending on how the samples are taken and on the value function's shape. Anyway we discovered that trying different values for parameter α can lead to convergence; we set a default value of 0.65 for alpha and then we adjust it until the algorithm converges.

We encoded the shown algorithm inside `ClipBlender` and we used it to compute the value function for our character. Samples are taken rotating the character around its foot (foot-two) increasing θ of 90° every time until it makes a complete rotation and gets back to its original orientation; samples are taken around both left and right foot. Therefore for every clip we acquire three samples at 0°, 90°, 180° and 270°. To solve the linear program we implemented `CPlex`, inside `ClipBlender`, to take advantage of this good problem solver. An approximation of the value function is visible in Figure 4.5-2.

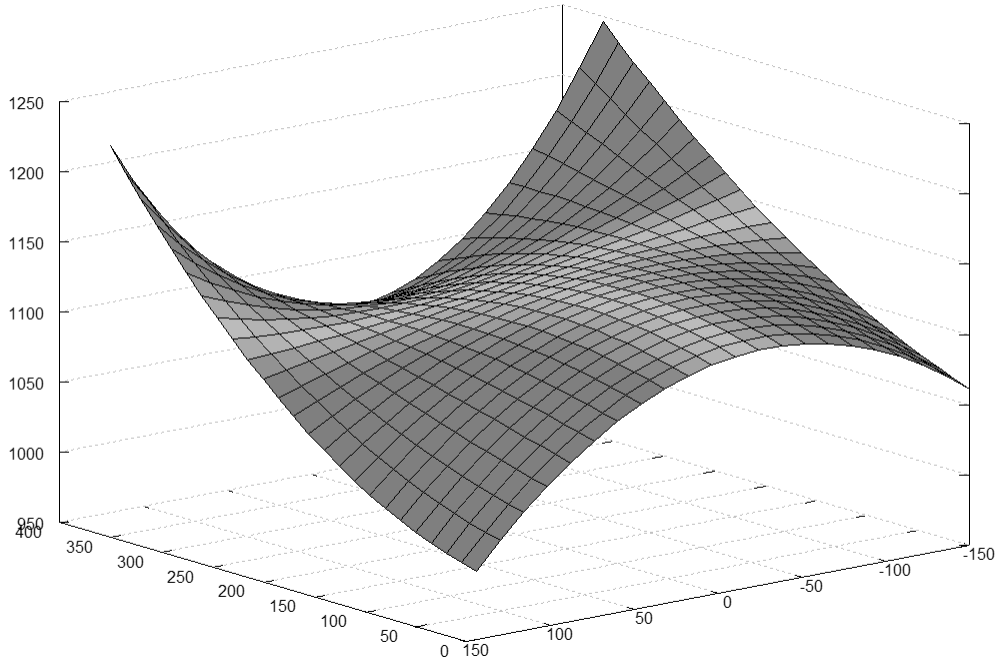

Figure 4.5-2: Approximated value function $\widetilde{V}_{\Pi_*}$

Both the presented policies allow the controller to work properly when it has to choose the next clip for the sequence. We now define how the controller works at run-time to have completely described its implementation.

4.6 Runtime control

The runtime control algorithm is fast and simple since all it has to do is to select the next clip using the enabled policy. Notice how user inputs, such as changing the motion direction, torso orientation or gait, change the state variables so that the system can automatically adjust itself to match the requirements in the next state transition. Therefore any time a clip finishes the runtime controller just invoke the policy function to compute the next state on agree with the user inputs:

$$C' = \Pi(X)$$

$$X' = f(X, C')$$

Equation 4.6-1

where Π can be Π_b or $\widetilde{\Pi}_*$.

4.7 Differences between the two policies

Since motion requires planning we expected the near-optimal policy to work better then the greedy one; instead, we discovered that both the policies were acting almost in the same way. This is probably due to the controller definition we made, and it can result different if we define some specific task, however, for the navigation task, the two policies worked almost identically.

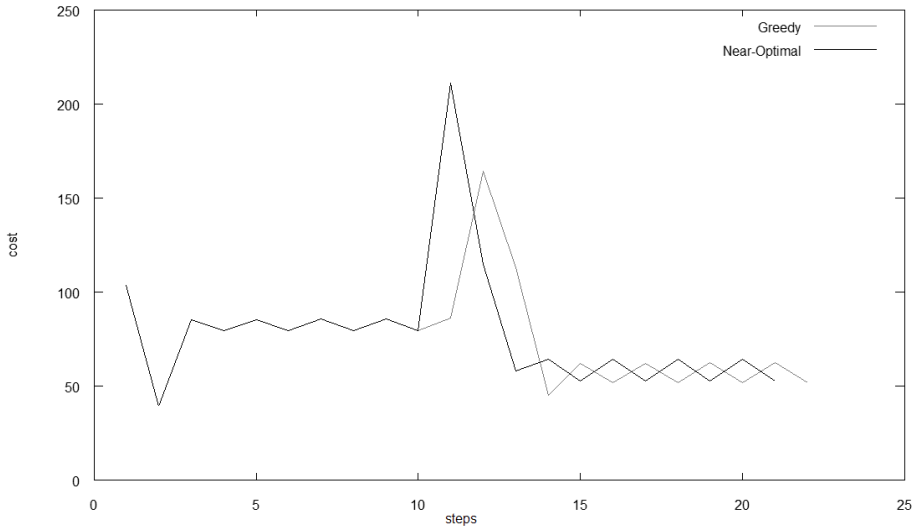

Figure 4.7-1: First test graph

To better compare the two policies we have defined several tests with two characters, one that uses the greedy policy and another one that adopts the near-optimal policy. After we have run each test we have computed a graph that shows the cost, expressed as $c_s(X) + c_t(X, X')$, paid by the system at each step.

The first test we've carried out changes the characters direction of 180° after 10 steps; the whole test last 20 seconds and then the program quits. As shown both the policies choose the same clips until the inversion of motion, then they do different choices but converge quickly to a common pattern choosing again the same clips. This result is very different from what we expected and underlines how much the greedy policy's results are good.

In the second test we let the characters move for 30 seconds and we changed their direction every 6 seconds. The direction change were random (although identical for both the characters) obtained changing the direction of motion adding to it an angle between 90° and 180°. Even in this test we can see

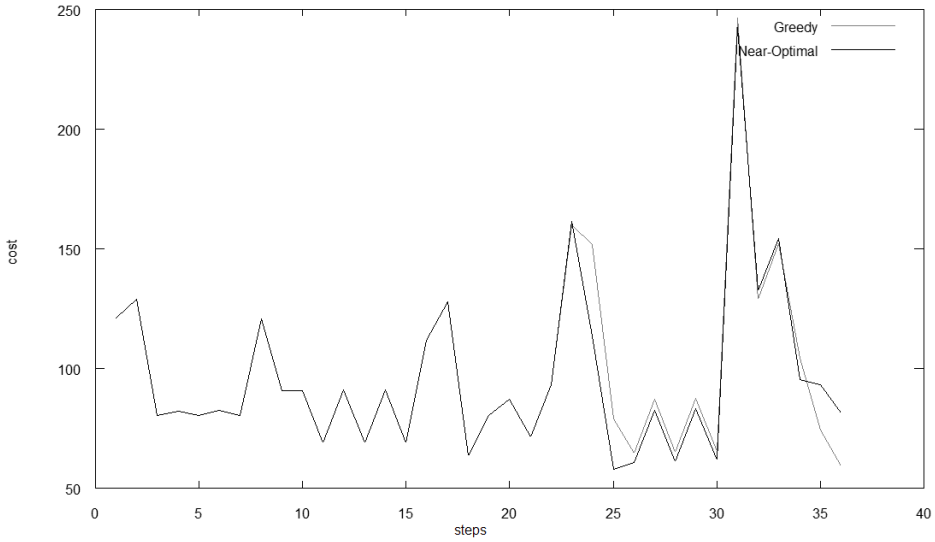

Figure 4.7-3: Second test graph

how the two policies are working similarly. Just few differences are noticeable after a turn around the 25th step.

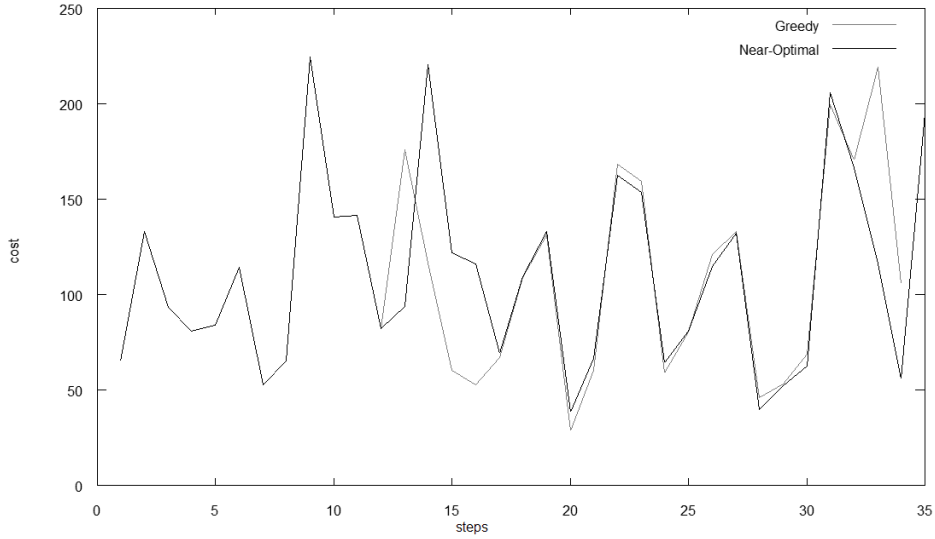

Figure 4.7-2: Third test graph

The third test stressed the movement with a continuous change to the characters direction adding to it a random angle between 150° and 220°. Even in this test results from the two policies are very similar. We then run a fourth test where we changed both the direction of motion and the torso orientation every 5 seconds for a whole of 40 seconds of test. The changes set the motion angle and the torso orientation randomly on the whole spectrum of 360 degrees. As the diagram shows the policies work

similarly, even if we notice that the near-optimal policy chooses a costly clip around steps 25th and 38th. These two high cost steps may be due to the approximations done to the value function.

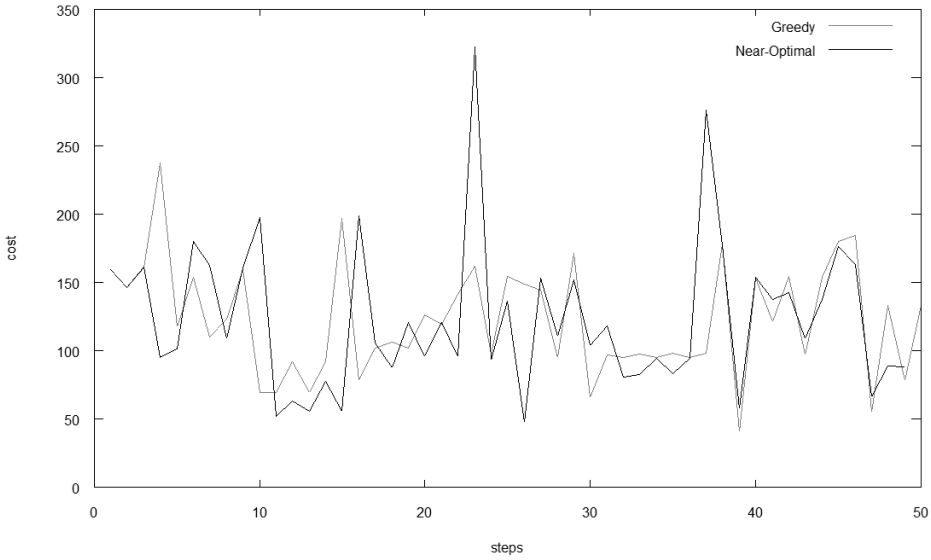

Figure 4.7-4: Fourth test graph

It would be interesting to analyze why the two policies are acting so similarly while in other works it has been proven that the greedy policy performs worse (Treuille, Lee and Popović 2007). In fact, if avoiding the training pass, required by the reinforcement learning algorithm, it would be still possible to achieve realistic results using a simpler first-step prediction, this would be a big improvement to this technique. However it is very probable that the performance of the greedy controller are related to the very definition of the coordinate system and of the controller structure, therefore, if it is so, it should be possible to identify some tasks that push the greedy policy to behave worse than the near-optimal one.

4.8 Conclusion

Now that we have completely defined the controller structure and behavior we can say to have finished our animation system. Our controller uses a cost system that is strictly correlated to the controller's task. Once we have defined the task we must introduce a policy for selecting the next clip evaluating the system's state and the various costs. We have proposed two policies, one that evaluate the cost at the very first step, and another one that take into account all the possible future steps. The latter has been approximated using basis functions. We can now analyze the results we obtained from both the policies and discuss some implementation details.

Part IV

Conclusions

5: Conclusions

5.1 Conclusions

5.1.1 Conclusion and results

Summarizing, this work presents a control system for characters animation in real-time. The system is divided into a motion model and a controller system. The motion model enables transitions from clip to clip while preventing foot-skating thanks to the definition of some constraints. The control system allows to represent the system state and to estimate its evolutions when a certain clip is added to the sequence. To choose the clip we defined a reinforcement learning paradigm with two possible choices of the next clip: a greedy and a near-optimal choice. The latter is based on an approximation of the value function through basis functions.

We led several tests using the character shown in chapter two. To achieve our results we captured about 160 motion capture clips using 17 markers. Every clip has been partitioned on the base of its gait in one of the three possible set, *stand*, *walk* and *quick walk*. Unfortunately there were no room in our laboratory for capturing a running gait, therefore we limited ourselves with the quick walk gait. In the final version of our program we discarded some of the clips because they were never selected from the algorithm or because the gait was not recognizable as a walk nor as a quick walk, or other similar problems. If we had had more time it would have been better to re-acquire those clips to have a bigger sample state space.

The last configuration we adopted used a subset of about 80 clips from the 160 we acquired; this sped up most of the computations still permitting us to achieve a good visual result (notice that clips are mirrored after being loaded, therefore the 80 clips mentioned are only those that starts with the left foot; when loaded they are mirrored reaching a grand total of 160 clips).

At runtime the controller chooses between clips very quickly; we measured the time it takes to query the policy and we have discovered that it takes about 30 milliseconds using any of the two policies. Learning the near-optimal policy, instead, is not that quickly since it can require from 1 up to 5 minutes and it can miss to converge; when this happens we have to retrain the system trying a different alpha value. In our last configuration we had alpha set at 0.55 and the approximation function converged in a about minute.

All the procedures for setting the constraints, as well as the procedure to compute the approximated value function, are embedded inside the `ClipBlender` tool. When loading our file the tool takes about 14 seconds to load everything up allocating ~170Mb of memory. The main animation program instead loads the system in less than 10 seconds, it requires ~250Mb of memory and plays ~180 frame per second (including rendering and animation algorithms).

Our test has been done on an Intel Core2 Quad CPU Q6600 at 2.40Ghz having 2.00 GB of RAM and using an NVIDIA GeForce GTS 250 (1.00 GB of RAM onboard) as video card.

In spite of the power of dynamic programming, continuous and high dimensionality controllers are challenging. A way to deal with it is to use *switch* the value function at runtime. This technique exploit the idea that some state variables remain constant during the clip transition, for example the gait does not change without an explicit user command. Since the controller cannot affect the gait we do not need to take it into account during the estimation of the value function.

We can then switch between two different value functions computed using two different gaits. For example, when transitioning from a walking gait to a quick-walking gait, we just change the value function used for evaluating the policy. This avoid us to put the gait variable inside the value function, speeding up the evaluation process. With this exploit we don't resolve the curse of dimensionality problem inherent in reinforcement learning, but we just reduce the amount of computation that the algorithm must perform.

5.1.2 Considerations and Future works

This animation synthesis technique has proved to be powerful in generating realistic walk animations, but it is not free from problems. The worst issue in this technique is related to the large amount of motion capture clips needed when several tasks with their own controller are defined. For example if we would allow the user to control the head orientation too, we should capture other motion capture clips with all the possible combination of head/torso/motion-direction. This will lead to thousands of captures, and obviously it is a big problem to deal with. Moreover controllers' dimensionality would increase very quickly leading to long learning time for the near-optimal value function.

A Solution may be to compute the additional clips via a digital synthesis process for example. A related problem is that transitions are limited by the number of different states which makes it harder to achieve a fine control such as stepping exactly to a given position. We should have a clip for any possible step in any possible position to achieve such a result, but such a quantity tends to infinite. Again a solution may be to use clips synthesis, for example via blending, to compute the missing steps in our database.

Another issue is related to character's reaction time. Even if our model has a high branching factor, which permits to choose any clip from the database, we can have several situations where well-timed movements lead to slow responses. The worst scenario is if user requires a 180° turn to the character as soon as the system has changed the clip; this means that the user has to wait two steps before the

character effectively turns to the desired direction. It would be really interesting to analyze a system that allows the character to react at the first next step instead that to the third next step.

APPENDIX A

A: Mathematical notations

A.1 Matrices

A matrix is a rectangular array of numbers with a given number of rows and columns. We usually say that a matrix is$m \times n$, meaning it has m rows and n columns and we refer to this size as the dimension of the matrix. In this thesis matrices are written as follows:

$$\mathbf{M} = \begin{pmatrix} a_{00} & a_{01} & \dots & a_{0n} \\ a_{10} & a_{11} & \dots & a_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m0} & a_{m1} & \dots & a_{mn} \end{pmatrix}$$

Every entry of the matrix$a_{\mathrm{ij}} \in \mathbb{R}$, the first index is used for rows and the second one is for columns. All the matrices used have dimension 4x4 and are said to be in$\mathrm{M}(4,4,\mathbb{R})$; every matrix is stored in row-major order.

The main reason to use 4x4 matrices is to represent object rotations, scales and translations in a three-dimensional space. For example, every bone may be totally represented by its roto-translation matrix$\mathbf{B} \in \mathrm{M}(4,4,\mathbb{R})$.

Multiplying matrices allows describing a sequence of rotations/scales/translations (or a mix of them).

A.1.1 Rotation matrices

In linear algebra, a rotation matrix is any matrix that acts as a rotation in Euclidean space. Rotation matrices are always square, and we assume them to have real entries, though the definition makes sense for other scalar fields. There are three basic rotation matrices in three dimensions and we will refer to them using the compact notation of functions. The following matrices represent counterclockwise

rotations of an object relative to fixed coordinate axes, by an angle of θ. The direction of the rotation is as follows: R_x rotates the *y*-axis towards the *z*-axis, R_y rotates the *z*-axis towards the *x*-axis, and R_z rotates the *x*-axis towards the *y*-axis:

$$R_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A.1.2 Matrix-to-Matrix multiplication

Since we are using only matrices in $\mathrm{M}(4,4,\mathbb{R})$ the matrix by matrix multiplication is always defined as follows:

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} \text{ or } \mathbf{C} = \mathbf{AB}$$

$$\mathrm{c}_{\boldsymbol{ij}} = \sum_{\boldsymbol{r=0}}^{\boldsymbol{3}} a_{ir} b_{rj} \quad \forall i, j: 0 \leq i \leq 3 \ \wedge 0 \leq j \leq 3$$

Since matrix multiplication is not commutative the operands order is relevant; in this work transformations are read from left to right, so a rotation followed by a translation is represented as:

$$\mathbf{M} = \mathbf{RT}$$

A.1.3 Vector-to-Matrix multiplication

To transform a given vector p against the desired transformation matrix a vector to matrix multiplication is performed:

$$p' = p\,\mathbf{M}$$

$$p'_i = \sum_{r=0}^{3} p_r \mathbf{M}_{ri} \quad \forall\, i: 0 \leq i \leq 3$$

where p is the original point, $\mathbf{M}$ it the transformation matrix and p' the transformed point. The point p here is represented using homogeneous coordinates so it is a vector $\in \mathbb{R}^4$ where the fourth component (w) is the homogeneous component.

$$p = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Any vector in this thesis, if not differently specified, has to be considered in the homogeneous coordinate system. The dot notation for vectors is used to specify a component from it, so the x value from vector p can be specified using $p.x$.

A.2 Other notations

A.2.1 Clamp function

Let define a clamp function for general purposes:

$$clamp_{(a,b)}(x) = \begin{cases} a, & x \leq a \\ b, & x \geq b \\ x, & otherwise \end{cases}$$

A.2.2 Normal vector

This function requires to be feed with an orthonormal matrix and returns the normal vector extracted from it.

$$normal(\mathbf{T}) = \begin{pmatrix} \mathbf{T}_{00} \\ \mathbf{T}_{01} \\ \mathbf{T}_{02} \\ 0 \end{pmatrix}$$

A.2.3 Translation matrix on plane XZ

This function read the position from the transformation matrix provided as input and returns a new matrix with the sole translation on plane XZ.

$$positionXZ(\mathbf{T}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \mathbf{T}_{30} & 0 & \mathbf{T}_{32} & 1 \end{pmatrix}$$

A.2.4 Angle between vectors

This function gets two vectors ($\mathbb{R}^4$) and returns the angle θ between them.

$$angle(v, v') = \cos^{-1}(v \cdot v')$$

APPENDIX B

B: Reinforcement Learning

B.1 Agents and Environment

In reinforcement learning the decision-maker is called *agent* while any other thing it interacts with is part of the *environment* (Sutton and Barto 1998). The agent and the environment interact continually, the former selects and performs an action and the latter responds to those actions presenting new situation to the agent. In reinforcement learning we want the environment to provide rewards to the agent to represent how good the agent is performing with his actions. These interactions are portrayed in figure B.1-1.

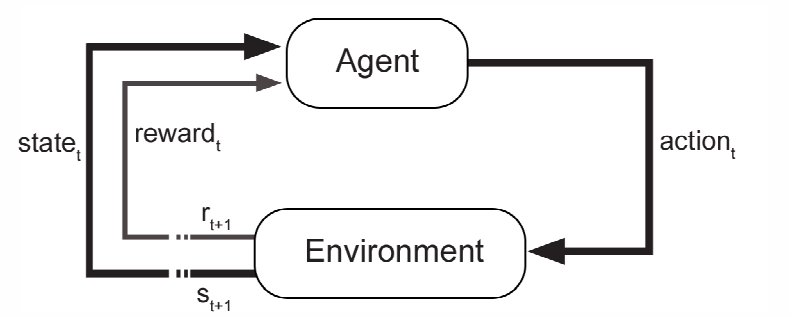

Figure B.1-1: Agent-environment diagram

At every time step the agent chooses a new action using a function called agent's *policy*, denoted where is the probability that if . Reinforcement learning specifies how the agent changes its policy as a result of its experience in the environment; the agent's goal is to maximize the total amount of reward it receives from the environment over the long run.

B.1.1 Reinforcement through rewards

A reward is a signal passed from the environment to the agent formalized as a number $r_t \in \mathbb{R}$. Informally, the agent's goal is to maximize the total amount of the reward, which means maximizing not the immediate reward but cumulative reward in the long run. The expected return is then defined as some specific function of the reward sequence. Depending if the agent-environment interaction does or does not break naturally into finite time steps or just goes on continually without any limit, some additional concepts must be introduced. If the final step $T = \infty$, meaning no limits is set, the additional concept of *discounting* is needed. Roughly speaking, at every step the reward is ever less precious for the agent to collect. The expected discounted return function is defined as follows:

$$\boldsymbol{R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1}}$$

Equation B.1-1

where γ is a parameter $0 \leq \gamma \leq 1$ called *discount rate*.

B.1.2 Value functions

Almost all reinforcement learning algorithms are based on estimating *value functions.* A value function estimates how good a state is for the agent to stay in or how good it is to perform an action in the given state.

$$V^{\Pi}(s) = E_{\Pi}\{R_t | s_t = s\} = E_{\Pi} = \left\{ \sum_{k=0}^{\infty} \gamma^k \, r_{t+k+1} \,\middle|\, s_t = s, a_t = a \right\}$$

Equation B.1-2

where E_{Π} denotes the expected value if agent is following policy Π and t is any time step.

In reinforcement learning it is usual to define an *action-value function* besides the specified *state-value function*, but in this thesis we assume actions having no costs. Value functions can be estimated from experience keeping averages, for each state encountered, of the actual returns of the state; the average will converge to the state's value $V^{\Pi}(s)$ as the number of state is encountered infinity times. Since value functions satisfy a particular recursive relationship (Sutton and Barto 1998) the following equation can be derived:

$$V^{\Pi}(s) = \sum_{a} \Pi(s,a) \sum_{s'} \mathcal{P}^{a}_{ss'} \left(\mathcal{R}^{a}_{ss'} + \gamma V^{\Pi}(s') \right)$$

Equation B.1-3

given that in, this thesis, we assume $\mathcal{P}_{ss'}^{a} = 1 \; \forall \, a \in \mathcal{A}(s), s \in \mathcal{S}, s' \in \mathcal{S}$, equation B.1-1 becomes

$$V^{\Pi}(s) = \sum_{a} \Pi(s,a) \sum_{s'} \left(\mathcal{R}_{ss'}^{a} + \gamma V^{\Pi}(s') \right)$$

Equation B.1-4

where $\mathcal{R}_{ss'}^{a}$ is the expected immediate reward on transition from s to s' under action a.

Equation B.1-3 is the *Bellman equation* for V^{Π}; it expresses a relationship between the value of a state and the values of its successor states. The value function V^{Π} is the unique solution to its Bellman equation.

Reinforcement learning methods relies on update or *backup* operations to transfer value information from a state back to its parent.

B.1.3 Optimal value functions

A policy Π is said to be better then another policy Π' if its expected return is greater than that of Π' for all the states formalized as $\Pi > \Pi'$ if and only if $V^{\Pi}(s) > V^{\Pi'}(s) \; \forall \, s \in \mathcal{S}$. The policy which is better than or equal to all other policies is called the *optimal policy* and it's denoted as Π^{*} while its state-value function is called *optimal state-value function* denoted as V^{*}.

$$\mathrm{V}^{*}(s) = \max_{\Pi} V^{\Pi}(s) \; \forall \mathrm{s} \in \mathcal{S}$$

Equation B.1-5

Because V^{*} is the value function for a policy it must satisfy the self-consistency given in Equation B.1-4 but since V^{*} is the optimal value function, its consistency condition can be written without reference to any specific policy.

$$\mathrm{V}^{*}(s) = \max_{\mathrm{a} \in \mathcal{A}(\mathrm{s})} \sum_{s'} \left(\mathcal{R}_{ss'}^{a} + \gamma V^{*}(s') \right)$$

Equation B.1-6

Equation B.1-6 is the Bellman optimality equation where $\mathcal{P}_{ss'}^{a} = 1 \; \forall \, a \in \mathcal{A}(s), s \in \mathcal{S}, s' \in \mathcal{S}$. This equation is actually a system of equations, one per state, so that for N states there are N equations and N unknowns. Having V^{*} it is possible to determine an optimal policy.

Since in our work actions lead to one and only one state Equation B.1-6 can be written as:

$$\mathrm{V}^{*}(s) = \max_{\mathrm{a} \in \mathcal{A}(\mathrm{s})} \mathcal{R}_{ss'}^{a} + \gamma V^{*}(s')$$

Equation B.1-7

BIBLIOGRAPHY

Bertsekas, Dimitri P. *Dynamic Programming and Optimal Control, vol 2.* Athena Scientific, 2001.

Charles, Rose, Brian Guenter, Bobby Bodenheimer, and Michael F Cohen. "Efficient generation of motion transitions using spacetime constraints." *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996: 147 - 154.

Choi, Kwang-Jin, Sang-Hyun Park, and Hyeong-Seok Ko. "Processing Motion Capture Data to Achieve Positional Accuracy." *Graphical Models and Image*, September 1999: 61(5):260-273.

Corazza, S., L. Muendermann, A. Chaudhari, T. Demattio, C. Cobelli, and T. Andriacchi. "A Markerless Motion Capture System to Study Musculoskeletal." *Annals of Biomedical Engineering*, June 2006: Vol. 34, No. 6.

de Farias, Daniela Pucci, and Van Benjamin Roy. "The Linear Programming Approach to Approximate Dynamic Programming." *Operations Research, Volume 51*, 2003: 850 - 865 .

Dyer, Scott, Jeff Martin, and John Zulauf. "Motion Capture White Paper." 1995.

Faloutsos, Petros, Michiel van de Panne, and Demetri Terzopoulos. "The virtual stuntman: dynamic characters with a repertoire of autonomous motor skills." *Computers and Graphics, 25 (6)*, 2001: 933-953.

Gleicher, Michael. "Retargetting motion to new characters." *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, July 1998: 33–42.

Kavan, Ladislav, Steven Collins, Jiri Zara, and Carol O'Sullivan. "Geometric Skinning with Approximate Dual Quaternion Blending." *ACM Transaction on Graphics*, 2008: 27(4).

Kovar, Lucas, and Michael Gleicher. "Flexible automatic motion blending with registration curves." *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, 2003: 214 - 224.

Kovar, Lucas, Michael Gleicher, and Frédéric Phigin. "Motion graphs." *ACM Transactions on Graphics (21, 3)*, July 2002: 473–482.

Kovar, Lucas, Michael Gleicher, Hyun Joon Shin, and Andrew Jepsen. "Snap-together motion: Assembling run-time Animations." *ACM Transactions on Graphics 22,3*, July 2003: 702-702.

Kwon, Taeso, and Sung Yong Shin. "Motion Modeling for On-Line Locomotion Synthesis." *Eurographics/ACM SIGGRAPH Symposium on Computer Animation*, 2005: 29–38.

Lau, Manfred, and James Kuffner. "Precomputed Search Trees: Planning for interactive goal-driven animation." *ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, 2006: 299-308.

Lee, Kang Hoon, Myung Geol Choi, and Jehee Lee. "Motion patches: building blocks for virtual environments annotated with motion data." *ACM Transactions on Graphics 25(3)*, July 2006: 898-906.

Naugle, Lisa Marie. "Motion Capture: Re-collecting the Dance." *International Council of Kinetography Laban*, July 1990.

Parent, Rick. *Computer Animation: Algorithms and Techniques.* Morgan Kaufmann; 1st edition, 2001.

Park, S. I., H. J. Shin, and S. Y. Shin. "On-line locomotion generation based on motion blending." *Proceedings of ACM SIGGRAPH/Eurographics Symposium on Computer Animation 2002*, July 2002.

Perlin, Ken. "Real time responsive animation with personality." *IEEE Transactions on Visualization and Computer Graphics*, March 1995: 1(1):5–15.

Pozzo, Thierry, Alain Berthoz, and Loïc Lefort. "Head Kinematics during Complex Movements." In *The Head-Neck Sensory Motor System*, 587-590. Oxford University Press, 1992.

Raymond, Eric. *LERP*. 2003. http://www.catb.org/jargon/html/L/LERP.html (accessed 02 15, 2010).

Rose, Charles F., Peter-Pike J. Sloan, and Michael F. Cohen. "Artist-Directed Inverse-Kinematics Using Radial Basis Function Interpolation." *Computer Graphics Forum*, 2001: 20(3).

Rose, Charles, Michael F. Cohen, and Bobby Bodenheimer. "Verbs and adverbs: Multidimensional motion interpolation." *IEEE Computer Graphics & Applications*, October 1998: 18(5).

Sutton, Richard S., and Andrew G. Barto. *Reinforcement Learning: An Introduction.* Cambridge: MIT Press, 1998.

Treuille, Adrien, Yongjoon Lee, and Zoran Popović. "Near-optimal character animation with continuous control." *ACM Transactions on Graphics (TOG)*, July 2007: 26(3).

Wen, Gaojin, Zhaoqi Wang, Shihong Xia, and Dengming Zhu. "From Motion Capture Data to Character Animation." *Proceedings of the ACM symposium on Virtual reality software and technology*, 2006: 165 - 168 .

Wiley, Douglas J., and James K. Hahn. "Interpolation synthesis of articulated figure motion." *IEEE Computer Graphics and Applications*, November/December 1997: 39–45.

Witkin, Andrew, and Zoran Popović. "Motion warping." *SIGGRAPH 95 Conference Proceedings, Annual Conference*, August 1995: 105-108.

Zhao, Jianmin, and Norman I. Badler. "Real Time Inverse Kinematics with Joint Limits and Spatial Constraints." January 9, 1989.

Zordan, Victor B., and Nicholas C. Van Der Horst. "Mapping optical motion capture data to skeletal motion using a physical model." *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.