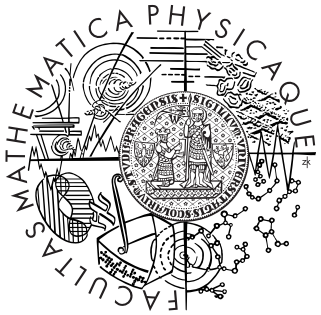
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Ladislav Kavan

Simulation of Fencing in Virtual Reality

Katedra software a výuky informatiky
Vedoucí diplomové práce: **Doc. Ing. Jiří Žára, CSc.**
Studijní program: **Informatika**
Studijní obor: **Počítačová grafika**

Poděkování

Na tomto místě bych chtěl poděkovat zejména svému vedoucímu diplomové práce Doc. Ing. Jiřímu Žárovi, CSc. za jeho takřka otcovskou podporu při realizaci vytyčených cílů a za řadu cenných rad a podnětů pro zdokonalování práce.

Dále bych chtěl poděkovat RNDr. Josefu Pelikánovi, CSc. za přínosné konzultace. Děkuji rovněž mému bratrovi Mojmíru Kavanovi za pomoc při testování aplikace a mým rodičům za podporu při studiu.

V neposlední řadě děkuji též všem učitelům, kteří mě seznámili s mnoha různými podobami šermu: Karlu Anderlemu, Janu Gruberovi, Michalu Vyhnálkovi a Jindřichu Ziegelheimovi.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne June 3, 2003 Ladislav Kavan

Contents

Název práce: Simulace šermu ve virtuálním prostředí
Autor: Ladislav Kavan
Katedra: Katedra software a výuky informatiky
Vedoucí diplomové práce: Doc. Ing. Jiří Žára, CSc.
e-mail vedoucího: zara@fel.cvut.cz
Abstrakt: V nedávné době se objevily pokusy o počítačovou simulaci tenisu. Tato práce pokračuje ve studiu možností virtuální reality pro simulaci sportovní činnosti, konkrétně šermu. Realistická simulace netriviální, soutěživé lidské činnosti vyžaduje spolupráci mnoha technologií z různých oborů. Jmenovitě se jedná o problématiku animace virtuálních humanoidů, inverzní kinematiky, detekce kolizí a reakce na ně. Přínos této práce spočívá 1) ve vylepšeních některých současných algoritmů a 2) v sestavení aplikace, která umožňuje simulaci šermu pro dva uživatele. Tato aplikace nevyžaduje speciální hardware, a tudíž je otevřena pro širokou veřejnost.
Klíčová slova: simulace, šerm, virtuální humanoidi, virtuální realita

Title: Simulation of Fencing in Virtual Reality
Author: Ladislav Kavan
Department: Katedra software a výuky informatiky
Supervisor: Doc. Ing. Jiří Žára, CSc.
Supervisor's e-mail address: zara@fel.cvut.cz
Abstract: Recently, some effort has been done in computer simulation of tennis. This work continues with studying the possibilities of virtual reality for simulation of sports, especially fencing. The realistic simulation of non-trivial, competitive human activity involves cooperation of many technologies from different areas. Namely, we deal with problematics of realistic skin deformation, inverse kinematics, collision detection and response. The contributions of this work are 1) some improvements of contemporary algorithms, and 2) an application, that enables simulation of fencing for two users. The application runs on an average hardware, thus broad public can make use of it.
Keywords: simulation, fencing, virtual humanoids, virtual reality

Chapter 1

Introduction

One of the contributions of this thesis is to put together (compile) different technologies used in contemporary computer graphics, as well as in other areas. Therefore, we divide the thesis into several sections, each describing a particular set of problems. In the beginning of each section we would like to briefly summarize the related work and highlight our contribution in the area (if any). This section addresses some technical issues and explains why have we chosen the topic of virtual fencing.

1.1 Background and Motivation

This work has been inspired by the recent activity of VRlab from EPFL [29] and CGG at FEL ČVUT [32] concerning virtual reality simulation of tennis. VRlab has realized a project of virtual tennis using very advanced (and expensive) motion tracking system: magnetic motion capture. Such hardware converts the human motion into a computer understandable data in real-time. With combination of head mounted display to view the virtual environment, it enables quite high level of immersion.

Unfortunately, these advanced resources were not accessible in neither of our schools of computer science. We have therefore come to a decision to use only a low-level hardware for our simulation of fencing. It has also a great advantage: the resulting application can be run by a lot of users – its hardware demands are not higher that those of an average 3D computer game. Some aspects of virtual reality teaching of tennis using a low-level hardware were studied in the diploma thesis by Ing. Vladimír Štěpán [33], but the complete framework of tennis simulation has not been designed there.

Why have we chosen fencing instead of tennis? One reason is that the author is familiar with it more than with any other sport. Another is that

the virtual reality offers absolute safety even in situations that would be dangerous in real world fencing. The risk of injury is certainly higher than in tennis, and this adds a reason for virtual reality simulation of fencing. Another fact is, that we believe that the simulation of fencing is more difficult than the simulation of tennis in certain aspects. Of course, both fencing and tennis have many features in common. Novertheless in fencing, the players contact is much closer. The weapon interacts instantly and continuously with the opponent's weapon, instead of indirect "communication" via a ball. This is a big difference indeed, because it means that the fencer does not control his weapon exclusively, as does a tennis player – the fencer's weapon can be deflected or pushed away by the opponent's weapon. So another purpose of this work is to examine the possibilities of contemporary technologies for simulation of non-trivial virtual user interaction. Exaggerating a little, we can say that fencing simulation is another step in virtual reality simulation of sports.

1.2 Concerning Fencing

There are many styles of fencing in the modern age, and even much more of them existed in history. We faced the question which one should be chosen for our simulation. Since this is neither a history nor a physical education oriented work, we mention only the most popular fencing schools in a brief, just in order to select one of them, that is the most amenable for virtual reality simulation. If you are interested in the history of European fencing, consult a book [30].

The history of European fencing dates from the ancient Rome. It is believed the fencing skills were highly developed in this era, considering the quality of the weapons and the armor. However there are no preserved documents concerning the antique fencing itself. Probably the oldest manuscript about fencing was written in Germany by Hans Talhoffer in 1442. This is the bible of one branch of the contemporary historical fencing, called a German school. In this school a raw force, as well as a heavy sword are emphasized. However there is also a collection of more sophisticated techniques in Talhoffer, somewhat similar to judo.

Up to approximately the 15th century, the king of the weapons was a sword, usually for one hand. With the fall of chivalry (tied to the boom in gunpowder), new weapons were developed to satisfy current needs. The swords transformed reluctantly into lighter or longer weapons: rapiers, two handed swords or swords for one and half hand (sometimes held in one hand, sometimes in both). The Italians mastered such weapons first and their

schools were appreciated soon in the whole Europe. Books about Italian fencing from famous masters, such as Achilles Marozzo, Camillo Agrippa and Giacomo Grassi are available today. They form another branch of modern historical fencing known as the Italian school.

The evolution of the fencing led to lighter weapons with focus on the point, instead of the blade: an epee was created. Besides Italy, it became very popular also in French, where the training version of epee was discovered: a foil. We are getting to the modern fencing, which has rules ensuring a safety sport.

The modern fencing differs three weapons: an epee, a foil and a sabre. They are very similar to one another – very fast and easily bendable. The techniques often involve only a small movement with the tip of the weapon, the strength of the fencers plays a little part. Modern fencing emphasizes good reactions and perception. Often a duel is so fast that an untrained eye can not see which fencer was hit.

Besides Europe, the fencing evolved independently in other parts of the world. Famous is the Japanese fencing, known as Kendo (*ken* is the Japanese word for sword). In Kendo, a big attention is paid to the discipline and mental training. In Japanese mentality is more important to keep honor than to survive. To be successful in Kendo, it is not sufficient to be able to hit the opponent, as in the European fencing; the attacker must also exhibit an unfailing, robust will – a fighting spirit.

So, let us return to the question asked in the beginning of this section: which style should be chosen for virtual reality simulation of fencing? We must account for the limited possibilites of input devices (keyboard, mouse and perhaps joystick) and for the latency of the simulation engine. The Kendo is, in our opinion, impossible to simulate without simulation of the human thinking[1]. The modern fencing weapons – epee, foil and sabre are also not very suitable, because of their speed and necessity of precise control.

Due to these reasons, we have chosen to simulate fencing with a one-hand sword, similar to an Italian fencing school. This school defines the eight basic parry positions (and several advanced, which are not supported by the application) according to Fig. 1.1. The cuts are derived from the corresponding blocks. In fact there are only seven cuts, since the head cut can be parried by either quinte or sixte. An important features of a correct cut are

- large swing, that enables the weapon to gain speed

[1]When a Japanese sensei was asked about a virtual reality simulation of Kendo, he laughed...

- movement of the weapon in one plane – the fluctuations waste the energy
- the hit performed by the last third of the blade. Closer to the guard is smaller actual velocity, which means a weak strike. Close to the point is higher actual velocity, but the cut is shallow, resulting only in flesh wound. The last third of the blade is the best trade-off.

These conditions are enforced by the application to some extent.

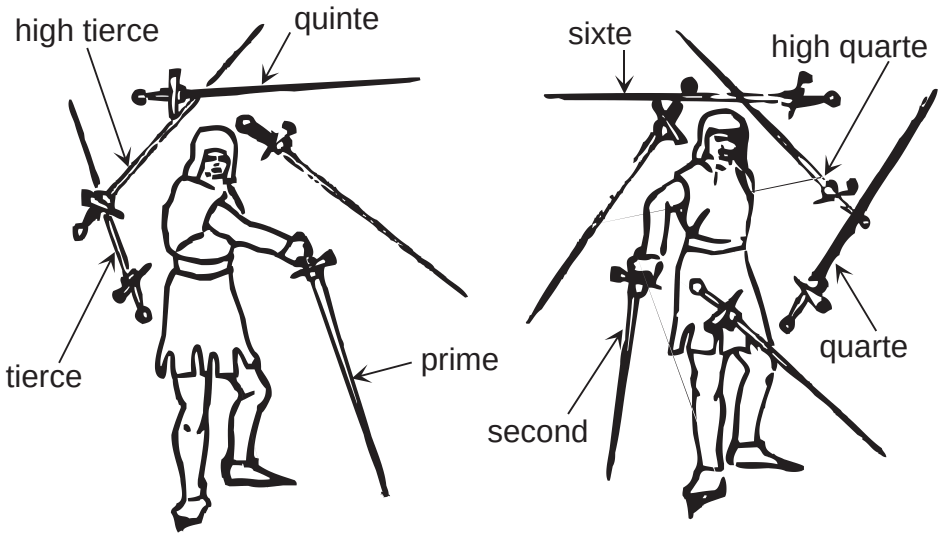


Figure 1.1: Two series of parry positions (from [30])

In former versions of the application, a hit by the point was also supported, but it was canceled, because it was too strong[2].

Concerning the footwork, the nowadays interpretation of the Italian fencing system is quite similar to the modern fencing. The fight is carried on a straight line, which is the most effective possibility to control the distance from the opponent. The application supports four basic leg routines: entering the guard position (which must be taken before any other action), step forward, step backward and lunge. The guard position with sabre, which is quite similar to that one with sword is in Fig. 1.2.

A lot of fencing styles define only a small area on the opponent's body, where the hit is legal. This is partially due to the safety reasons and partially

[2]One of the reasons why it is so hard to parry fast attacks (such as the hit by the point) in virtual reality is, that the virtual characters do not radiate any lateral information, such as some face-play and gestures.

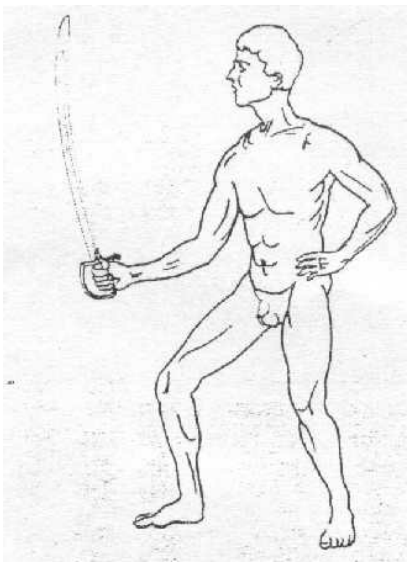


Figure 1.2: Basic guard with a sabre (from [40])

to emphasize the severe hits over non-serious ones. In virtual reality simulation we take advantage of the fact, that we can safely hit anywhere on the opponent's body. Thus the hit area is the whole surface of the body except the hand holding the sword, and its forearm. This exception was introduced to make the duel more interesting – it is quite easy to hit the hand, which degrades the fencing style.

1.3 Application Design

One of the goals of this work is to develop an application that enables the users to fight each other in a virtual environment. We do not consider a fencing with more than one opponent (this could be hardly called "fencing", perhaps fight is more appropriate). The system can be used for example by the instructor of fencing to demonstrate the typical mistakes of a real fencer. Another usage is for organization of a virtual match.

Being limited only to a low-end computer peripherals, we decided to divide the control of a virtual player (*avatar*) to two independent parts: the avatar legs and the avatar's armed hand (holding a sword). The legs movement is commanded by the keyboard, which runs the recorded sequences of basic steps. The hand, as well as the weapon, is controlled by some type of 2D input device: a mouse or joystick. The conversion from the 2DOFs

(Degrees of Freedom) space of the input device to the 6DOFs of the position and orientation of the hand is non-trivial. It is discussed in section 4.2.1.

To the 2D input devices is connected also another problem: typical computer has only one mouse and no joystick at all, i.e. only one 2D input device, while the players are two. In order to do not require a joystick in a compulsory way, we have implemented a network support. In this mode the players use more than one computer. However, the joystick is still recommended due to the following reasons

- the input area of a joystick is limited, which corresponds to a limited space of a hand reach (the mouse movement is limited only by the length of the cable)
- the joystick supports auto-centering, which corresponds to a basic position of the weapon (pointing to opponent)
- some better joysticks support **force-feedback**, which is exploited by the application to simulate the strikes of the weapon
- in network game the users must account for network latency, which can be disturbing during fast fencing

Nevertheless, the network connection has also some advantages, for example the possibility of spectators. They can view the match on their computer, although they can not interfere with it.

The hits are evaluated straight-forwardly using a collision detection mechanism, with the exception of the arm holding the weapon, as described in section 1.2.

A left-handed fencer is supported by a simple trick: reflexion defined by an appropriate plane projects right-handed fencer to the left-handed one. However, this option was never necessary in practice.

The application runs on a Win32 platform with OpenGL drivers for the visualization. Due to the force-feedback joystick support, it is also necessary to have a DirectX library installed, version 7 or later. The hardware requests (processor, graphics accelerator) are similar as for an average 3D computer game, for example Pentium III processor and GeForce2 graphics accelerator is a sufficient configuration.

1.4 Used Software and Libraries

The application was developed using Microsoft Visual C++ 6.0 with OpenGL interface for 3D graphics. For the virtual humanoid animation were used

two shareware programs, MilkShape3D 1.5.6 (Chumbalum Soft) and CharacterFX 1.2 (Insane Software). Each one offers only a limited set of functions, but they complement each other. From the CharacterFX was also taken a virtual humanoid, a kung-fu style fighter called Yopago. Originally, we intended to comply a H-Anim standard for humanoid animation, but this idea has been abandoned because of problems with virtual humanoid and animation software acquisition. To find a virtual humanoid model and an appropriate animation software for free is not an easy task....

To include the support of MilkShape3D file format in our application, we used some routines from the library PortaLib3D [31]. For inverse kinematics we integrated the IKAN system, developed at the University of Pennsylvania [34]. The Microsoft DirectX 7.0 libraries provide an interface for joystick, together with support of force-feedback.

1.5 Notation and Conventions

The number sets are denoted by Z for integer numbers, R for real numbers and C for complex numbers. We will work with vectors from generally n-dimensional Euclidean space, which will be denoted as R^n. Unless said otherwise, we will work in R^3. The standard basis vectors of R^3 are $e_0 = (1,0,0)^T, e_1 = (0,1,0)^T, e_2 = (0,0,1)^T$, sometimes we use shortly x, y, z instead of e_0, e_1, e_2.

The vectors are by default considered column, and they are multiplied with the matrices from the right, for example in R^3 the formula $Mx = y$ should be interpreted as

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}$$

The coordinate system in R^3 is assumed to be right-handed, i.e. when looking from the positive direction of the z-axis, the rotation from the x-axis to the y-axis runs counter-clockwise, see Fig. 1.3. This corresponds with the 3D rotation convention: a rotation given by an axis $a \in R^3$ and an angle α means a counter-clockwise rotation of angle α when looking from the positive direction of axis a.

Any vector from R^3 can be expressed in *homogeneous coordinates* as R^4 vector. These vectors are related by

$$(x, y, z)^T \equiv (xh, yh, zh, h)^T$$

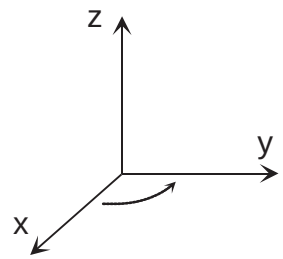


Figure 1.3: Right-handed coordinate system

The transformation in homogeneous coordinates can be expressed by *homogenous matrix*, which has by convention following structure

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & t_0 \\ m_{10} & m_{11} & m_{12} & t_1 \\ m_{20} & m_{21} & m_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} M & t \\ 0 & 1 \end{pmatrix}$$

where the matrix M is the original R^3 affine transformation and the vector t is a translation.

If the matrix has to be stored in an array, the column-major convention is used, i.e. the elements are stored in order $m_{00}, m_{10}, m_{20}, 0, m_{01}, \ldots$. This is the same convention as used by OpenGL.

The identity matrix will be denoted by I, and the transpose of a matrix by superscript ^{T}. The dot product of two vectors a, b will be written just as ab, and the cross product as $a \times b$. The numbering starts by default from zero, also when referring to elements of a matrix, as in the case of matrix M above.

Chapter 2

Rotation Representation

Intuitively, the 3D rotations are important for the virtual humanoid animation, because a joint movement can be nicely approximated by a rotation. However, the rotations are not a trivial algebraic object, mainly because they are not commutative. There are used several different representations of rotations in computer graphics, because each of them has its advantages and disadvantages – no one is the best in general. Good comparison of rotation representations together with performance issues is given in [9]. Somewhat more programming-oriented outfit presents [25]. There is only a few original ideas in this section, such as some clarifications (equivalence of two different formulas for SLERP) and the analogy between 2D/3D rotations and complex numbers/quaterions.

One can find this section somewhat tedious, but the results presented here are very important for the virtual fencing application. For example, the SLERP algorithm is used in a lot of parts of the program, thus we believe it is really worth a rather detailed description.

2.1 Rotation in 2D

We start with rotations in R^2, because they have more simple structure than 3D rotations, and there is a strong analogy with the 3D case. A rotation in 2D is given by a point $p \in R^2$ and an angle α. We can identify the point of rotation p with the origin 0, since a general rotation can be decomposed to a translation by $-p$, rotation of α about the origin and translation by $+p$. In the following, we consider only the rotation about origin. The convention (according to the 3D rotations convention established in section 1.5) is that the rotation runs in a counter-clockwise way.

The formula for computing a rotation of vector $(x, y)^T$ is

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \tag{2.1}$$

This can be neatly expressed in a complex plane. If we define the complex number $c = x + iy$, then we can write the transformation as

$$c' = e^{i\alpha} c$$

where $c' = x' + iy'$. This is equivalent to the formula (2.1), because of the Euler's identity

$$e^{i\alpha} = \cos(\alpha) + i \sin(\alpha) \tag{2.2}$$

Note that 2D rotations exactly correspond to unit complex numbers: each 2D rotation can be expressed as $e^{i\alpha}$ for some α, and any unit complex number describes a 2D rotation.

2.2 Matrix Representation

A 3D rotation can be represented by a matrix in an analogous way as the 2D rotation, see formula (2.1). A rotation x' of a vector $x \in R^3$ is expressed as

$$x' = Mx$$

The interpretation of the 3×3 matrix

$$M = (m_0, m_1, m_2)$$

is quite intuitive: the vectors m_0, m_1, and m_2 are the transformed standard basis vectors. We must pose some restrictions on the matrix M, otherwise we get a general affine transformation instead of rotation. A rotation does not change a magnitude of the transformed vectors and keeps the angles between two vectors intact. This means the vectors m_0, m_1, m_2 should have unit length and their dot product should be zero,

$$m_0 m_1 = m_0 m_2 = m_1 m_2 = 0$$

since this is true for the standard basis. This is a condition of *orthonormality* of matrix M, which can be shortly written as

$$MM^T = M^T M = I$$

This also means that M is regular and $M^{-1} = M^T$. Using the basic properties of determinant

$$det(M)det(M) = det(M)det(M^T) = det(MM^T) = det(I) = 1$$

thus $det(M) = \pm 1$. The orthonormal matrix describes either rotation or reflexion. Their difference is that the reflexion changes the orientation of the space, while rotation does not. The orientation of a coordinate system is described by the sign of the determinant. Thus the orientation of the space is not changed by orthonormal transformation M if and only if $\det(M) = 1$. To conclude: the rotation can be expressed as an orthonormal matrix M with $\det(M) = 1$, and any orthonormal matrix M with $\det(M) = 1$ describes a rotation.

2.3 Euler Angles

The Euler's rotation theorem [33] says that any rotation matrix M can be decomposed to product

$$M = R_{e_2,\alpha} R_{e_0,\beta} R_{e_2,\gamma} \tag{2.3}$$

where $R_{e_i,x}$ is a rotation matrix denoting rotation about axis e_i with angle x. For example $R_{e_2,\alpha}$ looks like this

$$R_{e_2,\alpha} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The angles α, β, γ are known as Euler angles. The interpretation of Euler angles is quite intuitive: it represents the rotation formed by concatenation of rotations about the principal axes. The decomposition presented in (2.3) is called the x-convention; there are also other conventions (e.g. pitch-roll-yaw, which works with e_0, e_1 and e_2).

Note that the axes e_i are the original (unrotated) axes of the world coordinate system. However, it is possible to express Euler angles in form

$$M = R_{e_2'',\gamma} R_{e_0',\beta} R_{e_2,\alpha}$$

where the e_0' is the vector e_0 rotated by $R_{e_2,\alpha}$, and analogically e_2'' is the vector e_2 rotated by $R_{e_0',\beta} R_{e_2,\alpha}$. It can be easily verified that this is really equivalent to (2.3), because

$$R_{e_0',\beta} = R_{e_2,\alpha} R_{e_0,\beta} R_{e_2,\alpha}^T \quad R_{e_2'',\gamma} = R_{e_0',\beta} R_{e_2,\alpha} R_{e_2,\gamma} R_{e_2,\alpha}^T R_{e_0',\beta}^T$$

The conversion from Euler angles to rotation matrix is described by equation (2.3). The matrix product can be evaluated, which results in efficient algorithm, as described in [25]. The conversion from matrix to Euler angles is based also on (2.3), and can be found in [25] or [9] as well. As a by product, it proves the Euler's rotation theorem.

The only advantage of Euler angles is that they are not redundant. A 3D rotation has exactly 3 degrees of freedom, and Euler angles use only 3 real numbers (matrix representation uses 9 reals, but there are 6 constraints due to orthonormality). However for majority of operations, other representations are are more advantageous. In fact, Euler angles are seldom used in the application.

2.4 Axis-Angle Representation

A 3D rotation can be also represented by an axis of rotation and an angle α. We can assume the axis of rotation contains the origin (because of the translation argument as for the 2D rotations), so it is sufficient to define the axis by its unit direction vector a. Note that a rotation given by $-a$ and $-\alpha$ is equivalent to that one given by a and α.

The axis-angle representation of a rotation can be converted to a matrix representation as follows. We define a coordinate system with basis vectors $a, a \times e_1, a \times (a \times e_1)$, which is a local frame associated with the axis of rotation. We transform this frame to the standard basis e_0, e_1, e_2, so that the axis of rotation aligns with e_0. Because of $\|a\| = 1$, this transformation is a again a rotation, that can be described by a matrix

$$R_{a \to e_0} = \begin{pmatrix} a^T \\ (a \times e_1)^T \\ (a \times (a \times e_1))^T \end{pmatrix}$$

The inverse transformation is the transposed matrix

$$R_{e_0 \to a} = (a, a \times e_1, a \times (a \times e_1))$$

The rotation about the x-axis (vector e_0) preserves the e_0 subspace and in e_1, e_2 acts like a 2D rotation, in a matrix form

$$R_{e_0,\alpha} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}$$

Thus we can express the rotation given by the axis direction a and angle α as

$$R_{a,\alpha} = R_{e_0 \to a} R_{e_0,\alpha} R_{a \to e_0}$$

By some rather lengthy derivations it can be shown, that this is equivalent to the form presented in [9],

$$R_{a,\alpha} = I + (\sin\alpha)A + (1 - \cos\theta)A^2 \tag{2.4}$$

where A is the matrix computed from the vector $a = (a_0, a_1, a_2)$ as

$$A = \begin{pmatrix} 0 & -a_2 & a_1 \\ a_2 & 0 & -a_0 \\ -a_1 & a_0 & 0 \end{pmatrix}$$

This matrix is skew-symmetric, and has a nice property: its multiplication by a vector acts like a cross-product with a, i.e. for any vector x holds that $Ax = a \times x$.

The opposite conversion, from matrix to axis-angle representation, can be based on formula (2.4), as derived in [9]. We need to compute the axis direction a and angle α from $R_{a,\alpha}$. The *trace* of a matrix is a sum of its diagonal elements. Especially for matrix $R_{a,\alpha}$, we have

$$\begin{aligned} Trace(R_{a,\alpha}) &= 3 - 2(1 - \cos\alpha)(a_0^2 + a_1^2 + a_2^2) = 1 + 2\cos\alpha \\ \cos\alpha &= \frac{Trace(R_{a,\alpha}) - 1}{2} \end{aligned}$$

We can confine only to a solution with $\alpha \in \langle 0, \pi\rangle$ and compute it as $\alpha = \cos^{-1}((Trace(R_{a,\alpha}) - 1)/2)$. Since A is skew-symmetric, it holds that

$$R_{a,\alpha} - R_{a,\alpha}^T = (2\sin\alpha)A \tag{2.5}$$

If $\alpha = 0$, then any direction a is valid, because the rotation is just identity. If $\alpha \in (0, \pi)$, then we can extract $a = (a_0, a_1, a_2)$ from (2.5), where it is only scaled by $2\sin\alpha$. In particular,

$$a_0 = A_{21} = \frac{1}{2\sin\alpha}(R_{a,\alpha} - R_{a,\alpha}^T)_{21}$$

and $a_1 = A_{02}$, and $a_2 = A_{10}$ can be extracted analogically. There is no need to compute the $2\sin\alpha$, since the constant can be canceled out by normalization to unit length. If $\alpha = \pi$, then equation (2.5) does not help, because $\sin\alpha = 0$. However note that in this case

$$R_{a,\alpha} = I + 2A^2 = \begin{pmatrix} 1 - 2(a_1^2 + a_2^2) & 2a_0a_1 & 2a_0a_2 \\ 2a_0a_1 & 1 - 2(a_0^2 + a_2^2) & 2a_1a_2 \\ 2a_0a_2 & 2a_1a_2 & 1 - 2(a_0^2 + a_1^2) \end{pmatrix}$$

Consider the case, that the first diagonal element $1 - 2(a_1^2 + a_2^2)$ has the maximal magnitude from all the diagonal elements. Then also $a_0^2 = \max(a_0^2, a_1^2, a_2^2)$, and it can be extracted from the matrix as

$$4a_0^2 = (1 - 2(a_1^2 + a_2^2)) - (1 - 2(a_0^2 + a_2^2)) - (1 - 2(a_0^2 + a_1^2)) + 1$$

The sign of a_0, which could not be recovered, is fortunately not important because the rotation angle $\alpha = \pi$. When we have already computed a_0, the other components a_1, a_2 are then computed directly from the first column of $R_{a,\alpha}$. The cases with $a_1^2 = \max(a_0^2, a_1^2, a_2^2)$ or $a_2^2 = \max(a_0^2, a_1^2, a_2^2)$ are handled in an analogous way.

2.5 Quaternion Representation

The axis-angle representation of 3D rotations has certain disadvantage because it is ambiguous. Even if we insist on unit axis of rotation, the rotations with angles $\alpha + 2k\pi, k \in Z$, are identical. To overcome this shortcoming, recall that 2D rotations are uniquely represented by unit complex number. The uniqueness is achieved by storing the sine and cosine of the angle, instead of the angle itself. The same trick can be applied to 3D rotations, which leads to quaternions – a generalization of complex numbers, introduced by Mark Hamilton.

There is also another, rather algebraic approach leading to quaternions. We can project a unit sphere (in 3D) to the complex plane by stereographic projection. Then any transformation of the unit sphere corresponds to some transformation of the complex plane. Especially, to the rotation of the unit sphere corresponds the transformation of the complex plane in the form

$$f(w) = \frac{aw + b}{-\bar{b}w + \bar{a}}, \; w \in C$$

The coefficients $a, b \in C$ are closely connected to the coefficients of the quaternion. This procedure is described in detail in [35].

A quaternion q is given as $q = w + xi + yj + zk$, where w, x, y, and z are real numbers and i, j, k are quaternion elements. The addition and subtraction of quaternions is done by components, as with R^4 vectors. The multiplication of quaternions is given by the multiplication of quaternion elements: $i^2, j^2, k^2 = -1, ij = k, jk = i, ki = j, ji = -k, kj = -i, ik = -j$. Namely the product of

two quaternions $q_0 = w_0 + x_0 i + y_0 j + z_0 k$ and $q_1 = w_1 + x_1 i + y_1 j + z_1 k$ is

$$\begin{aligned} q_0 q_1 &= (w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1) + \\ &\quad (x_0 w_1 + w_0 x_1 + y_0 z_1 - z_0 y_1) i + \\ &\quad (y_0 w_1 + w_0 y_1 + z_0 x_1 - x_0 z_1) j + \\ &\quad (z_0 w_1 + w_0 z_1 + x_0 y_1 - y_0 x_1) k \end{aligned}$$

Note that the multiplication of quaternions is not commutative. The conjugate of quaternion q is $q^* = w - xi - yj - zk$. The norm of a quaternion q is $\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}$, unit quaternion is a quaternion with norm 1. A complex number is a special case of quaternion with $y = z = 0$.

The 3D rotation given by unit axis a and angle α can be represented by a unit quaternion q such that

$$q = \cos(\alpha/2) + a_0 \sin(\alpha/2) i + a_1 \sin(\alpha/2) j + a_2 \sin(\alpha/2) k$$

Because the axis of rotation has unit length, it can be easily shown that $\|q\|$ is really 1. On the other hand, from any unit quaternion can be extracted the angle and axis of some 3D rotation[1]. For unit quaternion q holds that $q^* q = q q^* = 1$, where $1 = 1 + 0i + 0j + 0k$. The 1 can be interpreted as identity (zero angle rotation) and the conjugate quaternion q^* as an inverse rotation.

The analogy between complex numbers and quaternions goes even further: the Euler's identity for complex numbers (2.2) generalizes to quaternions. If we identify the unit axis of rotation a with quaternion $a = a_0 i + a_1 j + a_2 k$, we can express the rotation about axis a with angle 2α by quaternion

$$e^{a\alpha} = \cos\alpha + a \sin\alpha \tag{2.6}$$

This is in fact the Euler's identity with complex element i replaced by quaternion a. The equation can be proven by taking the Taylor series for exponential function e^x, substituting $a\alpha$ for x and realizing that the quaternion product $aa = -1$ (due to the unit length of axis a).

To convert an axis-angle representation to quaternion is straightforward, as well as quaternion to axis-angle. Because we have already described how the rotation matrix can be converted to axis-angle representation and vice-versa, we have also a procedure that converts matrix to quaternion and

[1]The reason for taking cosine and sine of $\alpha/2$ instead of α is in the rotation expression by two reflexions. Any 3D rotation can be given as a composition of two reflexion planes, whose intersection defines the axis of rotation. If these two planes incline an angle θ, then the angle of rotation is 2θ.

quaternion to matrix. However, the conversion via the axis-angle representation is not the most effective one. Direct (and faster) conversions between matrix and quaternion are described in [9].

2.6 Spherical Linear Interpolation

A 3D rotation interpolation is far more complicated than the interpolation of position. The attempt to linearly interpolate the rotation matrix element by element is naive: this method can violate the orthonormality of the matrix, and even worse one of the interpolated matrices may have a lower rank. It means that if some object is transformed by the interpolated matrix it can be arbitrarily zoomed, skewed and even projected to a plane. The orthonormalization of the matrix solves some of the problems, but not all of them, as we clarify in the following.

Similarly, the element by element interpolation of quaternions will not perform well. As discussed before, the 3D rotations can be represented by unit quaternions. The set of all unit quaternions can be viewed as 4D sphere with unit radius. Because it is somewhat difficult to visualize 4D sphere, we demonstrate the problem for the case of 2D rotations, which correspond to a unit circle in the complex plane. Consider two 2D rotations given by unit complex numbers r_0, r_1. The naive way of interpolation would look like

$$r_t = (1-t)r_0 + tr_1,\ t \in \langle 0, 1\rangle$$

The $\|r_t\|$ needs not be 1, but we can normalize it to $r_t/\|r_t\|$. The problem is, that it is the line segment r_0, r_1 which is interpolated with constant step. The normalization of the interpolated values is not uniform in the arc length, as illustrated in the picture 2.1. If we treat the parameter t as time, then it means that the angle of rotation does not change with constant velocity, i.e. not linearly.

Let us summarize the requirements on the linear interpolation s_t of rotations[2] p, q with parameter $t \in \langle 0, 1\rangle$

- $s_0 = p, s_1 = q$
- s_t is indeed a rotation for each $t \in \langle 0, 1\rangle$
- the interpolation is uniform: an object transformed by s_t rotates with constant angular velocity

[2] in arbitrary dimension (2D/3D) and representation

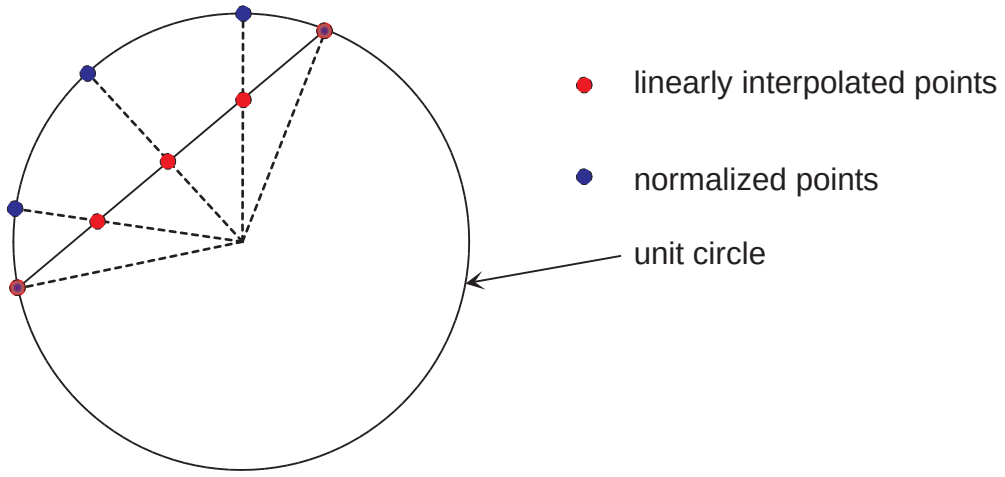


Figure 2.1: Non-uniform sampling of the circle

It is easy to meet these conditions for the 2D rotations r_0, r_1 from the example above, because they can be written as

$$r_0 = e^{i\tau},\ r_1 = e^{i\nu}$$

and interpolated as

$$s_t = e^{i(1-t)\tau + it\nu} \tag{2.7}$$

It is obvious that $s_0 = r_0, s_1 = r_1$, s_t is a 2D rotation (unit complex number), and that the interpolation is uniform, since it is the angle of rotation which is interpolated here.

The situation is somewhat more complicated for the 3D rotations, because there is no common axis of rotation as in the 2D case. But since we have the generalization of the Euler's identity (2.6), we can generalize the just described interpolation procedure to 3D rotations as well. Consider two 3D rotations given by quaternions p, q. According to formula (2.6), they can be written as

$$p = e^{u\theta}, q = e^{v\phi}$$

where u, v are unit quaternions with zero real part, describing the axis of rotation. Now it is possible to generalize the formula (2.7) for them, leading to

$$s_t = e^{u(1-t)\theta + vt\phi}$$

This can be equivalently expressed using only p, q

$$s_t = p(p^*q)^t \tag{2.8}$$

It is clear that $s_0 = p$, and $s_1 = q$. The conjugate of a unit quaternion is a unit quaternion, the product of unit quaternions is again a unit quaternion and the power of unit quaternion is also a unit quaternion – algebraically speaking, the set of quaternions forms a group. Therefore, s_t is a unit quaternion, i.e. a 3D rotation. If we define $r = p^*q$ and expand it as $r = e^{h\sigma}$, we can rewrite equation (2.8) to

$$s_t = pr^t = pe^{h\sigma t}$$

which says that s_t is the composition of constant rotation p and rotation about axis h with angle $2\sigma t$. It follows that the interpolation is uniform in the angle, and the interpolation formula (2.8) satisfies all the requirements made above.

The interpolation of rotations according to formula (2.8) is known as *Spherical Linear Interpolation*, which is usually abbreviated to *SLERP*. The reason for this name is following: imagine the space of unit quaternions as unit sphere in R^4. Any two rotations p, q define points on this sphere. What does the SLERP compute is the linear interpolation between p and q along an arc that connects p and q on the unit sphere. This can be seen easily in the 2D case from the formula (2.7). The parametrization of an arc on the unit sphere in R^4 is not so trivial, and can be found in [9].

There is yet one more subtlety that remains to be clarified. The $r^t = e^{h\sigma t}$ term expresses interpolation of rotation about a fixed axis h with angle $2\sigma t$. However, there are two ways how such interpolation can be done: either along the shorter or the longer arc (if $2\sigma = \pi$, then both arcs have equal length), as illustrated in Fig. 2.2.

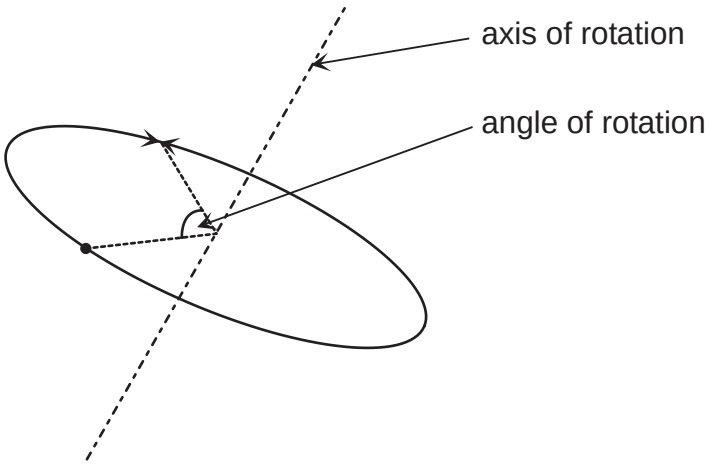


Figure 2.2: Two ways of rotation interpolation

Which way the interpolation travels is determined by the real part of quaternion r. Because the quaternion r is unit, it follows that it can be written as

$$r = \cos\sigma + h\sin\sigma$$

and h has zero real part.

On the other hand, quaternions p, q can be expressed as

$$p^* = w_p - x_p i - y_p j - z_p k,\ q = w_q + x_q i + y_q j + z_q k$$

thus the quaternion product

$$r = p^* q = w_p w_q + x_p x_q + y_p y_q + z_p z_q + \cdot\, i + \cdot\, j + \cdot\, k$$

The coefficients marked by $\cdot$ are not interesting. The real part of quaternion r is nothing else than the standard R^4 dot product pq. By comparing these two equal expressions of r we can conclude that $\cos\sigma = pq$. This is sufficient to determine σ uniquely, because $\sigma \in \langle 0, \pi)$ is sufficient to express any angle of rotation in $\langle 0, 2\pi)$.

If the dot product $pq > 0$, then the angle σ is acute and the angle of rotation 2σ is within $(-\pi, \pi)$, thus the interpolation travels the shorter path. If $pq = 0$, then both paths have equal length, and either of them can be chosen. If $pq < 0$, then it corresponds to the larger path. This can be avoided by considering that the quaternions q and $-q$ denote the same rotation, because both the axis and the angle are negated. Nevertheless, employing $-q$ instead of q in the SLERP formula results in $-r$ instead of r. Although r and $-r$ denote of course also the same rotations, the interpolation of $-r$ uses the other path, because it rotates about a negated axis with negated angle. To conclude, to ensure the shortest path interpolation, it is sufficient to check the sign of the dot product pq and if negative, then use $-q$ instead of q. In certain situations it can be advantageous to know that there are two possible ways that can be chosen from, see section 6.5.1.

The parametrization of the shortest arc according to [9] results to the formula

$$s_t{}' = \frac{\sin((1-t)\sigma)p + \sin(t\sigma)q}{\sin\sigma} \tag{2.9}$$

with the same notation as above. Using this equation the SLERP can be computed quite efficiently, but it is not obvious if the result is really equivalent to formula (2.8). To show this, recall that quaternion r was defined as

$r = p^*q = e^{h\sigma} = \cos\sigma + h\sin\sigma$. Using this we can prove that

$$\begin{aligned} {s_t}' &= p\,\frac{\sin((1-t)\sigma) + \sin(t\sigma)r}{\sin\sigma} \\ &= p\,\frac{\sin((1-t)\sigma) + \sin(t\sigma)\cos\sigma + h\sin(t\sigma)\sin\sigma}{\sin\sigma} \\ &= \frac{p}{\sin\sigma}\left(\sin\sigma\cos(t\sigma) - \cos\sigma\sin(t\sigma) + \sin(t\sigma)\cos\sigma + h\sin(t\sigma)\sin\sigma\right) \\ &= p\left(\cos(t\sigma) + h\sin(t\sigma)\right) \\ &= pr^t = s_t \end{aligned}$$

2.6.1 SLERP on matrices

It may look like that the SLERP is tightly connected to quaternions. This is not true – an algorithm equivalent to SLERP can be performed on the rotation matrices as well. It is a little bit less effective than the SLERP on quaternions, but it demonstrates more clearly what the SLERP is actually doing (without confusing with quaternions and the 4D sphere).

The idea is that the equation (2.8) can be rewritten for rotation matrices P, Q in a straightforward way

$$S_t = P(P^TQ)^t,\ t \in \langle 0, 1\rangle$$

One can immediately see that $S_0 = P$ and $S_1 = Q$. The transpose of a rotation matrix (orthonormal with a positive determinant) is a rotation matrix and a product of rotation matrices is a rotation matrix.

The only problem is how the power of a matrix $R = P^TQ$ should be defined. The matrix R can be converted to the axis-angle representation, using the algorithm described in section 2.4. Assume the angle $\beta \in \langle 0, \pi\rangle$ and unit axis b were extracted. The interpolated rotation can be defined as the rotation with angle $t\beta$ about the axis b. This can be then converted back from the axis-angle representation to the matrix.

The conversion, as described in section 2.4 ensures that the result will be a rotation matrix, therefore S_t is also a rotation matrix. The interpolation is uniform – the physical analogy of constant angular velocity is quite obvious here.

Since we bounded the angle of rotation to $\beta \in \langle 0, \pi\rangle$ in the conversion to axis-angle representatin (to ensure a uniqueness), the interpolation corresponds to the shortest path, according to the Fig. 2.2. The longer one can be obtained by taking the axis $-b$ (instead of b) and angle $2\pi - \beta$ (instead of β). Similarly as for quaternions, this does not change the resulting position S_1, but changes the direction of interpolation.

This algorithm can be advantageous if we have already the routines for conversion between axis-angle and matrix representations, and if the speed of the interpolation is not the primary objective. This is how the SLERP was implemented in our application. In fact the function performing SLERP is a part of IKAN libraries, which are discussed in section 4.2. The function names `linterpikMatrix` and is located in the module `myvec.cxx`.

Chapter 3

Virtual Humanoid Visualization

One of the basic requirements on the application is to simulate virtual humanoid actions and reactions. Therefore, it is essential to design an effective method for virtual humanoid visualization, with a stress on a virtual fencer. This is important since the fencer model has some specific demands – the most important is the very large range of motion in the joints of the armed hand.

A natural approach would be to simulate the anatomy of the human body: bones, muscles, fat tissues and the skin. However such methods are used especially in the non-interactive rendering, because of the high computing complexity. For our application is necessary virtual character animation in real-time, which requires faster algorithms.

The common approaches used for the real-time virtual humanoid animation are the basic skeletal animation and vertex blending, described in the next two sections. Articles dedicated to game developers that describe basic skeletal animation as well as vertex blending are [21], [23]. The source code for the basic skeletal animation algorithm has been adopted from [31]. The basic skeletal animation algorithm is also used by the program MilkShape3D 1.5.6. The animation software CharacterFX 1.2 uses the vertex blending algorithm.

The bones blending algorithm [15] is our new contribution. It was developed in order to overcome some artifacts of the previous methods. An improvement of vertex-blending with similar goals was presented in [38], but we believe that our solution has certain advantages over this one. As we have discovered recently, yet another approach has been published [6]. We suppose it could give better results than the bones blending algorithm, but with a cost of complicated auxiliary structures (the medial). An interest-

ing generalization of vertex blending is [37]. It uses more than one blending parameter per bone, which allows more freedom in skin deformation. The cost is that the tunning of these numerous parameters is more difficult; it is done via the least-squares regression, using a set of corresponding skin and skeleton postures.

3.1 Basic Skeletal Animation

In this section we explain the basic algorithm, which is used in PortaLib3D [31] as well as in MilkShape3D to animate a virtual character, and compare its advantages and disadvantages.

The virtual humanoid body is modeled by two incidental objects: a digital skin and a skeleton. The *digital skin* is a representation of the surface of the body. It consists of

- set of vertices, which are points from R^3.
- set of triangles, which are ordered[1] triplets of points.
- texture, a 2D bitmap of colors. Each vertex has assigned a point from this bitmap.
- normal vectors, which are unit vectors from R^3. They are defined in each vertex for the purposes of lighting.

The vertices and triangles of our humanoid model are drawn in Fig. 3.1, with culled back faces.

The direct animation of the skin would be complicated. To simplify this task we introduce the skeleton, which has a nice physiological justification. Mathematically speaking, the *skeleton* is a tree. The nodes of this tree represent joints and the edges correspond to bones. We denote the joints by positive integer numbers. The root of the tree is assumed to have index 0 and the parent of joint j is denoted as $p(j)$. In each node j of the skeleton is given a transformation represented by a 4×4 homogenous matrix $R(j)$. Naturally, we do not allow general affine transformation, such as zooming or skewing: besides translation, only a pure rotation is allowed. More formally it means, that the homogenous matrix $R(j)$ has the form

$$\begin{pmatrix} M(j) & t(j) \\ 0 & 1 \end{pmatrix}$$

[1]The order of the vertices follows common counter-clockwise convention.

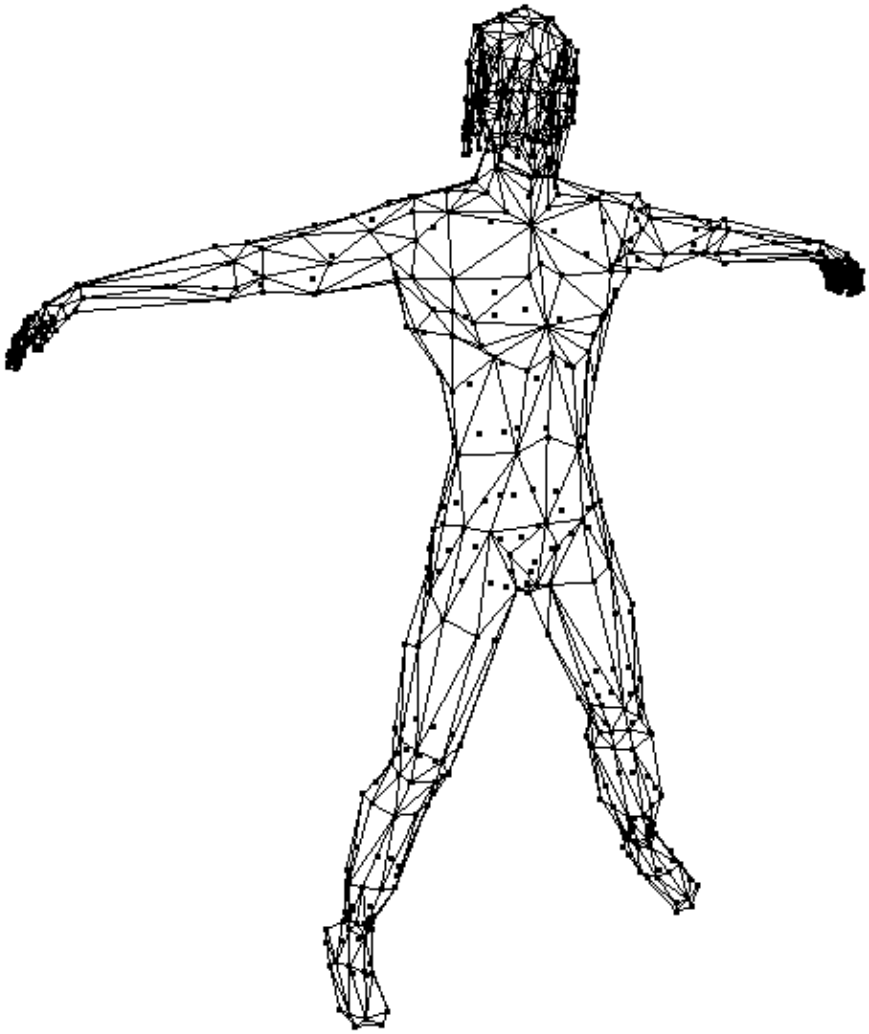


Figure 3.1: Triangular mesh of virtual humanoid Yopago.

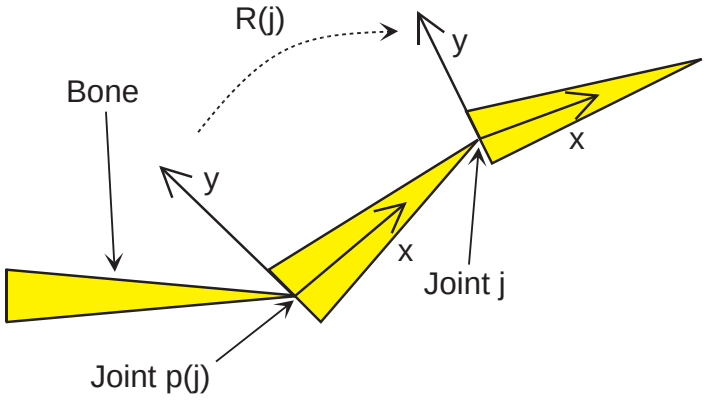


Figure 3.2: Reference skeleton joint transformation

where $M(j)$ is an orthonormal 3×3 matrix and $t(j) \in R^3$. The transformations $R(j)$ for all the joints j define the initial posture of the skeleton, usually called a *reference position*.

Each joint can be assigned a local coordinate system. $R(0)$ is the transformation of the whole model, relative to the world space coordinate system. $R(j)$ expresses the transformation from the coordinate system in joint $p(j)$ to that one in joint j, as depicted for the 2D case in Fig. 3.2. The total transformation $A(j)$ from the world coordinate system to the joint's j coordinate system then can be written as

$$A(j) = R(0) \cdots R(p(j))R(j)$$

A picture of our model's skeleton is in Fig. 3.3. In the picture, the joint j (blue sphere) is drawn in the position given by the translation part of $A(j)$ (the hands have joints for every finger, thus the crowd of blue spheres). The bones (yellow links) are simply connecting the joints. Of course it would be possible to describe the reference skeleton posture with rotational components of all $R(j)$s set to identity, but the possibility of general rotation simplifies the skeleton design – the designer can rotate the whole sub-trees of the skeleton.

To animate the skeleton we define a transformation matrix $T(j)$ for every joint j to describe the actual transformation of this joint. Note that only pure rotation (no translation) is acceptable for $j \neq 0$. The reason is that translating a non-root joint does not lead to natural skeleton movement (the joints in a real body can be translated only within a negligible distance). The root joint is an exception – its translation means translation of the whole model. The final transformation $F(j)$ from the world coordinate system

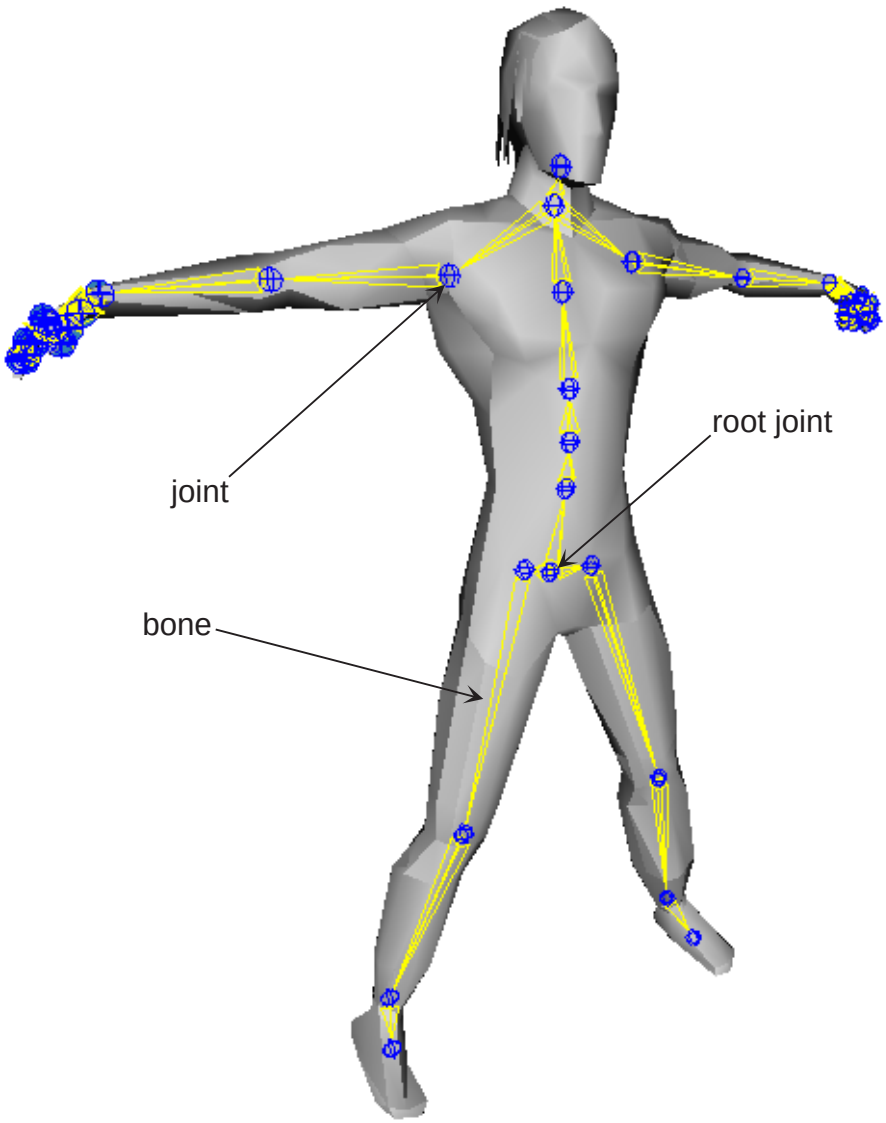


Figure 3.3: Smooth shaded Yopago model with skeleton.

to joint j's coordinate system, in the animated skeleton can be expressed recursively as

$$\begin{aligned} F(0) &= R(0)T(0) \\ F(j) &= F(p(j))R(j)T(j) \end{aligned}$$

Matrix $F(j)$ is the analogy of matrix $A(j)$ in the manipulated skeleton. The joints are stored in an array indexed with joint numbers. If any joint j meets the condition $p(j) < j$, then $F(j)$ (resp. $A(j)$) can be computed exactly from the definition by a single pass of the array. The numbering satisfying $p(j) < j$ can be obtained for example by the depth first search algorithm, starting in the root.

In the application, the joint is represented by structure `Model::Joint`. In this structure, `Joint::m_relative` corresponds to $R(j)$, `Joint::m_absolute` means $A(j)$, `Joint::m_final` is $F(j)$ and `Joint::m_transform` stands for $T(j)$. The computation of final matrices is done in function `Model::setFinalMatrices`.

The skeleton helps us significantly with the virtual humanoid animation – instead of animating vertex by vertex, it is sufficient to manipulate only the skeleton. We have already described the skeleton positioning, and it remains to discuss how the posture of the skeleton should be propagated to the skin (i.e. vertices forming a triangular mesh). In fact, this is the main problem of the skeletal animation, since it is necessary to achieve a smooth skin deformation using only a non-smooth (segmented) skeleton.

To arrange the automatic propagation of the movement from the skeleton to the skin, we must define the skeleton-skin relationship in some way. The first approach, we call the basic skeletal animation proceeds in a straightforward way. It assumes each vertex is attached to one and only one joint, and the vertices are transformed as if they were firmly connected to the bone. Put in a more formal way, if a vertex in position $v \in R^3$ is attached to a joint j, then its transformed position v' is computed as

$$v' = F(j)A(j)^{-1}v$$

The interpretation of the formula is obvious: the $A(j)^{-1}$ transforms the vertex from a joint j's local coordinate system to the world coordinate system (with 0 being the origin). This is because we need the rotation $T(j)$ to run upon the joint j. The multiplication by $F(j)$ can be understood as follows: first apply the joint rotation $T(j)$ and then return the rotated vertex from the world coordinate system to the joint j's actual local coordinate system. Note that if all $T(j)$'s are identities (the actual skeleton position is exactly the reference position), then $v' = v$ as expected.

The skin deformation resulting from the basic skeletal animation is illustrated in Fig. 3.4. In the picture, the color indicates the vertex to bone assignment. We see that for the vertices in the middle, it is hard to decide to which bone they should be attached – both bones are equally suitable. From the picture is also apparent the main drawback of the basic skeletal animation: the triangles on the boundaries of two bones bear all the deformation. The other triangles (with all vertices belonging to the same joint) are not deformed at all, which is definitely not a plausible skin deformation. Put in another words, the resulting shape of the skin is not smooth. A better algorithm should distribute the deformation over more triangles nearby the joint to achieve smoothness.

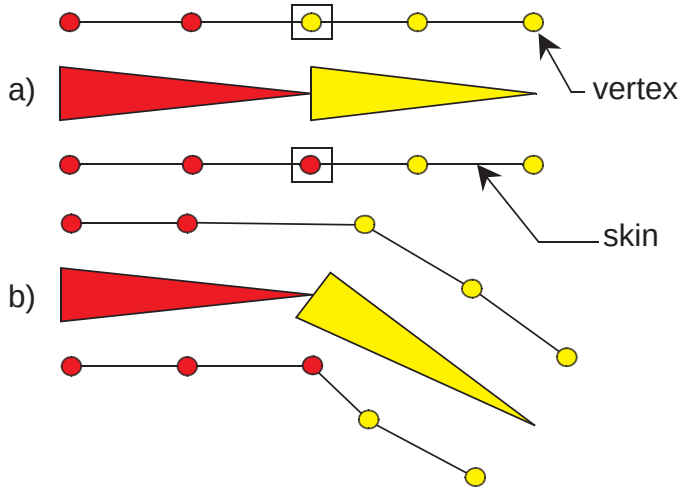


Figure 3.4: Basic Skeletal Animation: a) reference position, b) after rotation

3.2 Vertex Blending

Vertex blending is an attempt to correct the disadvantage of the basic skeletal animation. It is a generalization of the basic skeletal animation with a more sophisticated skeleton to skin relation. Its idea is that a vertex can be attached to a set of bones, instead of only one bone, as in the basic skeletal animation. This set has usually a small cardinality (about 2 to 4). The resulting position of the transformed vertex is computed as a convex combination of the individual transformations. More precisely, if the vertex in position v is attached to joints $j_0, \ldots, j_n$ then its transformed position v' is

computed as

$$v' = \sum_{i=0}^{n} w_i F(j_i) A(j_i)^{-1} v$$

where the w_i's are the coefficients of a convex combination, i.e. satisfying

$$\sum_{i=0}^{n} w_i = 1, \;\; w_i \geq 0 \;\forall i \in \{0, \ldots, n\}$$

The w_i can be interpreted as the influence of joint j_i on the transformation of vertex v. The weights w_i can be edited by hand, or they can be computed by a heuristic algorithm [38]. In a brief, the bones with the smallest distance to the vertex are found, and the weights are proportional to that distance. However, we did not need to implement such algorithm because the weights were already provided with our virtual humanoid model.

Recall the vertices in Fig. 3.4 marked by the rectangle: it was hard to decide to which one bone such vertices should be attached to, but it is easy when a set of bones is allowed. A natural good choice is to assign these vertices to both bones with weights $w_0 = 0.5$ and $w_1 = 0.5$. The other vertices are connected to only one bone as before. The deformation of the skin using vertex blending is drawn in Fig. 3.5. In the picture is demonstrated the averaging (blending) of the vertices transformed by both bones.

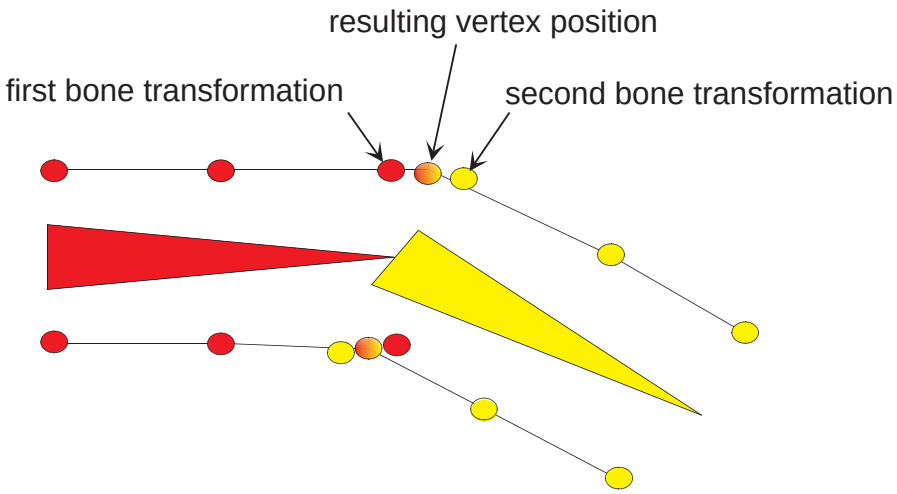


Figure 3.5: Vertex Blending

Although the vertex blending algorithm with properly weighted vertices produces smooth skin deformation, they may not look naturally for all joint

rotations. It was already described in [38] that vertex blending does not perform well, when the joint rotations become large. A classic example of this behavior is known as *elbow twist*, or a *candy-wrapper* artifact. It is a position with the elbow twisted 180 degrees around its axis. The situation is illustrated in Fig. 3.6, if we imagine the two bones represent an arm and that the yellow bone is a forearm.

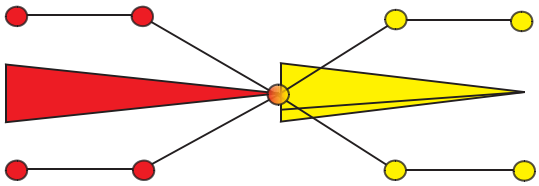


Figure 3.6: Twisting Elbow Problem

The same situation in the 3D case presents Fig. 3.11a. The weights are assigned as above – namely the vertices marked by the rectangles are connected to both bones with equal coefficients. In Fig. 3.6, the elbow joint is twisted 180°. After averaging, both vertices end up in the same position, the joint itself. This is of course not acceptable skin deformation. One may object that 180° is a not natural position – and it is true – but even for common, comfortable postures, similar artifacts were observed, see Fig. 3.12a.

In spite of these problems, the vertex blending algorithm is commonly used for real-time virtual humanoid animation, especially in computer games. In some applications the vertex blending is indeed sufficient, but when large joint rotations are necessary, as in fencing, different solutions must be applied. [38] proposed a method to overcome problems similar to twisting elbow. This method is called *bone links* and it works with auxiliary bones. The auxiliary bones replace the original ones, and they spread a large rotation to more smaller ones. Then the vertex blending leads to better skin deformation. Unfortunately, [38] says nothing about how the auxiliary bones should be positioned.

3.3 Bones Blending

We developed an alternative algorithm producing smooth skin deformation without twisting elbow-like problems [15]. This approach builds only on the basic skeletal animation algorithm, therefore it can be considered as an alternative to vertex blending. It is one of the advantages over the bone links solution [38], because the designer needs not to bother with the vertex

weights. The set of parameters for bones-blending is much smaller and thus easier to interpret and tune.

The problems of vertex blending arise from the fact that averaging is done on vertices. If we could arrange the blending on a lower level, such as bones, it would give better results. As mentioned earlier the joints are essentially rotations. The main idea of bones blending technique is not to apply the whole rotation at once, as in the basic skeletal animation, but to perform a blend of previous rotation and the current one. Blending of two rotations is easily accomplished by SLERP, described in section 2.6. Then the only problem is where to take the interpolation parameter t.

Intuitively, it should be connected to the distance along the bone in the reference posture. To specify the distance along the bone we use the matrix $A(j)^{-1}$ that transforms the vertices from the reference position to the normalized position, with joint j centered at the origin. If the following joint is $k, p(k) = j$, then the bone segment is determined by $R(k)$, namely by the translation component of $R(k)$ – we unitize this vector and denote the result as n. Then the distance of the normalized vertex along the bone can be defined as distance from the plane with normal n containing origin. It can be computed by the dot product

$$t' = v_0 n, \;\; v_0 = A(j)^{-1} v$$

where v_0 is the normalized version of the given vertex v. To prove that t' is really vertex to plane distance consider that $nn = 1$, thus

$$(v_0 - t'n)n = v_0 n - t' = v_0 n - v_0 n = 0$$

The t' is positive for points in the n direction, zero for points incident to the plane and negative for points behind the plane (in the $-n$ direction). Let us assume we have two values:

- t_{min} - the value of t' where only the parent joint's rotation is applied
- t_{max} - the value of t' where only the child joint's rotation is applied

Now it is straightforward to derive the interpolation parameter

$$t = \frac{t' - t_{min}}{t_{max} - t_{min}}$$

clamping values to $\langle 0, 1 \rangle$. Note that t' and therefore t depend only on the reference skin and skeleton posture, so they can be pre-computed. This method assumes that the limbs for bones blending are outspread in the reference posture, which is often the case. Nevertheless, it would not be too complicated to generalize the algorithm for bended limbs as well.

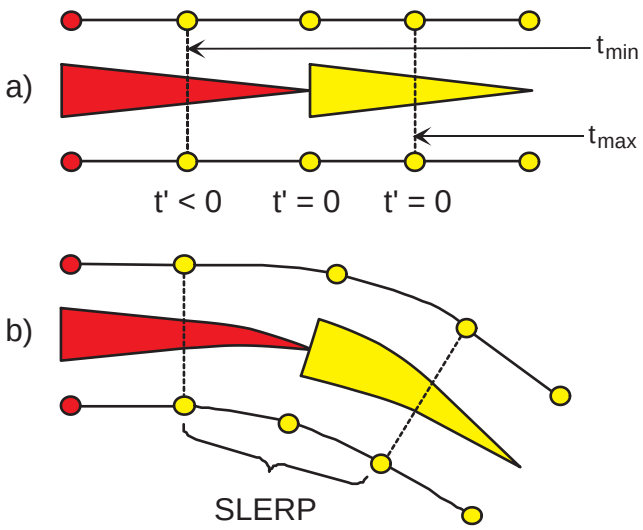


Figure 3.7: Bones Blending: a) vertex assignment and parameters, b) after rotation

3.3.1 Parameter tuning

The parameters t_{min} and t_{max} are object of editing and tuning, as well as the vertex-bones assignment, which needs already the basic skeletal animation. The vertices are usually attached to bones manually - using an editor, although some heuristic algorithms also exist [38]. We need not have to solve this problem, because the data were already provided with the Yopago model. The general rule for the basic skeletal animation is to assign vertices to the nearest bones.

For bones blending the situation is somewhat different. The vertex assignment in this case is connected with t_{min} and t_{max} parameters. The general rule can be now: assign all the vertices that should be modified when joint j rotates to the joint j. It means that all the vertices around the joint j should be connected to the bone beginning in j instead of the bone that ends in j. Those newly assigned vertices will be those that give negative t'. The vertices with non-negative t' are still assigned to joint j, unless they are subject of blending with the following bone - it depends on parameter t_{max}. Compare the vertex to bone assignment for the basic skeletal animation in Fig. 3.4a and for bones blending in Fig. 3.7a.

If t_{max} - t_{min} is large, then the bend is broad, spread over all attached vertices. On the other hand when it is too small, the result is similar to basic skeletal animation. The extreme case with t_{max} - $t_{min} = 0$ would lead

to the same skin deformation as produced by the basic skeletal animation, thus bones blending is indeed a generalization. The optimal values should be tuned in an editor, along with the vertex to bone assignment. An example of good parameter settings for the right arm of our virtual humanoid is in Fig. 3.8.

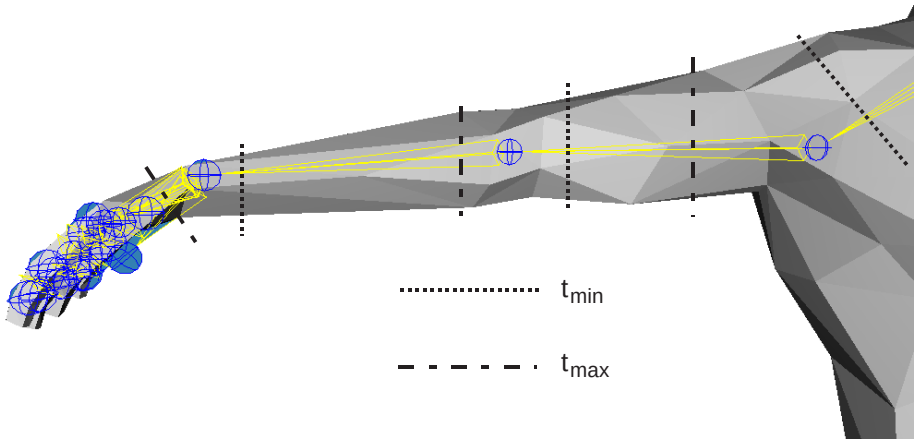


Figure 3.8: Our choice of t_{min} and t_{max} parameters

3.3.2 Triangle Subdivision

Bones blending brings little improvement if the model mesh is too coarse – the triangles are large and the effect of smooth bone deformation vanishes (the smooth curves drawn in Fig. 3.7b can be thought as a limit case for an infinite number of vertices). On the other hand the smoother the original mesh, the smoother the deformed skin, and no other parameters (such as vertex weights) must be defined. It is another advantage of bones blending over other methods such as the vertex blending and its generalizations (bone links).

The subdivision is no problem, if there are the original surfaces (Bezier, B-spline) the model was built of. However, we often know only the triangular mesh location (and even worse optimized/decimated to the reference position). In this case it is necessary to subdivide large triangles, especially in the most strained regions – around the joints with wide range of motion. The triangle connected to joint j is defined as a triangle with all vertices attached to j. The triangles for subdivision due to joint j are all the triangles connected to joint j. There are two basic methods for triangle subdivision, 1 to 3 and 1 to 4 triangles, see Fig. 3.9.

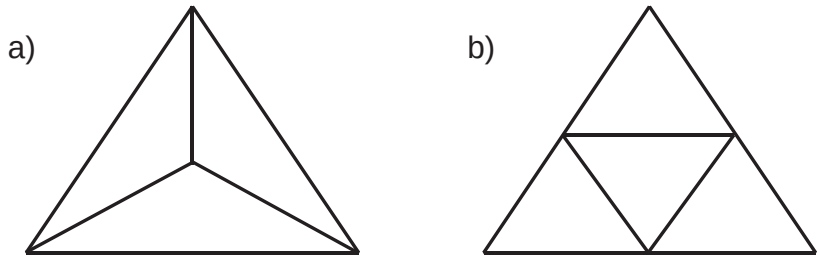


Figure 3.9: Two types of triangle subidivision: a) 1 to 3 triangles, b) 1 to 4 triangles

Each of them has its pros and cons: 1:4 produces T-vertex discontinuities, which results in holes in the deformed skin. This must be corrected by induced division of neighboring triangles. On the other hand 1:3 subdivision tends to produce oblong triangles (i. e. with high ratio of circumscribed and inscribed circle radius). To conclude: 1:4 offers higher quality and 1:3 gives faster animation. Another question is which joints should be selected for bones blending and then possibly for subdivision. Apparently not all the joints deserve the special treatment by bones blending: [38] states that for rotations with angle less than 60 degrees, the vertex blending produces good results. This problem, as well as the problem of t_{min} and t_{max} parameters tunning is of rather artistic nature, and should be solved during the virtual humanoid design in an interactive editor.

When developing the application, no editor supporting bones blending existed[2]. Therefore, we tuned the t_{min} and t_{max} parameters by hand. The vertex to bone assignment had to be altered too, but this is of course supported by the animation software.

3.4 Implementation and Comparison

The bones blending algorithm proved to be powerful enough to be used for the skin deformation of a virtual fencer. For the purposes of comparison we have also implemented the basic skeletal animation and vertex blending. Regardless of the actual method, the skin deformation is computed in method `Model::compActVers`. The bones blending is supported only for the right arm. The function `compActVers` takes the vertex positions in the reference skin `Model::m_pVertices` and transforms them to their positions in the de-

[2]However future versions of CharacterFX may support the bones blending algorithm, as promised by its programmers.

formed skin, `Model::m_pActVers`. These actual vertex positions are not used only for drawing to the display (by method `Model::draw`), but also for the the collision detection with the fencer's body, which is described in chapter 5.

The triangle subdivision is also implemented in class `Model`. The 1:3 subdivision is done by function `Model::subdivMesh1k3`, which simply scans all the triangles, and evaluates the subdivision condition for every triangle. As mentioned before, it means checking that all the triangle vertices are assigned to the same joint j. If this is true, the triangle is divided into three, according to Fig. 3.9a.

The 1:4 subdivision, implemented in function `Model::subdivMesh1k4`, is somewhat more complicated due to the induced division. In the first step, the triangles are scanned, tested and subdivided as in the 1:3 case, but of course 1:4. In the second step, the T-vertices are corrected by another pass of the array of triangles. For every triangle T is computed the number of neighboring triangles subdivided in the previous step. According to this number is subdivided the triangle T, see Fig. 3.10.

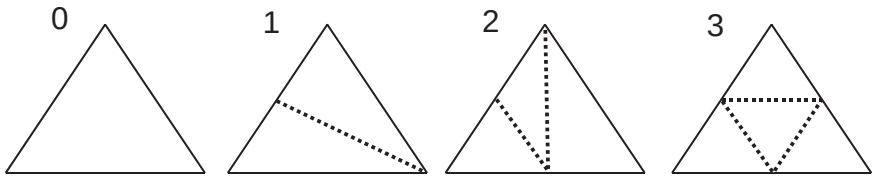


Figure 3.10: Subdivision induced by 1:4 subdivision of 0, 1, 2, and 3 neighboring triangles.

The nice feature is that the second step adds no new vertices, because they are already created in the first step. For the final user is accessible only one predefined type of 1:4 subdivision of the right hand. It subdivides the triangles around the shoulder, elbow and wrist joint in order to achieve nice appearance.

A virtual fencer is by definition a virtual humanoid equipped with a weapon – sword in particular. Although the basic skeletal animation is not suitable for skin deformation, it is appropriate for rigid objects, such as the sword (the elasticity of a sword is only marginal and can be neglected). The weapon is connected to the virtual character using an auxiliary joint named *sword-holder*. The sword-holder is linked to the wrist and it determines the position and orientation of the weapon. The weapon model itself is stored aside from the humanoid model, in a separate file. The purpose of the sword-holder is to enable a normalized position of the weapon model, independent of the actual virtual humanoid that will be holding it.

The visual quality of the skin deformation methods is compared in Fig. 3.11 for the extreme case of twisted elbow.

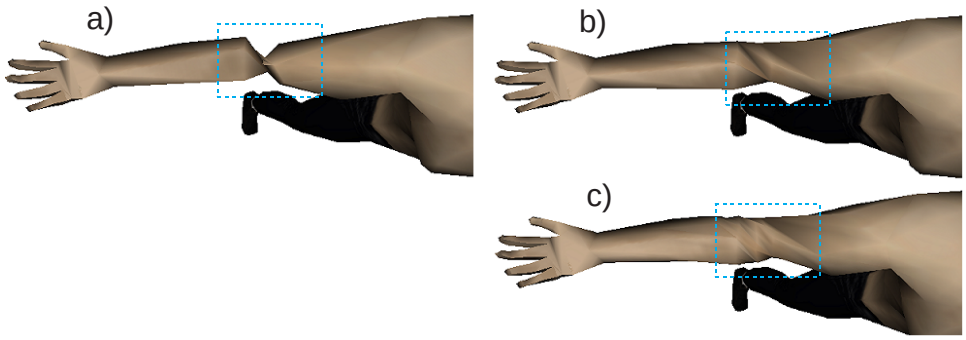


Figure 3.11: Elbow twisted 180 degrees rendered with a) vertex blending, b) bones blending, c) bones bleding with 1:4 subdivision

The vertex blending on the subdivided skin was not rendered, because it would not be worth the effort of vertex weights tunning. From Fig. 3.6 it is apparent that increasing the number of vertices would not improve the shape of the collapsed skin anyway.

As mentioned earlier, the elbow twist is not a natural posture. However, vertex blending produces artifacts even for natural postures, especially in the shoulder joint. This is no surprise, since the skin deformation around the shoulder is a challenge for animation algorithms, as stated in [6]. The comparison of vertex and bones blending for an everyday fencing posture is in Fig. 3.12.

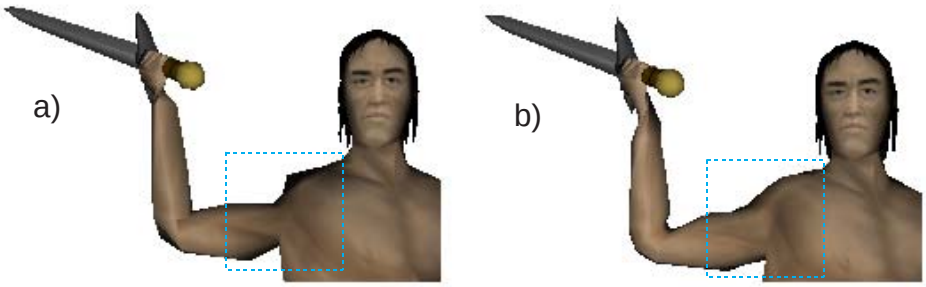


Figure 3.12: Natural posture rendered with a) vertex blending, b) bones blending with 1:4 subdivision

The rendering time was measured on an original triangle mesh and on a

model with right arm subdivided 1:4. Both models were rendered with vertex blending and with bones blending applied on the right arm. The resulting time is in Tab. 3.1, and the input data size in Tab. 3.2.

	original mesh	1:4 subdivided
vertex blending	16.57	23.31
bones blending	19.04	32.7

Table 3.1: One frame rendering time in milliseconds

	original mesh	1:4 subdivided
vertices	1507	2065
triangles	1372	1930

Table 3.2: Size of input data

In our implementation, the parameters t_{min} and t_{max} do not influence the rendering time. However, it would be possible to cancel the SLERP, when the interpolation parameter t is either 0 or 1. Then the number of vertices between t_{min} and t_{max} would affected the rendering time. The rendering time increment for the original model is acceptable when compared to the improvement seen in Fig. 3.11a, b. The comparison is slightly worse for the subdivided models, but it still retains nice real-time properties. It follows the subdivision is useful only if visual quality is a priority. Therefore, it is possible to switch between the original and subdivided mesh in the application.

The theoretical justification of the subdivision step can be found in [15]. It compares the curvature of the deformed skin with and without subdivision, and shows that subdivision really improves smoothness of the resulting shape. Intuitively, it is quite obvious from Fig. 3.11b, c.

Chapter 4

Virtual Humanoid Animation

In chapter 3, we have shown how the problem of virtual humanoid animation can be reduced to the problem of skeleton animation. In this chapter, we address the problem of skeleton animation, i.e. determination of joint transformations $T(j)$ for every joint j, using the notation from section 3.1.

We divide this problem to two subproblems, exploiting the fact that in fencing, the movement of the arm holding weapon and the movement of the legs can be treated separately[1]. Then the only problem is a proper synchronization of both parts, which is fully under the user's control, and therefore can be effectively trained in virtual reality.

For the legs animation is appropriate the interpolation of key frames, because the steps are nothing more than standard routines. The key frame interpolation was implemented pursuant the library [31], although this library offered only a very basic functionality. Some new features had to be appended to enable realistic simulation of fencing steps: support of multiple animations for single model, reverse play and root translation.

The hand animation is far more complicated, because there is a higher degree of freedom in the hand movement. The natural way is to let the user to control the weapon directly (i.e. the sword-holder joint) and propagate the motion automatically to the rest of the arm by inverse kinematics. Both these problems are not trivial. The weapon, being considered a rigid body, has 6 degrees of freedom (DOFs). However, common input devices such as mouse and joystick provide only 2 DOFs. Therefore a mapping from the 2D input device space to the 6D hand motion space must be defined.

The inverse kinematics is a well studied area covering both computer animation and robotics. A nice introduction presents [39], more recent results are compiled in [1]. A practical point of view is studied in [20], [19]. The

[1] Also in real modern fencing the legs and the hand are sometimes trained separately.

reduction of inverse kinematics to optimization problem is presented in [42]. However all these approaches are general and require a numeric solution, with its specific problems. For the purposes of virtual fencing is important the special case of human limb, which was solved analytically [34]. The "analytical" is in the context of inverse kinematics a synonym for "fast and reliable". The authors of [34] have implemented their solution in a library named IKAN. They kindly allowed us to use this library in our virtual fencing project[2].

Although a lot of foreign code and ideas has been used to implement these topic, some original approaches were also applied. It is especially the case of the goal determination (sections 4.2.1 and 4.2.3), and to some extent also the joint limits consideration 4.2.2.

4.1 Key Frame Interpolation

The first problem concerning the key frame animation is to create the key frames. We designed them in the MilkShape3D animation software, because the code for importing its file format is included in [31]. The file format is quite simple: besides the model data, as referred to in chapter 3, it stores exactly one animation sequence.

The basic building block of the animation is the key frame for an individual joint. It is described by the structure `KeyframeInterpolator::KeyFrame`. This structure stores the time when the joint transformation shall take place, the index of the joint and the actual data: a triplet of parameters. These triplet is interpreted either as a translation vector in R^3, or as Euler angles of a 3D rotation (see 2.3). Note that the translation is interesting only for the root joint movement. The joints are animated independently. Each joint has two arrays of key frames: one for translation key frames, second for rotation ones.

The key frame data, together with some additional information such as the total time of sequence, are loaded by class `MilkshapeKeyframeInterpolator`, which is the file-format specific descendant of class `KeyframeInterpolator`.

The actual key frame interpolation is performed by function `KeyframeInterpolator::setTransform`. Its parameter is the current time t_c, for which shall be determined the transformation matrices $T(j)$ (represented by `Joint::m_transform`, according to section 3.1). This function

[2]Unfortunately, the source code of IKAN library is not free for commercial applications. Please consider that IKAN source files are merged with others – see the header of the file if necessary.

assumes that the animation runs considerably faster than the key frames, which is a reasonable presumption. For each joint is stored the index of the rotation key frame i, that was used in the previous call of `setTransform`. The array of rotation key frames is then scanned from index in i, searching for the key frame j such that the time $t(j) \leq t_c \leq t(j+1)$. The rotations given by key frames j and $j+1$ are converted from Euler angles to quaternions and interpolated by SLERP, as described in section 2.6. The interpolation parameter for SLERP is

$$t = \frac{t_c - t(j)}{t(j+1) - t(j)}$$

This yields the resulting transformation of the given joint. The procedure is repeated for all joints. The same stuff is executed for the translation key frames, with the only exception that a simple linear interpolation is performed instead of SLERP.

After computing the $T(j)$ matrices, the movement is propagated to the final transformation matrices (taking into account the relative bone transformations) by function `Model::setFinalMatrices`. This function also handles the explicit root translation, which is necessary because the fencer moves itself by performing the steps.

We observed that the fencing step forward played backwards is exactly the step back. This means that it is not necessary to design new key frames for the step back, if we already have the key frames for the forward step – it is sufficient to reverse the keyframes. This is done by function `KeyframeInterpolator::reverseKeyframeData`.

It would be possible to improve the key frame interpolation from piecewise linear to smooth curves. Splines are usually used for key frame interpolation, and even for quaternions is possible the spherical spline interpolation [9]. However, for the purpose of the fencing steps animation, we found the linear interpolation quite satisfactory.

4.2 Inverse Kinematics of the Human Arm

In this section we briefly describe how does the IKAN system work, and how it is included in our application.

The model of the human arm, according to [34], consists of three joints: the shoulder, elbow and wrist, as illustrated in Fig. 4.1. The shoulder and the wrist are considered to be spherical joints, with full 3 degrees of freedom, which allows them to rotate in an arbitrary way. The elbow is restricted to rotate about its y-axis, which produces one additional degree of freedom.

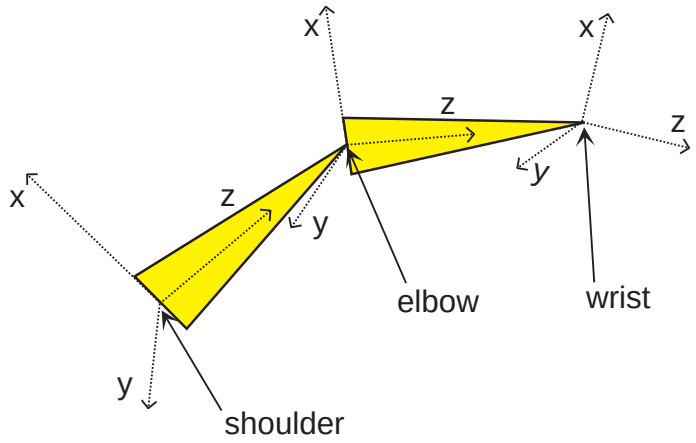


Figure 4.1: The model of the human arm

Together it sums up to 7 DOFs. The goal is to find the rotation in the shoulder, elbow and wrist joints, such that the resulting wrist position and orientation matches the given homogeneous matrix G. Using the inverse kinematics terminology, the wrist is the end-effector and G is the goal. The end-effector has only 6 degrees of freedom (3 for translation, 3 for rotation), thus the problem is under-constrained and there can be either zero, or infinite solutions.

However, as discovered in [34], it is possible to parametrize the space of all solutions in an intuitive way. First, we notice that the angle in the elbow (inclined by the forearm and brachial bone) does not depend on the orientation of the end-effector, and it is determined by the end-effector's position uniquely. If the goal is within reach, then all the solutions can be generated by rotating the arm about the axis connecting shoulder and wrist joints. This axis is called a *swivel axis*, see Fig. 4.2.

The redundant degree of freedom, parametrized by the swivel angle, can be exploited in several ways. First, it is possible to define a position, that the elbow should match as close as possible, and derive the appropriate swivel angle from this condition. Second, we can optimize the "comfortability" of the arm posture by varying the swivel angle, taking the joint limits into account. Both these approaches are implemented in the application.

When the swivel angle and the goal G are given, the IKAN system can very efficiently compute the rotations in the shoulder, elbow and wrist joints. The procedure is described in detail in [34]. Unfortunately, employing the IKAN library is not as straight-forward as it looks like, since it uses completely different conventions than the modeling software used for the virtual

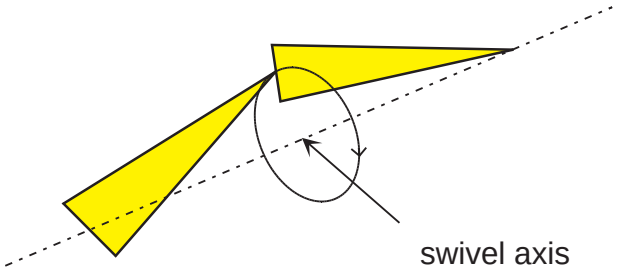


Figure 4.2: The elbow can freely move on a circle

humanoid design and animation. The row-major convention is used for matrix elements storage instead of column-major. Moreover, the local coordinate system in the arm joints in IKAN is different to that one in the Yopago model, see Fig. 4.3.

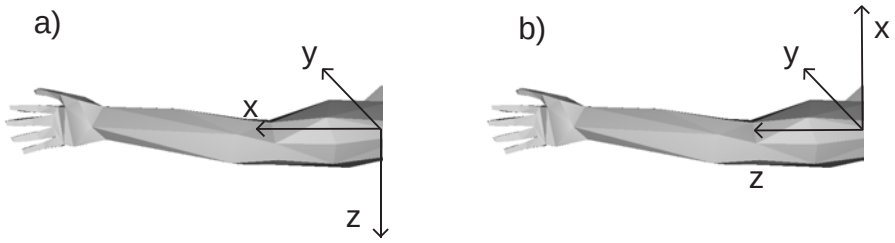


Figure 4.3: Local coordinate system in the right arm for a) Yopago model, b) IKAN convention

The conversion from the Yopago's to the IKAN coordinate system performs matrix T (`my2ikan`), which can be deduced from Fig. 4.3.

$$T = \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The matrix T is orthonormal, therefore the opposite conversion, from IKAN coordinates to Yopago's is given by $T^{-1} = T^T$. Using these matrices we can convert any matrix A expressed in IKAN coordinates to equivalent matrix expressed in Yopago's convention: $T^T A T$. The opposite direction (from Yopago's to IKAN matrix) is analogically $T A T^T$. Besides the coordinate

system transformation it is also necessary to convert from the column-major order to the row-major. This performs the function `Matrix2ikMatrix`. The opposite conversion from the row-major convention to the column-major is done by `ikMatrix2Matrix`.

The interface of the IKAN library to our application is the class `IKAN`. The main function is `IKAN::setGoal` which instructs the inverse kinematics to find a solution for the given goal (position and orientation). In fact, the end-effector used in the virtual fencing is not the wrist joint, but the auxiliary sword-holder joint, introduced in section 3.4. This is due to the fact that the user controls the weapon, not the wrist. Fortunately, this another joint does not participate actively in the inverse kinematics chain, since it is rigidly linked with the wrist[3]. Thus it suffices to transform the goal G from the sword-holder placement to the wrist placement, which is done by multiplication $GR(s)^{-1}$, if s is the index of the sword-holder joint, and $R(s)$ is the relative matrix from section 3.1. This is implemented in function `IKAN::SHGoal2HandGoal`.

4.2.1 Goal Determination

As mentioned above, the problem of determination the end-effector's goal (i.e. the position and orientation) is not trivial, if only standard, 2D input peripherals are assumed[4]. We solved this problem by key frame interpolation again. However here, the interpolation is not done on all the joints, as in section 4.1, but only on the end-effector. It is the task of the inverse kinematics to compute the rotations in the arm joints.

The problem is to find a mapping from a bounded subspace of R^2, which corresponds to the input device's range, to a bounded subspace of R^6, which corresponds to the reachable goals. Without loss of generality we assume the space of the input device is $I = \langle -1, 1\rangle \times \langle -1, 1\rangle$. The idea is to define several points in I and associate them with the goal key frames. Then any other goal can be computed by interpolation from the key frames.

We defined three sets of key frames: one corresponding to the swing movements (the preparation for an attack). Second stands for the attack postures and the third is for parry. The switching between these sets is a task of user control, which is explained in section 4.3. There are 9 key frames for the swing and attack sets, and 15 for the parries, summing to 33 end-effector positions and orientations. Almost each of them corresponds to some

[3]It corresponds to ideal, firm holding of the weapon. Loosely held weapon could be simulated by movement of the sword-holder, but this was not implemented.

[4]However, the problem is trivial using high-level hardware, such as a *tracker*, which provides 6 or even more DOFs.

well-defined fencing posture. These sword-holder positions and orientations for every key frame were designed in CharacterFX. The reason for defining more parry positions is to distinguish the sixte and quinte parry, as illustrated in Fig. 1.1. The distribution of the key frame points in I forms square or rectangular tiles, as depicted in Fig. 4.4.

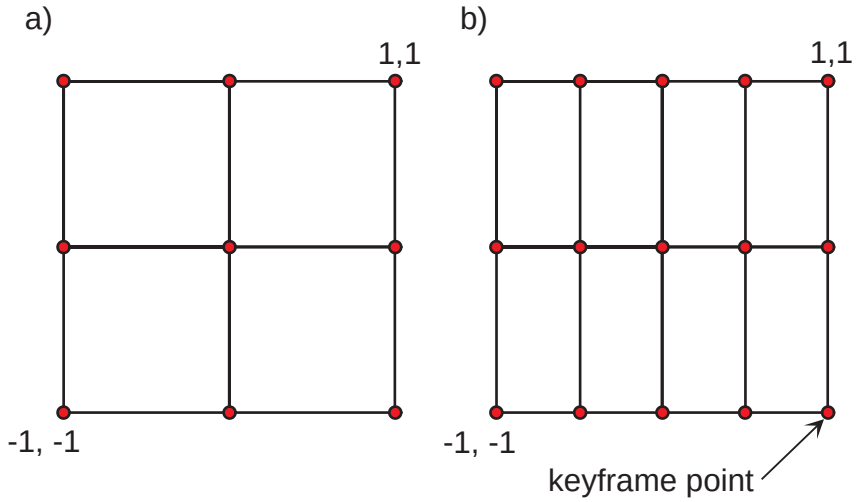


Figure 4.4: Key frames distribution: a) swing and attack key frames, b) parry

If we are given any position $(x, y) \in I$, we can find the tile containing (x, y), according to Fig. 4.4. The final end-effector position and orientation can then be computed by bilinear interpolation of four key frame values in the corners, see Fig. 4.5.

The interpolation of homogeneous matrices means the SLERP (see section 2.6) for the rotation part and simple linear interpolation for the translation vector. We can denote the interpolation of homogeneous matrices P, Q with parameter $t \in \langle 0, 1 \rangle$ as $i(t, P, Q)$. Assuming the box from Fig. 4.5 has lengths x_s, y_s, we can write the resulting matrix as

$$i\left(\frac{\Delta x}{x_s}, i(\frac{\Delta y}{y_s}, A, B), i(\frac{\Delta y}{y_s}, C, D)\right)$$

This computation is performed by class `KeyframeMID`[5]. One key frame stored in a MilkShape3D file can be loaded by function `KeyframeMID::loadKeyframe`. However, individual loading of all key frames

[5]The `MID` is due to the historical reasons a shorthand for Mouse Input Device.

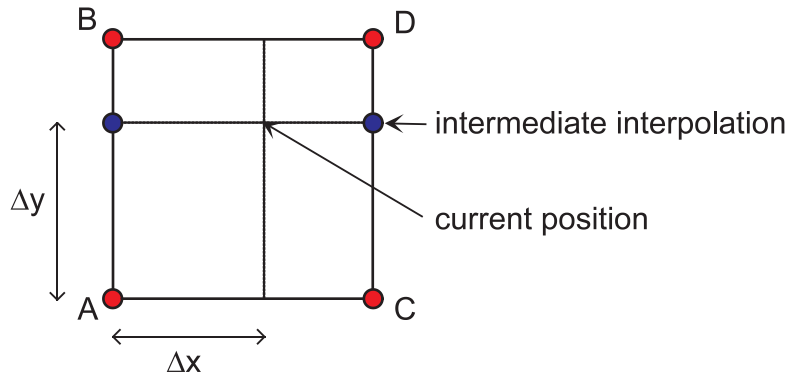


Figure 4.5: Bilinear interpolation of key frame homogenous matrices A, B, C, D

in this way would be inefficient, thus there is a possibility to load all key frames at once using `KeyframeMID::loadAllKeyframes`. The interpolated end-effector position and orientation is computed by function `KeyframeMID::getGoal`.

4.2.2 Joint Limits

For producing plausible solutions of the inverse kinematics problem, it is necessary to consider the joints range of motion. IKAN itself offers only a basic support of joint limits, based on the Euler angles limitation. This has not been found to be satisfactory, according to our experiments, as well as [1]. However, the IKAN's parametrization of the space of solutions by the swivel angle lends itself to an optimization procedure, since the space of solutions has only one dimension. The joint ranges handling, that is used in the virtual fencing application, is based on the ideas from [1]. In spite of this, the algorithm for the swing and twist extraction is original, as well as the optimization of comfortability.

The rotation in the elbow joint (with only one degree of freedom, i.e. about a fixed axis) is given only by the end-effector's position, as stated above. Thus it is meaningless to impose a limit on the elbow joint, since changing the swivel angle does not change the angle in the elbow. The spherical joints (shoulder and wrist) control an arbitrary 3D rotation, whose limitation is somewhat more complicated. For the description of the spherical joints range is useful the *swing and twist* decomposition of the orientation. It is quite intuitive: the swing describes the direction, which can be given by

a unit 3D vector, thus removing two DOFs. What remains is the twist – the rotation about the direction vector.

When given a rotation matrix R, we can extract the swing and twist component of the rotated z vector (according to the IKAN convention of the limb coordinate system). The swing given by angles α and β is extracted first, according to Fig. 4.6. In the picture the rotated z vector is denoted by $r = Rz$. Consulting Fig. 4.6, the swing angles can be computed as

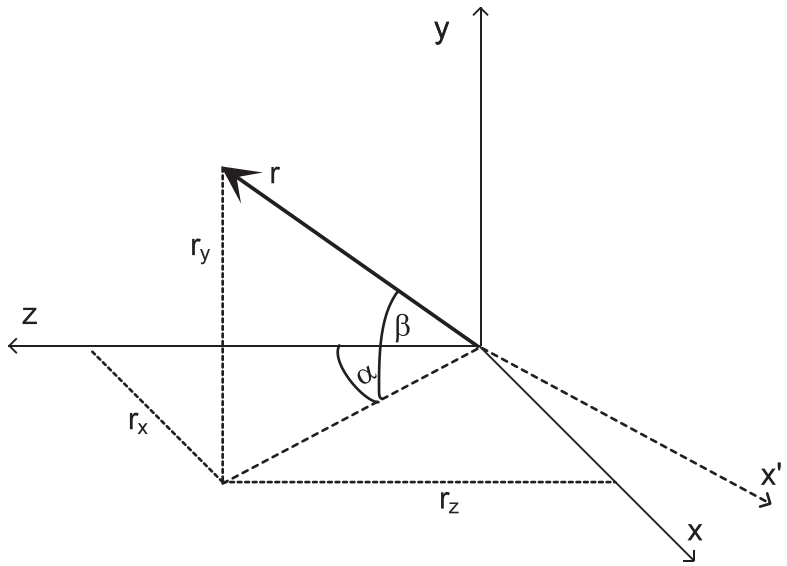


Figure 4.6: Determination of swing components α, β from vector r.

$$\alpha = atan2(r_x, r_z),\ \beta = asin(r_y/\|r\|)$$

where $\alpha \in \langle -\pi, \pi \rangle$, $\beta \in \langle -\pi/2, \pi/2 \rangle$.

Having the swing computed, we can proceed to the twist determination. Define the rotation S that aligns z with r without any twist, as the rotation with angle β about the rotated x axis

$$S = R_{x',\beta},\ x' = R_{y,\alpha}x$$

The S can be regarded as a reference zero twist rotation, and we can deduce the twist from respective rotation of S and R, since the swing is the same. Taking for instance vector y, we can say that the twist is the angle inclined by vectors Sy and Ry, and can be computed as

$$\tau = acos((Sy)(Ry))$$

yielding $\tau \in \langle 0, \pi \rangle$, which is sufficient for symmetric twist limit. This procedure is implemented by function `getSwingTwist`.

Using the swing and twist decomposition, the joint range can be given by bounding the swing, i.e. defining a valid surface on a unit 3D sphere. Then for each point within the swing boundary the twist limit can be imposed. However, it is often sufficient to assume a constant twist limit for all directions, as remarked by [1]. The twist can be bounded easily, since it has only one dimension. Somewhat more interesting is the swing boundary, which corresponds to certain subset of a unit 3D sphere. First realize that the sphere can be parametrized by the swing angles $\alpha \in \langle -\pi, \pi \rangle$ and $\beta \in \langle -\pi/2, \pi/2 \rangle$. A straightforward boundary could be restriction of α, β to some subintervals, which would lead to a spherical rectangle. However, such border does not approximate the joint range well: it is not smooth and the α and β limits are independent.

A better idea is to use a *spherical ellipse*. Here we also restrict α and β to certain subinterval, say $\alpha \in \langle \alpha_0, \alpha_1 \rangle$, $\beta \in \langle \beta_0, \beta_1 \rangle$, but the valid swing is only the α, β pair satisfying $f(\alpha, \beta) < 0$, where

$$f(\alpha, \beta) = \left(\frac{\alpha - (\alpha_1 + \alpha_0)/2}{\alpha_1 - \alpha_0} \right)^2 + \left(\frac{\beta - (\beta_1 + \beta_0)/2}{\beta_1 - \beta_0} \right)^2 - 1$$

The equation $f(\alpha, \beta) = 0$ describes the spherical ellipse, see Fig. 4.7. Even

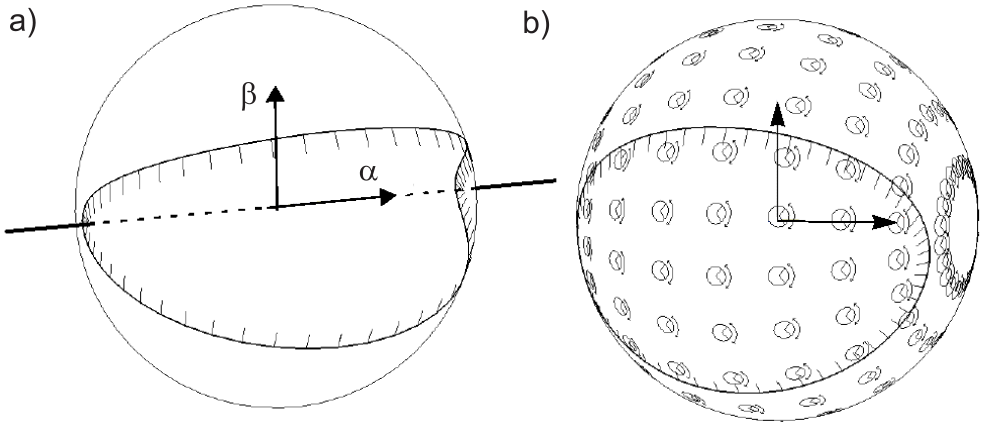


Figure 4.7: a) The spherical ellipse, b) with twist limits (from [1])

more sophisticated swing boundaries are described in [1], for example the spherical polygon boundary.

The parameters $\alpha_0, \alpha_1, \beta_0, \beta_1$, and the twist limit were tuned for both the shoulder and wrist joints by hand (they can be found in the constructor `IKAN::IKAN`). The above ideas represent a tool for determining whether a given rotation matches the joint limits. Nevertheless, we can exploit the concept of spherical ellipse even more, if we realize that the function $f(\alpha, \beta)$ in fact measures the deflection from the zero rotation. This measure is influenced by the swing ranges $\alpha_0, \alpha_1, \beta_0, \beta_1$, and can be interpreted as a measure of *comfortability* of the rotation, using the assumption that $\alpha = \beta = 0$ is the most comfortable position. The comfortability function (`IKAN::comfort`) that is used by the application is

$$c(\alpha, \beta, \theta) = f(\alpha, \beta) + \theta$$

assuming the angle θ is given in radians. Of course it would be possible to tune this function to produce better results (for example using a coefficient to scale θ, it is indeed a fortune that simple expression in radians works).

The comfortability is higher when the value of function c is lower, therefore optimization of the joint rotation is equivalent to minimization of c. Although c is a function of three variables[6], there is only one independent parameter due to the end-effector constraint: the swivel angle.

The procedure that optimizes the comfortability, involved in function `IKAN::pomSetGoal`, is very simple. For each swivel angle equal to $1, 2, \ldots, 360$ degrees, it calls the inverse kinematics, converts the resulting shoulder rotation to the swing and twist representation and stores the swivel angle corresponding to the minimal c. We can afford this due to two things: firstly, the IKAN code is very efficient, and the one degree precision is sufficient. Secondly, the optimization method is not appropriate for interactive posturing anyway. In majority of situations it is replaced by another method, described in section 4.2.3.

For given goals, the comfortability optimization produces plausible results indeed. The problem is that the resulting values of swivel angle are not continuous: a small change in the end-effector may result in a big difference in the swivel angle. This results in a sudden flounce of the elbow, which is somewhat annoying. Thus the optimization approach for swivel angle determination is used only in special situations, where the discontinuity does not matter, see section 6.5.1.

[6]Similar function can be defined also for the wrist joint, but the wrist joint range is neglected by the application. Experiments have shown that it is the shoulder joint, which has the major influence on the plausibility of a posture.

4.2.3 Elbow Positioning

As mentioned above, we would like to have a continuous function computing the swivel angle from given end-effector position and orientation. To achieve this, we can exploit the goal key frame interpolation scheme, that is described in section 4.2.1. The idea is to append the desired position of elbow to the already defined key frame values. We use the fact that the swivel angle is determined uniquely by the desired elbow position. Put in another words, requiring the elbow to meet some desired position as close as possible leads to unique swivel angle. The swivel angle matching this condition can be computed efficiently using an IKAN function. This function uses an analytical expression (no optimization), which is explained in [34].

We interpolate the elbow position in the same way as the end-effector position, as described in section 4.2.1. Fortunately, the interpolated elbow positions represent suitable values, leading to plausible swivel angles. The bilinear interpolation produces a continuous function (although not smooth), and it is obvious that a small change in the elbow position results only in a small difference in the swivel angle[7]. This algorithm is also very fast, thus it is used in the application for the swivel angle determination. The only drawback is that for every key frame must be defined the elbow position in addition to the end-effector's position and orientation. The optimal elbow positions (in terms of visual plausibility) were tuned in especially modified version of the application.

We have also experimented with direct interpolation of the swivel angle, but the results of arm posturing were worse. It would be possible to improve the elbow position interpolation scheme (as well as the end-effector position and orientation interpolation) to higher-order interpolation to achieve smooth functions.

4.3 User Interface

As mentioned above, a 2D input device such as a mouse or joystick is used to control the hand. The switching between the parry, swing and cut key frame sets is done by the mouse or joystick[8] buttons. When no mouse button is pressed, the swing key positions are used. When the user clicks the left mouse button, then a successive interpolation to the corresponding sword-holder position and orientation in the cut set of keyframes is performed.

[7] A formal proof would be possible using the formulas for projection of the elbow position to the swivel angle, which are derived in [34].

[8] We use the terms *left* and *right* button even for joystick buttons. It can be dynamically configured which joystick button is *left* and *right*.

To enforce more realistic fencing, a condition of big swing must be met before the avatar switches to cut. Recall that the 2D input device range is $\langle -1, 1\rangle \times \langle -1, 1\rangle$. However, in $\langle -0.8, 0.8\rangle \times \langle -0.8, 0.8\rangle$, the cut movement can not be entered. It can be entered only outside this interval, although the cut can then proceed inside $\langle -0.8, 0.8\rangle \times \langle -0.8, 0.8\rangle$ later. It is also not possible to hold the cut position for an infinite amount of time – after certain time the swing position is entered back automatically. We must also handle some marginal situations, like that the users releases the mouse button before the interpolation to another set has finished etc.

The keyboard is used to control the leg movement. In fact, the problems are quite similar to the case of the hand. The steps must be executed completely (it is not possible to change the intended movement in the middle of a motion), but in a lunge the fencer can stay as long as he wants.

Actually, we can consider the keyboard and mouse (or joystick) events together. Let us define the *user events* as the mouse movement, mouse clicks, and key strokes (both press and release). The *input to the key frame interpolator* are the interpolated key frames and the interpolation parameter. Then the relationship between the user events and the input to the key frame interpolator can be described by a state diagram. Its nodes correspond to states, such as a stand, guard, step forward etc. The edges represent a possibility of switching to another state, if some condition is met. This condition can be either a user or timer event. An example of the state diagram controlling the legs movement is presented in Fig. 4.8. The keys N,H,J,K are the defaults, and can be re-assigned dynamically.

A similar diagram can be drawn also for the hand states, but it would be somewhat more complicated – there is more in-between states (such as cut-parry etc.). It is wired in the implementation class `ActionMID`. The functions `ActionMID::feedButton` and `ActionMID::feedNewPos` inform the object about the mouse user events. The function `ActionMID::getGoal` is used to retrieve the actual interpolated position and orientation of the sword-holder. The keyboard processing is done in class `KeyboardInputDevice` with similar functionality as `ActionMID`.

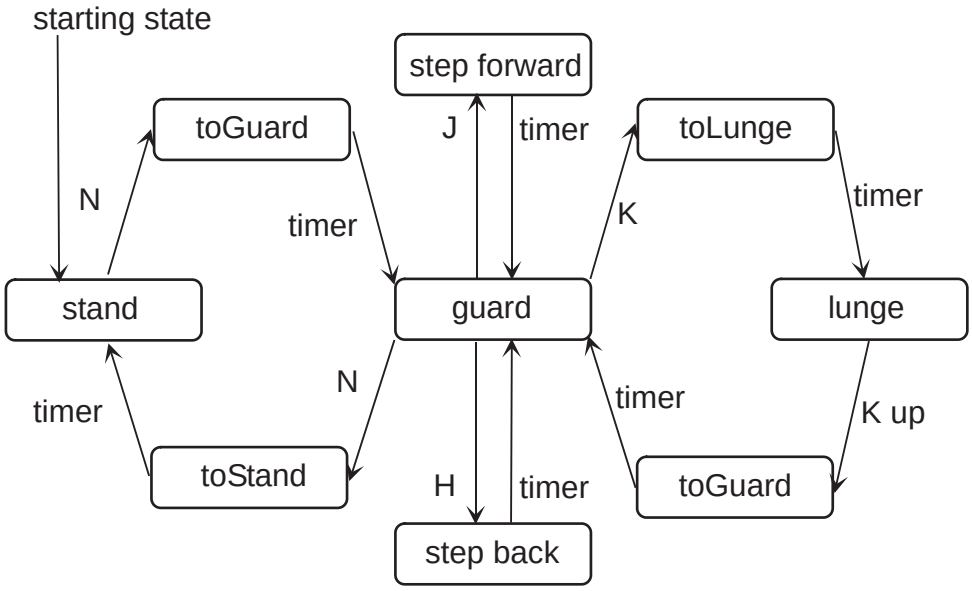


Figure 4.8: The legs movements diagram

Chapter 5

Collision Detection

As one would expect, the object collisions are very important in fencing. By objects we mean either the virtual humanoid body, or the weapon. Both these objects have its specific properties. The weapon is rather thin and moving fast, thus a collision can be missed easily. The advantage of the weapon is that it can be considered as a rigid body, because it is not deformed during the simulation[1].

The virtual fencer's body, on the other hand, can not be considered rigid. Its skin undergoes a lot of deformations resulting from motion, which was discussed in chapter 3. However, the virtual humanoid body is relatively large, and slowly moving, when compared to the weapon.

Any two of these objects can collide each other, which leads to following possibilities

- weapon-weapon collision – a very common situation in fencing. A sophisticated collision response is necessary.
- weapon-body collision – the fencer is hit. It is necessary to determine which part of the body was hit (for example in order to consider the legal hit area, as explained in section 1.2).
- body-body collision – a rare situation, which is usually not legal in the real fencing, since fencing is not a wrestling. Nevertheless, it must be considered, since the penetration of fencers bodies can not be allowed.

The collision detection (CD) itself can be divided into two large areas: the *static CD* and the *dynamic CD*. The static CD gets two triangular meshes M_0, M_1 on the input, and its task is to produce all pairs of triangles $(u_0, u_1) \in$

[1] Another advantage of historical weapons, such as sword: little flexibility.

$M_0 \times M_1$ such that u_0 intersects u_1. Sometimes it is required only to check whether any such pair exists.

The dynamic CD solves the same task, but considering that the triangular meshes are moving in time. It answers whether any two triangles $u_0 \in M_0$, $u_1 \in M_1$ intersect during some time interval $\langle t_0, t_1 \rangle$. The difficulty of the dynamic CD strongly depends on the allowed type of movement: linear (pure translation), rotation about a fixed axis, or both together.

There is an abundant number of articles concerning the static CD. We have focused on the algorithms based on a bounding volumes hierarchy. They have generally the same core, but the difference is in the used bounding volumes, which determine the effectiveness. The simplest and oldest are the sphere trees, but they have not said their final word yet [7]. Somewhat more interesting are the Axis-Aligned Bounding Boxes (AABB) trees, which may seem also obsolete, but it was found recently that they have nice properties for deformable models [36], which is exactly what we need for the humanoid body. Contemporary favorites are Oriented Bounding Boxes (OBB), presented in [11], competing with Discrete Oriented Polytopes (k-DOP) examined by [17], [18]. A method for dynamical alignment of DOP-trees for rotated objects is presented in [41].

Concerning a dynamic CD, only a few resources have been found. A linear movement is considered in [9]. The general motion problem can be reduced to a root-finding problem, as described in [10], and solved by numeric methods (namely regula falsi). Recently, a survey about both static and dynamic CD was found [13].

We implement the static CD pursuant the ideas of [36], and generalizing the approach to k-DOPs (from original AABBs). The dynamic collision detector we built from scratch, using some simplifications resulting from the special shape of the weapon. The method we use is described in [14].

5.1 Static Collision Detection

The input are two meshes of triangles M_0, M_1 (sometimes called triangular soups, to indicate that there is present no topological structure). We need to check if there is $u_0 \in M_0$ and $u_1 \in M_1$ such that $u_0 \cap u_1 \neq 0$. A naive static CD algorithm could test each pair from $M_0 \times M_1$. Such algorithm would have time complexity $O(|M_0||M_1|)$, which is unbearable even for models with small number of triangles (thousands), because the collision detection must be performed in real-time.

It is possible to speed up the CD using an additional structure – the hierarchy of bounding volumes. The *bounding volume* for a set of triangles T

is denoted by $B(T)$ and it is defined as a convex subset of R^3 such that $\forall t \in T : t \subseteq B(T)$. Obviously, the smallest bounding volume would be the convex hull. In practice, only approximations of convex hull are used: sphere, AABB, OBB, k-DOP. The AABB is a box with edges parallel to the coordinate axes of some fixed coordinate system (usually the world coordinate system). The OBB is an arbitrarily oriented box. The k-DOP, k even, is a convex polytope, whose facet's normals come from a fixed set of $k/2$ orientations. Note that if the first three orientations correspond to the coordinate axes, then the 6-DOP is equivalent to AABB. The comparison of these three types of bounding boxes in 2D is in Fig. 5.1.

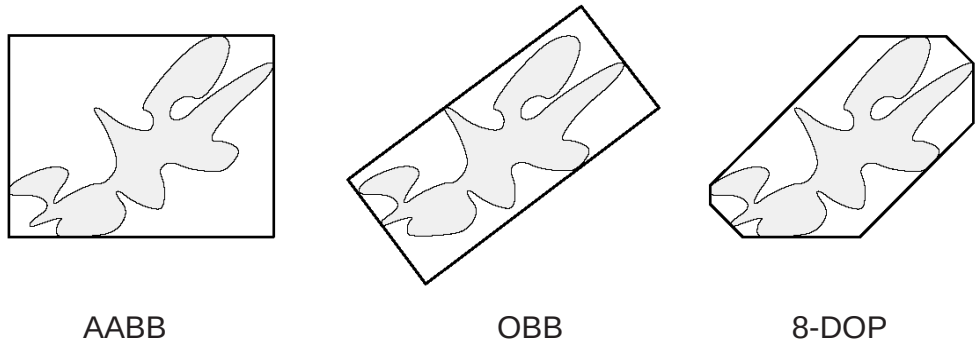


Figure 5.1: Comparison of bounding boxes (from [17])

The pre-processing step of a CD algorithm is to construct a tree of chosen type of bounding volumes for a given triangular mesh M. Each node of the tree corresponds to a set of triangles, for example the root r corresponds to the whole M. We can extend the definition of operator B even for a node u of a tree: $B(u)$ shall be interpreted as a bounding volume corresponding to the set of triangles given by node u. We assume only binary trees. Let us denote the left child of node u as $l(u)$ and the right child by $r(u)$.

The construction of the tree can proceed in a top-down manner, according to the algorithm 1. This algorithm is implemented in function `CDModel::build`.

Some points of the algorithm 1 depend on the actual type of the applied bounding volumes. For the case of AABBs, the determination of the smallest AABB containing M (step (2)) is a straightforward maximization of the triangle coordinates in the e_0, e_1, and e_2 component. For k-DOPs, realize that the projection of a vertex v into arbitrary direction vector d can be computed by the dot product dv (in fact selecting the i-th component for AABBs can be also treated as a dot product with vector e_i). Thus for each

Algorithm 1: Building bounding volume tree
Input: The triangular mesh M
Output: The root p of the created tree
BUILDTREE(M)
(1) create node p
(2) $B(p) = B(M)$
(3) split M according to some splitting rule into M_l and M_r
(4) **if** $M_l \neq 0$ **then** $l(p) =$ BUILDTREE(M_l)
(5) **else** $l(p) =$ nil
(6) **if** $M_r \neq 0$ **then** $r(p) =$ BUILDTREE(M_r)
(7) **else** $r(p) =$ nil
(8) **return** p

of the $k/2$ directions, the maximum and minimum of vertex projections to the given axis is computed, analogically to the AABB case.

The splitting rule, mentioned in step (3) of algorithm 1, also depends on the type of bounding volumes. For AABBs, it is recommended in [36] to divide the longest axis of $B(M)$ to produce two AABBs with equal volume. Other splittings, for example selecting the dividing plane which produces AABBs with almost equal number of triangles (thus leading to a balanced tree), are reported to have worse performance. When the splitting of the longest axis would produce a leaf node with more than one triangle, other axes are tested.

In algorithm 1, we did not address the implementation of the splitting of M into two sets of triangles. The triangular mesh M is represented by an array of triangle indices. The function `CDModel::prohazej` implements the splitting: its output is a number k and an array with swapped elements, in such way that the indices $0, \ldots, k-1$ correspond to triangles in M_l, and $k, \ldots, |M|$ correspond to triangles in M_r. The splitting is quite similar to the swapping step of the well-known Quicksort algorithm. The nice property is that the following recursive splitting of M_l works only on elements $0, \ldots, k-1$.

The general collision detection algorithm, exploiting a bounding volumes tree, is independent of the actual bounding volumes type. This is the algorithm 2, implemented in function `CDModel::collide`.

The function *leaf* used by the algorithm simply checks if its parameter is a leaf node. The function *disjoint* tests whether two bounding volumes are disjoint. However, the only condition we pose on the function *disjoint* is, that if it returns yes, then the bounding volumes are really disjoint. On the other hand it is acceptable that the function returns no, even for bounding

Algorithm 2: Collision detection using bounding volume trees
Input: The roots r_0, r_1 of bounding volumes trees of two objects
Output: Yes/no answer if the meshes represented by r_0, r_1 collide
colTest(r_0, r_1)
(1) **if** disjoint($B(r_0), B(r_1)$)
(2) **return** NO
(3) **else**
(4) **if** (leaf(r_0) **and** leaf(r_1))
(5) check all triangles from r_0 against all triangles from r_1
(6) **return** YES iff any pair intersected
(7) **else if** (leaf(r_0) **and not** leaf(r_1))
(8) **return** colTest(r_0, $l(r_1)$) **or** colTest(r_0, $r(r_1)$)
(9) **else if** (**not** leaf(r_0) **and** leaf(r_1))
(10) **return** colTest($l(r_0)$, r_1) **or** colTest($r(r_0)$, r_1)
(11) **else**
(12) r_i = the node with larger volume of $B(r_i)$
(13) **return** colTest($l(r_i)$, r_{1-i}) **or** colTest($r(r_i)$, r_{1-i})

volumes that are disjoint. This leads only to some redundant tests.

Recall that an AABB (6-DOP) is given by three intervals, corresponding to ranges in the e_0, e_1, and e_2 coordinate. If the intervals for at least one axis are disjoint, it implies that the AABBs are disjoint. Conversely, if the intervals are overlapping for all three axes, then the AABBs are also overlapping. Thus for the case of AABBs we have a perfect disjointness test.

The overlap test for k-DOPs works in the same way, but with $k/2$ axes (instead of 3). It still holds that if the intervals for at least one axis are disjoint, then the whole k-DOPs are disjoint. However, the opposite direction is no more valid: if all the $k/2$ intervals of two k-DOPs are overlapping, the k-DOPs may be disjoint. The reason for this is connected to the separating axis theorem; it is somewhat beyond the scope of this work, please consult [18] or [11] for further discussion. The important fact is that the test of only $k/2$ directions, called a *conservative* test, can still be used as a *disjoint* function. The conservative test is even believed to have better overall performance than the perfect disjointness test.

The function for robust and optimized triangle-triangle intersection test has been borrowed from the RAPID library [11] (the file `TriTriTest.cpp`). The algorithm 2 can be easily extended to output all pairs of colliding triangles. It suffices to return the set of colliding triangles in step (6) and replace the boolean **or** operators with set **union** operators. This is implemented in

function `CDModel::colTri`, where the set is represented by an array.

5.1.1 Deformable Objects

As mentioned above, the main problem of the collision detection with a virtual humanoid body is that the shape of the body is not constant. A movement such as translation and rotation can be propagated to bounding volumes relatively easily, but the deformations of the virtual humanoid skin are not so trivial (even in the basic skeletal animation). A re-build of the bounding volumes hierarchy, according to algorithm 1, would be awfully inefficient, since the skin's shape changes almost each frame of the simulation.

Fortunately, there has been published an article [36], describing how it is possible to quickly refit the AABB trees for a model that undergoes deformations. The *deformation* of a model in this context means an arbitrary shift of the vertices (possibly producing even a completely different shape). The main property, that allows a fast refit of the AABBs is following: let T_0, T_1 be two sets of triangles, and let $B_{AA}(X)$ denote the smallest AABB bounding X. Then

$$B_{AA}(T_0 \cup T_1) = B_{AA}(B_{AA}(T_0) \cup B_{AA}(T_1)) \tag{5.1}$$

In fact, this property generalizes to k-DOPs, but not to OBBs. The counter-example for OBBs presents Fig. 5.2, where the smallest OBB bounding X is denoted by $B_O(X)$. For k-DOPs, equation (5.1) justifies the bottom-up refitting algorithm 3.

Algorithm 3: k-DOP tree refit
Input: The root p of a k-DOP tree
Output: Corrected k-DOP sizes
REFIT(p)
(1) **if** (l(p) = nil **and** r(p) = nil)
(2) compute $B(p)$ directly from the triangles
(3) **else if** (l(p) $\neq$ nil **and** r(p) = nil)
(4) $B(p) = B(l(p))$
(5) **else if** (l(p) = nil **and** r(p) $\neq$ nil)
(6) $B(p) = B(r(p))$
(7) **else**
(8) $B(p) = B(B(l(p)) \cup B(r(p))$

In particular, the property (5.1) is exploited in step 8 of the refitting algorithm. It simplifies the algorithm considerably, because it is much faster

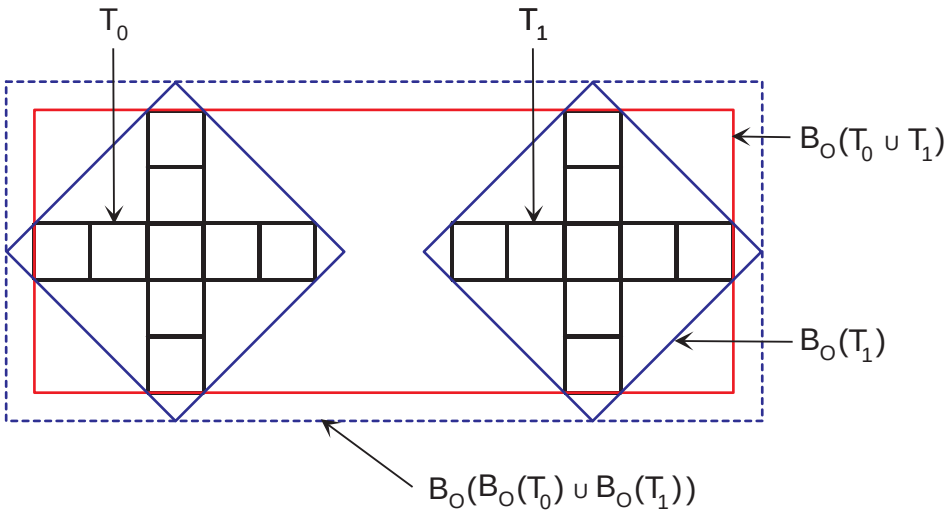


Figure 5.2: $B_O(T_0 \cup T_1) \neq B_O(B_O(T_0) \cup B_O(T_1))$

to compute a k-DOP that bounds two other k-DOPs than a general set of triangles. This reduces the time complexity in the worst case from $O(n^2)$ for the tree build (algorithm 1) to $O(n)$ of algorithm 3, where n is the number of triangles. This algorithm is implemented by function `CDModel::refit`.

5.1.2 Performance and Comparison

The used splitting rule (algorithm 1) does not guarantee that the leaves will have a bounded number of triangles. However, in practice it works well for reasonable triangular meshes: the Yopago model has 1372 triangles. The resulting AABB tree (for a given orientation) has 1195 leaf nodes, which gives an average number of 1.15 triangles per leaf node. The maximum number of triangles in one leaf is 4. Thus the test of all triangles from one leaf against all triangles from the other leaf (step (5) of algorithm 2) is expected to test 1.32 triangle pairs in average, and 16 in maximum.

The resulting axis-aligned bounding boxes of the leave nodes are illustrated in Fig. 5.3 (general k-DOPs visualization has not been implemented). Note that the weapon is not included in the AABB hierarchy, since its collisions are handled in a different way.

When comparing the performance of static collision detection algorithms based on bounding volumes hierarchy, there is always a trade-off between the tightly bounding objects (good approximation of convex hull) and the time for the overlap test of two bounding volumes. Because we are considering

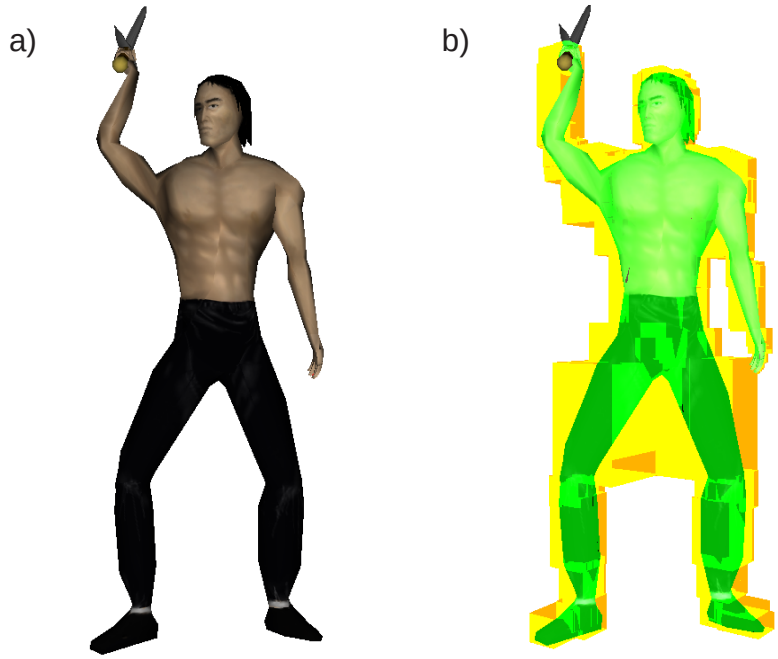


Figure 5.3: a) Yopago in a guard position, b) AABB of the leaves

deformable models, we can not just adopt the results from [18], since we must account also for the refitting time. The refitting procedure, as well as the collision detection itself must be executed each frame of the simulation. We are especially interested in the question which k should be chosen as the order of the used k-DOPs.

For the experiment, we measured the collision detection of two virtual humanoid models in an adjusted version of the application. It is obvious, that if the objects are far apart, then the time of the actual collision detection is negligible – the algorithm 2 often ends already in the first iteration. Thus the majority of time is spent in the refitting algorithm 3.

Somewhat more interesting is the case when the objects are in close proximity. We have executed two tests: the first assumes the virtual humanoids in close proximity, but not colliding, see table 5.1. The second one pushes the models in even closer proximity, allowing collisions, the results are in table 5.2. Note that the full collision test, i.e. producing all colliding triangles, was measured (not only the boolean collision query). This is because we want to highlight the colliding triangles in the application, in order to provide feed-back for the user.

We did not implement the 8-DOP (as well as [18]), since we exploited

	6-DOP	14-DOP	18-DOP	26-DOP
collision test	1.17	0.87	0.96	0.97
refit (2 models)	8.29	12.76	14.63	17.66
total	9.46	13.63	15.59	18.63

Table 5.1: Average time for close proximity without collisions (in ms)

	6-DOP	14-DOP	18-DOP	26-DOP
collision test	11.63	9.59	9.88	9.81
refit (2 models)	8.32	12.23	14.05	17.19
total	19.95	21.82	23.93	27

Table 5.2: Average time for close proximity with collisions (in ms)

the fact that the 6-DOP's directions (world coordinate axes) are contained in each of the 14, 18 and 26-DOPs. From the tables we see, that the best in the collision time is the 14-DOP. Nevertheless, it does not perform notably better than any other k-DOP. In fact, the order of the k-DOPs does not influence the collision query time considerably. On the ohter hand, the order of the k-DOP has relatively big impact on the refitting time. Recall that the refit time is quite important, because the refitting algorithm has to be executed each frame of simulation.

Let us return to the original question – which k-DOP should be chosen as our bounding volume. Consulting the tables 5.1 and 5.2, we see no reason for high k. The 6-DOP performs almost as well as higher order DOPs in collision time, and it is much faster in the refit time. This justifies that in [36], only AABBs are considered (although the comparison with higher order DOPs is not discussed there).

Due to these reasons, we concluded to employ the 6-DOP as the default. However, it is possible to select any $k \in \{6, 14, 18, 26\}$ by passing it to the constructor `CDModel::CDModel`. In this function can be also found the axes directions for each supported k.

5.2 Dynamic Collision Detection

The problem of the static CD, executed each frame of the simulation is, that it does not consider the motion between the previous and the current frame. Due to this, a quickly moving bullet can skip a wall, without any collision being detected. This is demonstrated in Fig. 5.4.

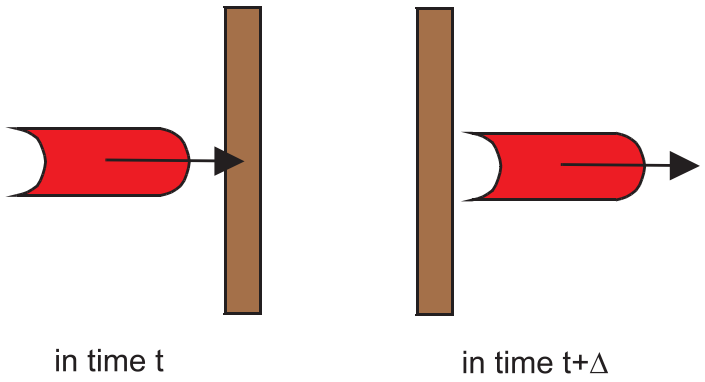


Figure 5.4: No collision is reported, although the bullet has penetrated the wall.

The static CD algorithms presented in section 5.1 are so far appropriate only for body to body test, since the virtual humanoid bodies are moving relatively slowly with respect to the simulation speed (FPS). However, when considering the weapon, the dynamization of the collision detection must be taken into account: it would be disappointing if the weapon could pass the opponent's body without collision. In spite of the importance of the dynamic CD, all the algorithms presented below are not perfect: they provide only an approximation – such that is sufficient in given situation.

The case of weapon to body collisions is somewhat easier than the case weapon to weapon, so let us examine it first. In fact, we apply a simple trick to approximate the dynamic CD for the weapon-body problem, inspired by the idiom "it is not the bullet what kills, it is the hole". Define the *fade* of the weapon as a structure consisting of two triangles, that connects the blade of the weapon in the previous frame with the blade of the weapon in the actual frame, see Fig. 5.5.

The fade of the weapon is then used for static CD instead of the weapon model itself. Of course, it is only a rough approximation of the actual swept volume of the blade. However this approximation is quite appropriate for the weapon-body collision detection[2]. An advantage is that the CD based on the weapon's fade is very fast, since the fading structure consists of only two triangles.

[2]The hits missed due to the approximation were not worth scoring anyway. Only a correct cut, as defined in section 1.2, is discovered using the weapon's fade.

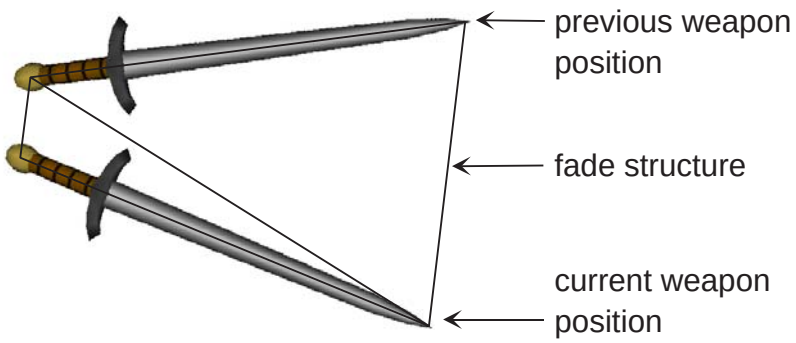


Figure 5.5: Two fading triangles connect the previous position of the weapon with the current one.

5.2.1 Weapon to Weapon Collisions

Unfortunately, the weapon's fade approximation is not applicable to weapon-weapon collisions due to the fact that both the objects have to be "dynamized", i.e two fades should be considered. However, if two fades intersect, we can not conclude that the objects have really collided[3]. This is because the intersection of swept volumes does not imply a collision [13].

To overcome this problem, we have implemented more precise method of dynamic CD for weapon-weapon collisions. It exploits the special shape of the weapon, that can be approximated by a simple object. The weapon is, for the purposes of dynamic collision detection with another weapon (and also for collision response, that is described later), approximated by a *capsule*. A capsule is a point set given by a line segment $\overline{AB}$ and a radius r

$$\left\{x : dist(\overline{AB}, x) \leq r\right\}$$

where *dist* denotes the point to segment distance. For an example of a capsule see Fig. 5.6a. The shape of the weapon is approximated by two capsules, one for the blade, the other for the guard, see Fig. 5.6b.

A capsule is similar to cylinder, but has certain advantages. A test if a capsule given by $(\overline{A_0B_0}, r_0)$ intersects another capsule given by $(\overline{A_1B_1}, r_1)$ is simply

$$dist(\overline{A_0B_0}, \overline{A_1B_1}) \leq r_0 + r_1$$

and the segment to segment distance can be computed efficiently. This allows very fast collision test of two capsules. We refer to an object composed of two

[3]We can only say that if they do not intersect, then the objects do not collide.

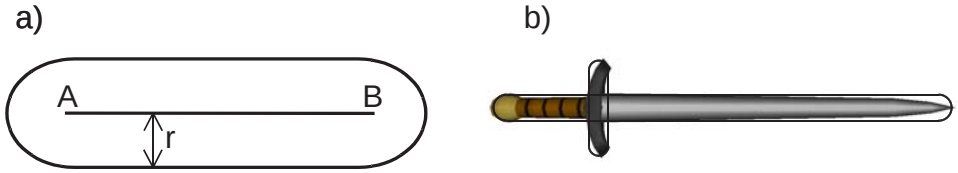


Figure 5.6: a) Capsule given by a line segment $\overline{AB}$ and radius r, b) approximation of the weapon by two capsules

capsules as a *capsuloid*, and the two line segments defining it as its *skeleton*. The distance of capsuloid given by line segments $\overline{A_0B_0}, \overline{C_0D_0}$ to capsuloid given by segments $\overline{A_1B_1}, \overline{C_1D_1}$ is defined as

$$\min\{dist(\overline{A_0B_0}, \overline{A_1B_1}), dist(\overline{A_0B_0}, \overline{C_1C_1}), dist(\overline{C_0D_0}, \overline{A_1B_1}), dist(\overline{C_0C_0}, \overline{C_1D_1})\}$$

The $\overline{A_iB_i}$ determines the blade of the weapon, and $\overline{C_iD_i}$ the guard, as demonstrated in Fig. 5.6b. In practice, it is possible to omit the $dist(\overline{C_0C_0}, \overline{C_1D_1})$ from the minimum, since the guard to guard collisions are not important in fencing.

The surface of a capsule is smooth, thus there are no problems with the definition of a normal direction in any point p of the capsule's boundary – the normal is simply the unitized vector, that connects p with the nearest point on the line segment. A capsuloid can have discontinuity in a point, which has equal distance from two lines of the skeleton. However, the same definition of the normal can be applied also in this case (even though the normal in the mathematical sense does not exist).

The task of the dynamic CD is to answer whether two objects with given linear and angular velocities have intersected during the time interval $\langle t_0, t_1\rangle$ and if yes, then return the exact time $t_c \in \langle t_0, t_1\rangle$ of contact. The time t_0 is the time of displaying the previous frame, and t_1 is the current time. Put in a more formal way, we require the time interval $\langle t_0, t_c\rangle$ to be collision-free, but the objects collide in time $t_c + \epsilon_t$, where ϵ_t is a given precision. Obviously, the rest of the simulation in $(t_c, t_1\rangle$ must be discarded, because the weapons have penetrated.

We divide the dynamic collision detection algorithm into two parts:

1. test if the objects collided and if yes then return *any* time t_e during the first collision. (There could have been more collisions during Δt, but the others are of no concern to us.)

2. t_0 and t_e are given such that there is no collision in t_0 but there is a collision in t_e. Find the exact time of contact $t_c \in \langle t_0, t_e\rangle$.

The first point is complex in general. We present its approximate solution in section 5.2.3. The second point is much easier; it can be quickly solved with arbitrary precision as described in section 5.2.4.

5.2.2 Upper Bound of Velocity

The idea used in the next sections, is to avoid large inter-penetration. The inter-penetration magnitude is connected to the velocity of the colliding objects. Therefore we need an upper bound for the maximal velocity of the line segments, which define the capsuloid. In fact, it is straightforward to generalize the result for a triangle and thus also for a surface of a triangular mesh[4].

Lemma: Assume that triangle T rotates around a fixed axis with angular velocity ω and translates with linear velocity v. Let $0, 1, 2$ be vertices of the triangle T and r_c^0, r_c^1, r_c^2 their coordinates relative to some fixed point on the axis of rotation. If r_c^p is any point of the triangle with velocity v^p then

$$\|v^p\| \leq \|v\| + \max_{i=0,1,2} \|\omega \times r_c^i\|$$

Proof: It is well-known that the actual velocity of point r_c^p can be expressed as

$$v^p = v + \omega \times r_c^p$$

Taking the magnitude and using the triangle inequality yields

$$\|v^p\| = \|v + \omega \times r_c^p\| \leq \|v\| + \|\omega \times r_c^p\|$$

It remains to show that

$$\|\omega \times r_c^p\| \leq \max_{i=0,1,2} \|\omega \times r_c^i\|$$

Since r_c^p is a point of the triangle we can write it as a convex combination of vertices

$$r_c^p = r_c^0 s + r_c^1 t + r_c^2 u$$

where $s, t, u \geq 0$ and $s + t + u = 1$. Then by the cross product distributivity and triangle inequality

$$\begin{aligned} \|\omega \times r_c^p\| &= \|(\omega \times r_c^0)s + (\omega \times r_c^1)t + (\omega \times r_c^2)u\| \\ &\leq \|\omega \times r_c^0\|s + \|\omega \times r_c^1\|t + \|\omega \times r_c^2\|u \\ &\leq \max_{i=0,1,2} \|\omega \times r_c^i\|(s + t + u) \\ &= \max_{i=0,1,2} \|\omega \times r_c^i\| \end{aligned}$$

[4]Which is actually not necessary for the application itself, but may be useful in other contexts.

□

Corollary: When looking for an upper bound of a maximal velocity of any point in the triangle mesh it is sufficient to maximize velocities in the vertices. Especially for our case of capsule's skeleton, defined by two line segments, it is sufficient to maximize the velocity only in the endpoints of these segments.

We did not make any assumptions about the actual object orientation and therefore the upper bound is correct for arbitrary simulation time. Note that for minimal velocity of a point on a triangle this lemma does not hold – minimum may not be in a vertex.

5.2.3 Collision Search Algorithm

The input data are two capsuloids[5] A and B, defined by their skeletons. The linear and angular velocities $v_A, \omega_A, v_B, \omega_B$ of both objects are assumed to be constant during the simulation period $\langle t_0, t_1 \rangle$. Using the results of section 5.2.2 we determine the upper bound of maximal velocities v^A_{max}, v^B_{max} in any point of the skeleton of A, resp. B.

As mentioned earlier we confine ourselves to only an approximate solution. The idea of our algorithm is very simple: we perform the simulation during $\langle t_0, t_1 \rangle$ with small enough steps Δt and check for collisions statically. We stop as soon as we encounter a collision and return this time as t_e. It means that we really may miss some instant of collision, but we can ensure it will not be a big one – just a touch. Since no point on the skeleton of A (resp. B) moves faster than v^A_{max} (resp. v^B_{max}), the maximal possible inter-penetration will not be greater than

$$(v^A_{max} + v^B_{max})\Delta t$$

If we can admit the maximal inter-penetration depth ϵ then it is sufficient to take

$$\Delta t \leq \frac{\epsilon}{v^A_{max} + v^B_{max}} \tag{5.2}$$

Obviously if the denominator $v^A_{max} + v^B_{max} = 0$ then there is no need for dynamic collision detection, because it means the objects are not moving at all. The ϵ should be less than the thickness of the thinnest simulated object. Our particular choice of ϵ is discussed in section 5.2.5. Nevertheless, the smaller the ϵ, the slower the algorithm will be in the worst case, since the number of steps is

$$\frac{t_1 - t_0}{\epsilon}(v^A_{max} + v^B_{max})$$

[5]Nevertheless, the results still apply for general objects given by triangular surfaces.

This could be improved by introducing *dynamic* bounding volumes, which bound the whole swept volume, as described in [14]. However, for capsuloids, the speed-up is not necessary, since already this algorithm is quite fast (see section 5.2.5).

5.2.4 Time of Contact

As described in section 5.2.3 we have found the first time of collision t_e and we have the time t_0 without collisions as well. Then it is easy to determine the time of contact $t_c \in \langle t_0, t_e)$ by the binary search. The algorithm 4 finds t_c up to a given time precision $\epsilon_t > 0$.

Algorithm 4: Find time of contact
Input: Time t_0 without collision, t_e with collision
Output: The time of contact $t_c \in \langle t_0, t_e)$
FINDCONTACT(t_0, t_e, ϵ_t)
(1) $t_s = t_0$, $t_k = t_e$
(2) **while** $t_k - t_s \geq \epsilon_t$
(3) $t_m = \frac{t_s + t_k}{2}$
(4) **if** there is a collision in time t_m **then** $t_k = t_m$
(5) **else** $t_s = t_m$
(6) **return** t_s

The correctness of the algorithm is obvious from the invariant: t_s is always collision-free and in t_k is always present a collision. The time complexity is

$$\mathcal{O}\left(\log \frac{t_e - t_0}{\epsilon_t}\right)$$

Note that the algorithm always returns the time without a collision, when the system is in a correct state.

5.2.5 Implementation

The line segments are defined by the weapon's model file. By convention, the last mesh in the weapon model contains the auxiliary vertices, that define the weapon's shade and the capsuloid's skeleton.

There are actually three values of radii used by the application: $R_n = 0.5cm, R = 1cm, R_b = 1.25cm$. The inner capsuloid corresponds to the "rigid core", which really can not penetrate the other one. The $R = 1cm$ capsuloid is the approximating one, as depicted in Fig. 5.6b. The outer layer represents

a "safe distance" that will be used in the separation of capsules, section 6.4. The non-penetration condition of the inner capsuloids implies the value of ϵ, used in the collision search algorithm from section 5.2.3, namely $\epsilon = R - R_n$.

The dynamic collision detection routines are implemented together with collision response in class `ColResp`. The collision search algorithm, together with the time of contact computation, is implemented in the function `ColResp::findCollision`. It uses a subroutine `ColResp::maxStep`, which computes the step Δt according to the equation (5.2). A useful property is that the movement with constant linear and angular velocity (about a fixed axis) is equivalent to SLERP of the starting and final homogeneous matrices.

An average number of steps the collision search algorithm has to iterate is about 55. However each step is quite fast, since it consists of only two SLERPs and three segment to segment distance calculations. The average running time for the collision search algorithm is thus only 1.36ms. The time of contact computation according to algorithm 4 is executed only if the collision search algorithm has reported collision. It is even faster than the collision search, the average running time is below one millisecond.

Chapter 6

Collision Response

What makes the fencing interesting is the behavior of the weapon colliding with another weapon. Our task is to simulate the weapon-weapon collisions in a plausible way. For the purposes of simulation, the weapon is considered to be a rigid body, which is a good approximation for tough weapons, such as a sword. This enables exploiting the rigid body mechanics to simulate the object's dynamics, especially focusing on the reaction to collisions.

Concerning the rigid bodies collision response, a lot of work has been done, but no uniform approach is the best in general. See [2] for a survey. Physically based collision response together with dynamics simulation (via numeric solution of differential equations) is explained in excellent tutorial [4, 5]. Besides the colliding contact, it involves also a resting contact treatment, which is one of the most difficult problems. [4, 5] show a physically based solution of the resting contact using static forces. These forces are computed with quadratic programming. Another method for the contact forces computation is based on the linear-complementarity problem (LCP) [3, 8]. [28] proposes an alternative to static forces by so called micro-collisions – many small impulses that are used instead of static forces.

Well readable articles intended for game developers are [22, 12]. They cover mainly the basic physics and colliding contact. An interesting recent result is described by [26] – they formulate the dynamics simulation as an optimization problem and solve it by quadratic programming.

The existing algorithms are quite complex, because their goal is to enable a general simulation of rigid bodies. Thus, we have extracted only the stuff relevant for the simulation of two rigid bodies, which represent the weapons. We simplified certain approaches, namely the resting contact handling. Our method is described in [14]. Because no public simulation library was found, we have implemented the routines ab initio.

6.1 Rigid Body Mechanics

A *rigid body* is a solid object that can not be deformed in any way – its shape does not change during the simulation. If we denote the body's volume $V \subseteq R^3$ and density function $\rho : V \rightarrow R$ we can define the body's mass

$$m = \int_V \rho(r) dV$$

and the center of mass

$$\frac{\int_V r\rho(r) dV}{m}$$

Vectors relative to the center of mass are denoted by subscript $_c$. The moments of intertia are defined as follows

$$\begin{array}{ll} I_{xx} = \int_V (r_{c,y}^2 + r_{c,z}^2)\rho(r_c) dV & I_{yy} = \int_V (r_{c,x}^2 + r_{c,z}^2)\rho(r_c) dV \\ I_{zz} = \int_V (r_{c,x}^2 + r_{c,y}^2)\rho(r_c) dV & I_{xy} = \int_V r_{c,x} r_{c,y} \rho(r_c) dV \\ I_{xz} = \int_V r_{c,x} r_{c,z} \rho(r_c) dV & I_{yz} = \int_V r_{c,y} r_{c,z} \rho(r_c) dV \end{array} \quad (6.1)$$

where $r_c = (r_{c,x}, r_{c,y}, r_{c,z})$ is a body space vector relative to the center of mass. The moments form together an inertia tensor

$$I_{body} = \begin{pmatrix} I_{xx} & -I_{xy} & -I_{xz} \\ -I_{xy} & I_{yy} & -I_{yz} \\ -I_{xz} & -I_{yz} & I_{zz} \end{pmatrix}$$

All integrals are computed in the body space relative to the center of mass.

The placement of a rigid body is advantageously described relatively to its center of mass. In a world space coordinate system in time t we define the position of the center of mass $r_c(t)$ and orientation given by rotation matrix $R(t)$. Then the vector r_b in body space maps to the vector $r_w(t)$ in world space by formula

$$r_w(t) = r_c(t) + R(t) r_b \quad (6.2)$$

The actual inertia tensor $I(t)$ for a rotated body is computed from the body's space inertia tensor by equation

$$I(t) = R(t) I_{body} R(t)^T \quad (6.3)$$

as derived in [4].

The movement of a rigid body can be decomposed to a linear velocity $v_c(t)$ of the center of mass and an angular velocity $\omega(t)$ relative to the center of mass. $\omega(t)$ is the unitary axis of rotation multiplied by the angular velocity.

The velocity $v(t)$ of a point that has world space position $r(t)$ is computed as

$$\dot{r}(t) = v(t) = v_c(t) + \omega(t) \times (r(t) - r_c(t)) \tag{6.4}$$

The linear momentum $p(t)$ is computed from the linear velocity using the mass of the body:

$$p(t) = mv_c(t) \tag{6.5}$$

By analogy the angular momentum $b(t)$ is computed from the angular velocity using the inertia tensor:

$$b(t) = I(t)\omega(t) \tag{6.6}$$

To describe the state of a moving body it is recommended to use rather moments than velocities because they are conserved in nature unlike the velocities[1].

The rigid body's linear momentum can be changed by an application of force F acting in the center of mass. The change in linear momentum is described as

$$\frac{\partial p}{\partial t} = F$$

To rotate the object it is necessary to apply a torque τ. The torque is determined by the force F and the point of its application r in the world space relative to the center of mass:

$$\tau = (r - r_c) \times F$$

The change in angular momentum is similar to the linear case[2]

$$\frac{\partial b}{\partial t} = \tau$$

From the linear and angular moments it is straightforward to derive the velocities by multiplying equations (6.5), resp. (6.6) by inverse mass m^{-1}, resp. inverse inertia tensor I^{-1}. Note that the inertia tensor is a regular matrix if and only if the rigid body's mass is not zero.

[1] In a gyroscope $\omega(t)$ can change even if $b(t)$ is constant.

[2] More correctly we should say that τ means the *total external* torque as well as F is the *total external* force, because all the contributions of individual forces and torques reflect in moments.

6.1.1 Simulation of Rigid Body Collisions

In nature, the rigid bodies do never penetrate each other. When a collision occurs, the velocities of the colliding objects are changed in a discontinuous way, so that the bodies do not penetrate. The physical explanation for this phenomena is an *impulse*. The impulse of force J_F is

$$J_F = \int_{t_0}^{t_1} F dt$$

if $\langle t_0, t_1 \rangle$ is the period of collision. The impulse of force corresponds to the difference of linear moments

$$\Delta p = p(t_1) - p(t_0) = J_F$$

Consider that a rigid body A with linear momentum p_A collides with a rigid body B with linear momentum p_B. If the change of linear momentum Δp is added to p_A, then the opposite impulse $-\Delta p$ must be added to p_B to satisfy the law of conservation.

The impulsive torque of a force F applied in point r in world space is defined as

$$J_\tau = (r - r_c) \times J_F$$

Like the impulse of force changes the linear momentum, the impulsive torque changes the angular momentum

$$\Delta b = b(t_1) - b(t_0) = J_\tau$$

Since the angular momentum must be conserved too, the opposite impulsive torques have to be applied to both rigid bodies, as in the linear case.

We are especially interested only in the impulsive forces and torques that arise during the collision and prevent the rigid bodies from penetration.

6.2 Types of Contact

Assume that the algorithms presented in section 5.2 reported a collision between capsuloids A and B and computed the exact time of the first contact t_c. Now we shall examine the collision event and methods for its handling pursuant [5]. Let $r_A(t_c)$, resp. $r_B(t_c)$ be the point of contact of body A, resp. B in world space with respect to the center of mass of A, resp. B. The coordinates of these points are returned by the function that computes the segment to segment distance.

The points $r_A(t_c)$ and $r_B(t_c)$ coincide in the absolute coordinate system (not relative to the center of mass), but their velocities $\dot{r}_A(t_c)$, resp. $\dot{r}_B(t_c)$ can be quite different. Let us assume that the linear velocity of weapon A in time t_c is $v_A(t_c)$ and angular $\omega_A(t_c)$, analogically for weapon B. How to estimate these velocities is discussed in section 6.5.

Substituting to equation (6.4) we obtain following relations

$$\dot{r}_A(t_c) = v_A(t_c) + \omega_A(t_c) \times r_A(t_c) \tag{6.7}$$
$$\dot{r}_B(t_c) = v_B(t_c) + \omega_B(t_c) \times r_B(t_c) \tag{6.8}$$

for the actual velocities of the contact points.

The direction of the collision is described by the unit normal vector $n_B(t_c)$ of the surface of rigid body B[3] in point $r_B(t_c)$. Important fact is, that $n_B(t_c)$ points out of the object B's volume. As described in section 5.2.1, the normal to a capsuloid can be computed efficiently.

Now examine the relative velocity v_{rel} of two objects projected to the $n_B(t_c)$ direction

$$v_{rel} = n_B(t_c)(\dot{r}_A(t_c) - \dot{r}_B(t_c)) \tag{6.9}$$

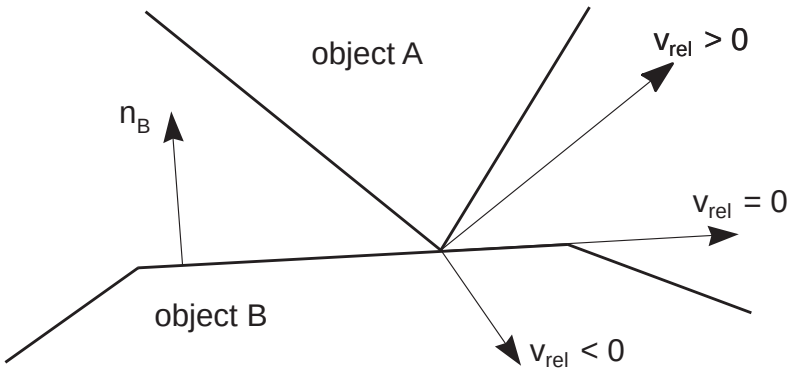


Figure 6.1: Possible relative velocity of two objects

If v_{rel} is positive (see Fig. 6.1), then the bodies are moving apart. If we have correctly computed the contact point then this option is theoretically impossible. If v_{rel} is negative, then the bodies are approaching, and if we do not change the object velocities immediately, then an inter-penetration occurs. This option is known as *colliding contact* and a method for computing new velocities is presented in section 6.3. If v_{rel} is zero, the bodies are neither

[3]We could choose the body A as well, it is just a convention introduced by [5].

receding, nor approaching. This situation is called a *resting contact*, and it is discussed in section 6.4.

Consider that we always work up to a certain numeric precision, thus we can not test if $v_{rel} = 0$, since it will practically never be true. This must be replaced with something like $|v_{rel}| \leq \epsilon_r$. It follows that what we classify as a resting contact, can be a small retreating or colliding contact in reality, which introduces certain difficulties. However the colliding contact event is quite clear since the condition is $v_{rel} < -\epsilon_r$.

6.3 Colliding Contact

The physical model for a colliding contact response is an impulse J, as explained in section 6.1.1. The impulse direction is given by $n_B(t_c)$ and what remains to compute is the impulse magnitude j (a scalar) so that

$$J = jn_B(t_c)$$

Once we have computed J, it is easy to compute the change of linear and angular moment (section 6.1.1). Recall that we use normal $n_B(t_c)$ outgoing from the rigid body B, pointing toward the rigid body A. Therefore the impulse acts positively on the rigid body A, and a negative impulse $-J$ must be applied to B due to the laws of conservation. From the new moments we obtain the new linear and angular velocities by inverting equations (6.5) and (6.6), as explained in the end of section 6.1.

In order to compute the impulse magnitude j we denote the pre-impulse quantities by superscript $^-$ and the post-impulse one's by $^+$. The empirical law for frictionless collisions states

$$v_{rel}^+ = -Cv_{rel}^- \tag{6.10}$$

where C is a *restitution coefficient* satisfying $C \in \langle 0, 1 \rangle$. It is connected with the elasticity of the collision: $C = 1$ means perfectly bouncy collision, where no kinetic energy is lost. On the other hand $C = 0$ means no bounce at all, the maximum of the kinetic energy is transformed.

The post-impulse velocities are connected with the pre-impulse one's by equations

$$v_A^+(t_c) = v_A^-(t_c) + \frac{jn_B(t_c)}{m_A} \tag{6.11}$$

$$\omega_A^+(t_c) = \omega_A^-(t_c) + I_A^{-1}(t_c)(r_A(t_c) \times jn_B(t_c)) \tag{6.12}$$

where m_A is the mass of the object A and I_A is its inertia tensor. Recall that I_A depends on the rotation of the object, eq. (6.3), and therefore on the time t_c. In the following we omit the time variable since it is always t_c.

Plugging equations (6.11) and (6.12) into (6.7) for post- and pre-impulse velocities we obtain

$$\begin{aligned}\dot{r}_A^+ &= v_A^+ + \omega_A^+ \times r_A = v_A^- + \frac{jn_B}{m_A} + (\omega_A^- + I_A^{-1}(r_A \times jn_B)) \times r_A \\ &= \dot{r}_A^- + j\left(\frac{n_B}{m_A} + (I_A^{-1}(r_A \times n_B)) \times r_A\right)\end{aligned}$$

The same can be derived for B considering that B is object of opposite impulse, i.e. of magnitude $-j$

$$\dot{r}_B^+ = \dot{r}_B^- - j\left(\frac{n_B}{m_B} + (I_B^{-1}(r_B \times n_B)) \times r_B\right)$$

Substituting these formulas into the v_{rel}^+ expression according to equation (6.9) and using the unit length of vector n_B, $n_B \cdot n_B = 1$ we have

$$\begin{aligned}v_{rel}^+ &= n_B(\dot{r}_A^+ - \dot{r}_B^+) = n_B(\dot{r}_A^- - \dot{r}_B^-) + \\ &\quad j(\frac{1}{m_A} + \frac{1}{m_B} + n_B(I_A^{-1}(r_A \times n_B)) \times r_A) + n_B(I_B^{-1}(r_B \times n_B)) \times r_B)) \\ &= v_{rel}^- + j(\frac{1}{m_A} + \frac{1}{m_B} + \\ &\quad n_B(I_A^{-1}(r_A \times n_B)) \times r_A) + n_B(I_B^{-1}(r_B \times n_B)) \times r_B))\end{aligned}$$

Applying the restitution law (6.10) yields

$$\begin{aligned}(-1 - C)v_{rel}^- &= j(\frac{1}{m_A} + \frac{1}{m_B} + n_B(I_A^{-1}(r_A \times n_B)) \times r_A) + \\ &\quad n_B(I_B^{-1}(r_B \times n_B)) \times r_B))\end{aligned}$$

from which we can already derive the impulse magnitude j since all the other variables are known. The inertia tensor inversion can be efficiently computed using equation (6.3) and realizing that

$$I^{-1} = (RI_{body}R^T)^{-1} = RI_{body}^{-1}R^T$$

The I_{body}^{-1} can be computed off-line.

The impulse J can be used not only for the post-collision velocities computation. If the input device supports force-feedback, we can project the impulse to the input device 2D space and set a force on the device proportional to the impulse magnitude.

6.4 Resting Contact

As mentioned above, resting contact is a situation where the relative velocity of two bodies is negligible, i.e. $|v_{rel}| \leq \epsilon_r$. Physically based treatment of resting contact is quite difficult, as mentioned in the beginning of this chapter. We simplify this task considerably by not accounting the friction and by exploiting the special properties of capsuloids. Originally in [14], only convex objects, such as one capsule, were considered. However, the capsuloid approximating the weapon (Fig. 5.6b) is not convex. The algorithms for convex objects can still be used (with certain adjustments), because our capsuloid is union of only two convex capsules.

Consider the convex objects first. Recall that in time t_c the objects are very close, but still not colliding. Because of the convexity assumption, we can use a separation theorem from computational geometry. It claims that if two convex sets are disjoint, then there exists a separation plane [24]. The problem is how to find the separation plane. To do this in a mathematically correct way, we would need the nearest points from both bodies and construct the separation plane as in the proof of the theorem (see for example [24]). But since the algorithm of this section is rather heuristic, it is sufficient to approximate the separating plane, exploiting the fact that the objects are very close to each other.

We have already computed the normal $n_B(t_c)$ of object B in point $r_B(t_c)$. Assume for a while that the point $r_A(t_c)$ is equal to $r_B(t_c)$ in absolute coordinate system. Then, since the (non-strict) separating plane exists, it must pass through the point $r_B(t_c)$. Because it must separate the bodies, the only choice in general is the tangential plane in point $r_B(t_c)$, i.e. with normal $n_B(t_c)$. This is a good approximation since the points $r_A(t_c)$ and $r_B(t_c)$ can be made arbitrarily close by the algorithm in section 5.2.4.

The idea of the resting contact solution for capsules (or general convex objects) is simple: we let the bodies move as if they were not influenced by each other – using the zero friction assumption. A problem may occur when the actual v_{rel} is small but negative. In this case the objects may even collide, which is tested dynamically as described in section 5.2. Small inter-penetration can be prevented by process we call *separation*: pushing the rigid bodies apart in the direction of the normal of the separation plane. We need to push the capsuloid skeletons, so that their distance is at least $2R_b$, using the R_b from section 5.2.5. If the original capsules distance is d, we accomplish this by translating the object A by vector $\frac{2R_b-d}{2}n_B(t_c)$, and the object B by $-\frac{2R_b-d}{2}n_B(t_c)$. Note that this really gets the capsules to distance $2R_b$ apart, as demonstrated in Fig. 6.2.

The separation for a non-convex capsuloid is somewhat more complicated,

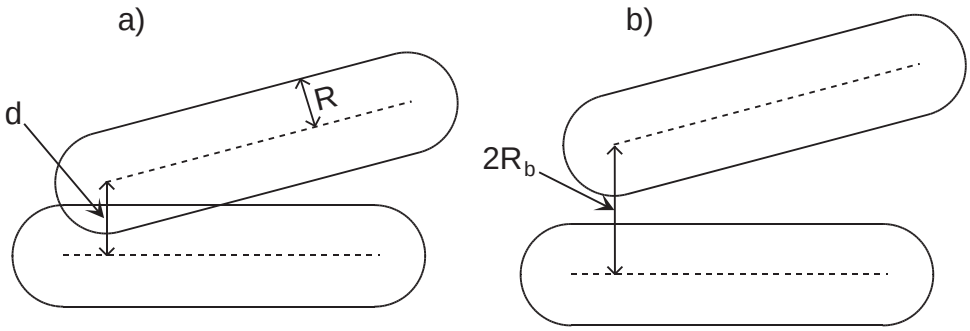


Figure 6.2: a) The resting contact situation, b) after separation to the distance $2R_b$

since it is not possible to exploit the separating plane theorem. The general non-convex objects can come into a position, from which the separation is non-trivial (e.g. pieces of a puzzle). However, it is not a failure, if the separation could not be executed – the objects simply stop moving and wait until the user breaks the clinch. In the fencing, this corresponds to weapons locked together[4].

The idea of pushing the objects apart can be still used even in the non-convex case, but the choice of the pushing direction is difficult. The exact computation of the smallest-distance pushing direction is presented in [16], but this approach would be swatting flies with a cannon. We settle for an approximation of the ideal separating direction. It depends on the number of features (blade-blade, blade-guard, guard-blade) that are in contact:

1 – the same direction as used in the convex case is a good attempt. However, it does not guarantee a successful separation because of non-convexity.

2 – an average of the two directions is applied.

3 – a rare case indeed, indicating a weapon lock. Nevertheless, we try the average of all the three directions.

Suppose we have determined the unit separating direction n'_B. Now it is necessary to compute the distance s, such that the translation of object A by $\frac{s}{2}n'_B$ and B by $-\frac{s}{2}n'_B$ makes the capsuloids distance $2R_b$ apart (according

[4]Which is really used in the historical fencing to perform some more sophisticated actions, such as disarming the opponent.

to the capsuloids distance definition from section 5.2.1). This is not as easy as in the convex case.

Let $D(s)$ denote the distance of capsuloid A shifted by sn'_B to capsuloid B. $D(s)$ is a continuous function, although not necessarily smooth. Note that $D(0) < 2R_b$, because otherwise no separation would be necessary. If we find an upper bound u such that $D(u) > 2R_b$, then we can apply a binary search algorithm, similar to algorithm 4. This is correct, because thanks to the continuity of $D(s)$, there exists $x \in (0, u)$ such that $D(x) = 2R_b$. The property that the left margin of the interval has $D(s) < 2R_b$ and the right margin has $D(s) > 2R_b$ is preserved by the invariant, so the binary search algorithm really exits with some[5] x' such that $|x' - x| < \epsilon_s,\ D(x) = 2R_b$. The $\epsilon_s > 0$ is the given precision.

It remains to be clarified how one finds the upper bound u such that $D(u) > 2R_b$. This can be done using a sphere that bounds the capsuliod's skeleton. Let $c_X, X \in \{A, B\}$, be the center of the longer line segment of capsuloid X, and r_X half of that line's length. The sphere with center c_X and radius r_X then bounds the skeleton of capsuloid X (the smaller line segment, corresponding to guard, also fits into this sphere). Now we push the sphere centers c_A, c_B to the distance $r_A + r_B + 2R_b$. Put in a more formal way, we compute u such that

$$\forall v \geq u :\ |c_A - c_B - vn'_B| > 2R_b + r_A + r_B$$

The inequality can be simplified to the quadratic function $u^2 + C_0 u + C_1 > 0$, which can be efficiently solved for the unknown u (taking the maximum of the roots).

6.5 Simulation Issues

This section could be also named "putting it all together". So far we have discussed how to handle single colliding or resting contact, but several questions are still open. Note that we have to simulate the whole time period $\langle t_0, t_1 \rangle$, i.e. the evolution of the system from the previous to the current frame. In order to implement the simulator, following problems must be addressed

- compute the inertia tensors of the weapon models
- estimate the velocity of the weapons
- consider more than one collision during the simulated time interval

[5] In some special situations there could be more x satisfying $D(x) = 2R_b$. The algorithm then returns one of them.

- animate the weapons after the colliding contact (deflection)

The moments of inertia are defined by integrals (6.1). We assume constant density in the whole object. The integrated functions are quite simple, but the integration domain is somewhat complex – it is the volume of the weapon. Direct numerical integration is very slow and leads to inaccurate results. Much more effective technique for moments computation has been described in [27]. It exploits the divergence theorem to cast the integrals to 2D, and then the Green's theorem for reduction to 1D. The 1D integrals are computed analytically, since the integrated functions are still simple.

Using the first moments, the weapon model is placed so that its center of mass aligns with the origin of the model's coordinate system. The weapon position and orientation is then always expressed relative to its center of mass[6].

The estimate of both linear and angular velocity is trivial, if the objects do not collide, because we know the positions and orientations in times t_0, t_1. However, if there is a collision in time $t_c \in (t_0, t_1)$, we can not use this formula directly, because the period $(t_c, t_1\rangle$ corresponds to an incorrect state of the system. Put in another way, the movement after t_c is not possible anyway, thus the estimate would be biased. Our solution is to store the linear and angular velocities from the previous frame, and interpolate them with the current ones, using $\frac{t_c - t_0}{t_1 - t_0}$ as the interpolation parameter. Only the magnitudes of the velocities are interpolated this way, there is no point to interpolate the direction (axis of rotation). Note that we do not actually simulate the linearly accelerated movement – the interpolation is used just to estimate the actual velocity of the objects in time t_c.

The core of the simulator is function `ColResp::GO`. It computes t_c using the algorithms described in section 5.2. After estimating the velocities, we determine the type of collision using the v_{rel} from section 6.2. In the case of colliding contact, the impulse, together with the post-impulse velocities, is computed by function `ColResp::applyImpulse`, according to equations from section 6.3. Now we try to simulate the movement of the weapons during $(t_c, t_1\rangle$ using the post-collision velocities. If the dynamic CD routines return no collision, we are done with simulating the time period $\langle t_0, t_1\rangle$. If the dynamic CD report a collision, the whole cycle is repeated – next collision is simulated. The number of collisions due to the colliding contact is bounded, because the restitution law implies an exponential decrease of the relative velocity. Thus, after a few iterations we eventually end up with resting

[6]For the deflection simulation, one could expect rotation around the point, where the weapon is held by the hand. However, the weapon's center of mass is a good approximation, because the handle is near to the center of mass, and the hand grip is not firm.

contact.

The resting contact (as well as the colliding contact with low relative velocity) is resolved using the concept of separation, as presented in section 6.4. There is the same problem as with the colliding contact – one separation may not be enough during $\langle t_c, t_1\rangle$. The solution is a succesive separation, which works according to algorithm 5.

Algorithm 5: Successive separation
Input: Simulated time period $\langle t_c, t_1\rangle$, object positions, orientations and velocities in t_c
Output: Object positions and orientations in t_1
sepPerPartes()
(1) $t = t_c$
(2) **repeat**
(3) compute separation direction n'_B and distance s
(4) separate the objects
(5) $t_s = t + f(s)$
(6) **if** $t_s \geq t_1$ **then return**
(7) simulate the motion during $\langle t_s, t_1\rangle$ with dynamic CD
(8) t = first collision during $\langle t_s, t_1\rangle$
(9) **until** no collision during $\langle t_s, t_1\rangle$

The function $f(s)$, used in the algorithm, can be interpreted as the time for the separation to distance s. It has been introduced in order to bound the number of iterations of algorithm 5. A simple linear function can do this job. The algorithm 5 is implemented in `ColResp::separatePerPartes`, which uses function `ColResp::separate` for the actual separation.

6.5.1 Deflecting Weapon

Suppose we have successfully simulated the movement of the weapon during $\langle t_0, t_1\rangle$. However, this is not sufficient: the impulse should influence the movement of the weapon for a longer period of time (the $t_1 - t_0$ is usually less than 0.1 second). When the weapon is hit hardly by the opponents weapon[7], the fencer looses control of his own weapon for a short amount of time, because of impulse delivered by the other weapon.

The precise way would be to simulate the dynamics of the human arm, but we did not investigate this possibility because of its complexity – already the kinematics of human arm (section 4.2) is not easy. We deal with this

[7] action known as *batuta* aiming to deflect the opponent's weapon

problem in two steps. First, we restrict the movement of the weapon itself, so that it does not fly away. Second, we simulate the process of re-gaining the control of the weapon.

The restriction of the weapon movement due to the collision is based on the angular velocity – it influences the deflection more than the linear velocity. The maximal deflection angle is computed using the post-collision angular velocity by function g, that was tuned empirically

$$g(|\omega|) = \left(1 - \frac{1}{\frac{|\omega|}{6} + 1}\right) \frac{\pi}{3}$$

Obviously, the deflection angle converges to the maximal value, that was set to $\pi/3$, as the angular velocity approaches infinity. From the angle is determined the time t, which the angular velocity ω must act to achieve angle $g(|\omega|)$. The position and orientation corresponding to the maximal deflection (homogeneous matrix `maxDef`) is then computed using time t. This is implemented in function `ColResp::compDeflection`. The position and orientation corresponding to no deflection (homogeneous matrix `noDef`) is also computed there, but it is trivial.

Another important variables are t_{sc} (`timeSinceCol`), the time since the collision event, and t_{un} (`unControlTime`), the total time when the weapon is not controlled only by the hand, i.e. when the fencer is catching the weapon. The resulting position and orientation R of the weapon in given time t_a is then computed as SLERP of `noDef` and `maxDef` with interpolation parameter $\frac{t_a - t_{sc}}{t_{un}}$.

In some situations, the SLERP from the `noDef` to `maxDef` placements can animate the weapon through the body of the avatar, that holds this weapon. This is not desirable, although it is not completely unnatural[8]. We can avoid some of these situations by considering the alternative path of the SLERP, as introduced in section 2.6. If the original path is found to be colliding with the humanoid body (which is for these purposes approximated by a cylinder), the other way is examined. If it also collides, then the original, shorter one, is chosen.

The influence of the actual hand (sword-holder) position and orientation must be also accounted, because it affects the direction of the caught weapon. Let us denote the actual sword holder homogeneous matrix by S. The resulting sword-holder position and orientation is then computed by SLERP of homogeneous matrices R and S, with interpolation parameter $\frac{t_a - t_{sc}}{t_{un}}$. This stuff is implemented in function `ColResp::applyDisplacements`.

[8]Even in real fencing the fencer can be accidentaly injured by his own weapon.

When we compute the final position and orientation of the sword-holder joint, there is yet one more problem concerning the inverse kinematics of the human arm (section 4.2). Recall that we used sampled elbow positions for the key postures of the sword-holder joint. Unfortunately, we can not use the sampled elbow positions when simulating the deflected weapon, because the weapon can move arbitrarily after collision (with all 6 DOFs). The solution is to use the comfortability optimization for the swivel angle determination, as described in section 4.2.2. The most comfortable swivel angle θ_{no} is computed for the `noDef` position, and θ_{max} for the `maxDef`. Elbow positions corresponding to $\theta_{no}, \theta_{max}$ are interpolated to produce plausible in-between swivel angles. The switch to the optimized swivel angle can be discontinuous, but this only emphasizes the collision event.

6.5.2 Performance and Conclusion

As presented in section 5.2, a dynamic CD test for two capsuloids is quite fast. However, during one simulation step (one call of `ColResp::GO`), more than one dynamic CD query may be necessary. The average running time of one iteration of function `ColResp::GO`, simulating both weapons, is $3.83ms$. However, there are big fluctuations: the maximal time was $40ms$ and minimal $0.39ms$. This is apparent – the collision free events are processed quickly, but some complicated collisions involving several colliding and resting contacts can be rather lengthy. Nevertheless, even the peak time is still admissible for real-time simulation.

The dynamics of the complex structure of the human arm holding weapon has been considerably simplified, in order to be able to solve it in an interactive application. The drawbacks can be observed seldom: the already mentioned penetration of the fencer's body with his own weapon, unnaturally loose hold of the weapon, etc. In spite of this, we believe that the applied algorithms are adequate for the virtual fencing application. Undoubtedly, a lot of work can be done in realistic dynamics simulation in future.

Chapter 7

Application Implementation

In the previous sections, we attempted to describe certain general topics with only a sketch of the actual implementation. Although these topics can be interesting by itself, the important fact is that they cooperate together. This chapter describes certain software aspects of the application's design, exploring some new features. The material of this chapter is of course original.

To document all the classes and functions of the application, another book would be necessary. Thus, we have selected only the most interesting parts. Some other features are described in the user's handbook, appendix A.

7.1 Networking

Since this is a computer graphics thesis, the networking support has not been the primary goal. However the support of more than one computer was enforced, because not every computer has two 2D input devices. Originally, we intended a peer to peer network architecture. Later we found that this is not suitable, because of the collision response. The collision response can not be computed by each participant individually, unless a sophisticated synchronization would be applied. The client-server model appears to be more appropriate. In this model, the server does all the animation, collision detection and response. The client is only a terminal with little intelligence, which can only display the current state of the simulation and send to server the user events.

More than two clients can be connected to the server, but only two can control the avatars. The others can be only in the viewing mode, acting as additional cameras (serving for example to the real judge). Prior to any networking design is the question of the communication protocol. The basic communication entity is a message. Several types of messages are supported

by the application, for example avatar's posture, triangle hit, user input event etc. The list of messages can be found in file `Communication.h`. We have concluded that the data should be transmitted in the most confined form, since the network layers are slow. The avatar's posture is encoded very economically in class `CModelState`. It contains the position and orientation of the sword holder, and parameters of the legs animation (type of step, time).

The server is initialized by function `CServerSocket::startServer`. The incoming client connection is signaled by call-back function `CServerSocket::OnAccept`, which registers the new client (if there is room for it) and sends him the initialization message. The message, for example that one informing the client about a new posture of the avatars, is send to all clients by `CServerSocket::sendAllModelState`. The low-level TCP/IP functionality is inherited from standard MFC class `CAsyncSocket`.

The actual endpoint of the communication pipe represents class `CCommSocket`, which is also derived from `CAsyncSocket`. It is used standalone on the client's side, and also as a member of `CServerSocket`. The received message is handled by call-back function `CCommSocket::OnReceive`, which buffers the incoming message, and when it is completed, ensures its processing. For example, the mouse event message (sent by a client) is dispatched to the appropriate `ActionMID` object on the server side. The client is initialized by `CCommSocket::startClient`, which also tries to connect to a given server. A message, for example the user input event message, is sent to the server by function `CCommSocket::sendInputEvent`.

The application has strong demands on the quality of the network connection. The very important factor is the network latency, which determines the delay of the network layer. Consider that if the user on the client machine moves the mouse, the message must be sent to server. The server computes the inverse kinematics, collision detection and eventually response and sends back the state of the models. Only after the client receives this message, it can display the result to the user. Thus only fast LAN networks are admissible. However, the use of extra input devices instead of the network is still recommended.

On the other hand, the network can be sometimes advantageous. Imagine we have machine A with strong CPU and two others (B,C) with slower CPU, but fast graphics. Then the server can run on machine A, performing the computations and letting the client machines B,C to draw the results.

7.2 Operation Modes

We have already touched the operation modes in section 7.1 a little bit: the client and server mode. In fact, the application provides more modes, enabling additional functionality:

- simulation mode – the default. It is the main part of the program: the actual simulation of fencing between two avatars. It can act also as a server, if the networking is set up. The simulation is saved to file `sequence.sav`, so that it is possible to replay it.
- replay mode – loads and replays a saved sequence. This can be used to analyze the fencing mistakes, as well as the program bugs.
- animtest mode – only one fencer is displayed and controlled. The user can test the avatar's behavior and skin deformation without interfering with another virtual humanoid.
- viewer mode – client. This mode is automatically entered after a client has established connection with the server. It displays the state of the models transmitted by server, and sends the input events to the server.

Each of these modes is implemented by a class inherited from the base class of all modes, `CMainMode`. Each mode must provide some basic functionality, for example displaying the scene by function `CMainMode::draw`. The main computations (e.g. for the simulation mode it is the inverse kinematics, collision detection and response) are done in function `CMainMode::play`, which is executed when the idle state has been signaled, i.e. there are no messages from the operating system in the queue. There is no fixed FPS rate.

The saving of the simulation progress, as mentioned above, is accomplished by class `AnimSaver`. Each time its function `AnimSaver::saveState` is called, it stores the current state of both virtual humanoid models – the joint transformations, hit triangles etc. For replay is used class `AnimLoader`, which first loads the sequence file into the memory. Then it can retrieve the model's state in arbitrary saved time by function `AnimLoader::loadRecord`. Please note, that there are saved some additional frames in the sequence, in order to simplify debugging. During the replay, these frames are distinguished by different color of the background.

The modes can be switched dynamically, by simply destructing the old mode object and creating a new one. This does the function `CChildView::initMode`.

7.3 Multiple Cameras

The basic framework of a MFC application contains two window classes: `CMainFrame` and `CChildView`. The `CChildView` actually occupies some part of the `CMainFrame` window. In our application, more windows are supported, in order to enable multiple independent cameras viewing the virtual environment. We implemented this by additional window objects of type `CCameraView`. During the main loop in the `CChildView` class, the array of active cameras is scanned, and each of them is instructed to redisplay.

The object of type `CCameraView` encapsulates the interface to the actual OpenGL displaying. First, it initializes the OpenGL rendering context by `CCameraView::InitRendering`. The main drawing is performed by `CCameraView::DrawGLScene`. This function sets up the viewing transformations, depending on the actual settings of the camera, and then calls the `draw` function of the current operation mode (a virtual function in `CMainMode`).

There is a lot of options in the viewing transformation settings. They are described in the user handbook, appendix A.

7.4 Virtual Humanoids

As already mentioned in section 3.1, the virtual humanoid's skeleton and skin is represented by class `Model`. However, we need a higher-level control of the virtual fencer – this presentes the class `FencerModel`. This class encapsulates the virtual humanoid model, together with its weapon model (also represented by class `Model`) and k-DOPs hierarchy for the purposes of collision detection. The class `FencerModel` offers only a little functionality by itself. Somewhat more interesting are their descendant classes `RemoteModel` and `LocalModel`.

The `LocalModel` class represents a fencer model, which is controlled by the local machine, possibly acting as a server. Besides the fencer model itself, this class also contains the inverse kinematics class `IKAN`, keyboard input class `KeyboardInputDevice`, and 2D input device `ActionMID`. The function `LocalModel::updateModel` does the virtual fencer's animation: it checks the current state of the input devices, calls the weapon deflection routine (which simulates the collision reaction, see section 6.5.1). It also calls the inverse kinematics for the hand, and the key frame interpolation for the legs. Finally, it propagates the new joint transformations to the whole skeleton by `Model::setFinalMatrices`.

The class `RemoteModel` represents model, which is controlled from the client, i.e. the actual animation and simulation is executed on a remote

machine. Recall that only a position and orientation of the sword-holder, together with the current time of the legs animation in transmitted. The `RemoteModel` has to reconstruct the avatars posture from this values, using inverse kinematics and key frame interpolation. This is implemented in function `RemoteModel::changedModelState`.

Appendix A

User Handbook

The virtual fencing application is, by its specification, an experimental program. It has no on-line help, and the user-friendliness was not one of the primary objectives. This appendix may help to overcome this problem.

The application needs not to be installed, but please check the hardware and software requirements from section 1.3. The file `gl1.exe` should not be executed from the CD, because the simulation is recorded in the actual directory. At startup, the program opens one camera window "Camera 0", and the main control window "gl1".

A.1 Main Control Window

The menu of the main control window is an entry point to the majority of program functions. The purpose of this window's fields is sketched in Fig. A.1. The unlabeled fields are intended for debugging purposes only.

Note that some controls can be used only in certain mode, for example the replay buttons and slider bar are useful only when the replay mode is active.

The camera window opened at startup can be closed, and the simulation is still performed – just the results are not displayed (it can be useful in the server mode). The main menu commands are following

- **Main**
  - **New Camera** – creates new camera window. Multiple independent cameras can be active at once.
  - **Start Judge** – starts an automatic judge. The automatic judge displays its messages in each camera window. Its first message is

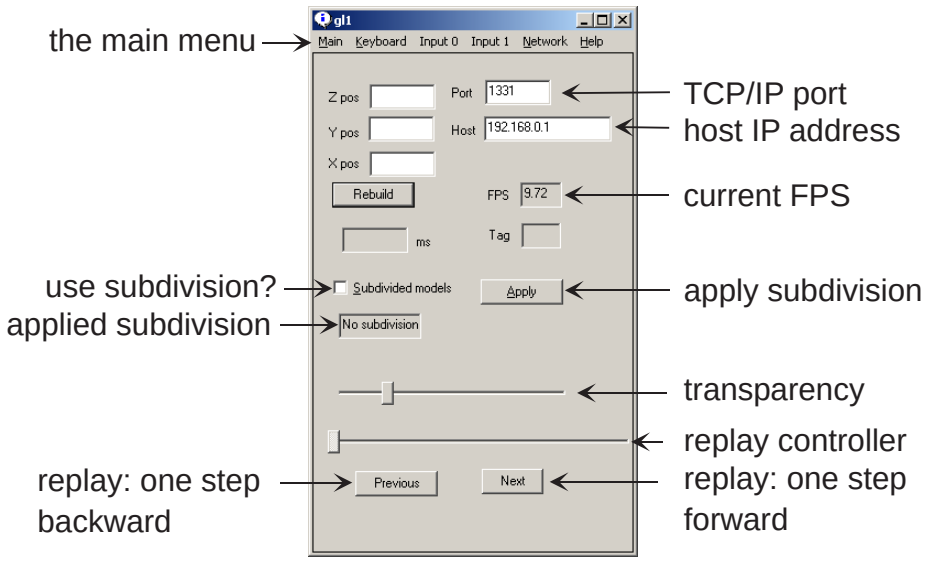


Figure A.1: The main control window

"En Guard", which means that the judge waits until both players go to guard position. Only after that the fencing can start.

- **Go** – switches to the simulation mode (the default).
- **Replay** – switches to the replay mode. An open file dialog appears, allowing to select the sequence file. The default one is `sequence.sav`.
- **AnimTest** – switches to the animtest mode.
- **Viewer (client)** – switches to the viewer mode. However, the user should not use this option directly, because without a connection there is nothing to display. This mode is automatically entered when a client connection is established.
- **Exit**

- **Keyboard** – opens the key selection dialog for the legs movements.
- **Input 0** – input devices for avatar 0. The selected input device for avatar 0 is checked. The contents of this pop-up changes dynamically. During initialization of the application, the list of all attached joysticks is appended to this menu. If a joystick does not appear in this menu, then there is some problem with the device or DirectX. The dialog of joystick attack and parry buttons assignment (corresponding to left

and right mouse buttons) is opened by another click on the already selected joystick.

Besides the joysticks, there are always two other possible input devices: **Nothing** and **Mouse**. The **Nothing** input device simply generates no input events – it is intended mainly for testing purposes. Note that both avatars can be controlled by the same input device (useful for testing in one person).

When the application is in the server mode and a new client connects, its IP address and port number are added to the list of input devices. If this is selected as the input device for avatar 0, it means that avatar 0 is controlled remotely by the appropriate client.

- **Input 1** – the same as input 0, but for avatar 1. The selected input device for avatar 1 is checked here.
- **Network**
  - **Listen (server)** – initializes the servers and starts listening on given port. Switches to the simulation mode.
  - **Connect (client)** – connects to the server on given IP address and port. If the connection is successfull, switches to the viewer mode.
  - **Close connection**
- **Help** – only a brief about box

A.2 Camera Window

As mentioned above, there can be active more camera windows in one instance of the application. Each of them can be configured independently using the camera window's menu.

- **Main**
  - **Fullscreen** – switches this camera to fullscreen mode.
  - **Close**
- **View** – defines where the camera is located
  - **Examine** – the view of both avatars. Useful mainly for replay.

  - **Avatar 0** – view from avatar's 0 head. Turns on avatar's 0 transparency.
  - **Avatar 1** – the same for avatar 1.
  - **Transparent avatar 0** – switches the transparency of avatar 0. The opacity is controlled using a slider in the main control window.
  - **Transparent avatar 1** – the same for avatar 1.

- **Look At** – determines how the camera is directed. Meaningful only if either avatar 0 or avatar 1 view is enabled, because it specifies the part of the opponent's body to which the camera is aimed.
  - **Root** – the pelvis
  - **Head**
  - **Hand**
  - **Elbow**
  - **Shoulder**

For every combination of **View** and **Look At** items, it is possible to fine-tune the camera's position and orientation (for example if you wish to aim the camera slightly above the root). When holding the *Shift* key and dragging the mouse, the camera translates. Holding *Ctrl* and simultaneously moving the mouse rotates the camera.

A.3 Virtual Fencer Control

It is not so easy to learn the virtual fencing, which is all right, because it is not easy to learn the real fencing as well.

First of all, it is important to enter the guard position. When the virtual fencer is not in guard, it can not do any step. If using judge, it automatically moves the fencers to the starting positions after hit. The guard then have to be entered again.

An ideal cut has three phases. The preparation, which enables the weapon to gain speed during the attack itself. Please note that it is necessary to swing the weapon up to the bounds of the 2D input device range – unless a cut will not be allowed, as discussed in section 4.3. After a succesfull preparation, the attack button (left button on the mouse) can be pressed, proceeding to the main phase of the cut. When holding the attack button, move the input device in the direction of the attack (just entering the cut position is not so effective). After performing the cut, release the attack button, or wait some

time, until the cut finishes automatically and the swing position is restored. It is also recommended to accompany the cut with a lunge, or at least with step forward. One can train a classic fencing principle, saying that the hand movement should precede the legs movement.

The parry positions can be entered anywhere, unlike the cuts, and it is also possible to stay in a parry for an unlimited amount of time. Because of high freedom of attack movements, it is difficult to parry, but it can be trained. The crucial thing is to move the weapon in direction *against* the opponent's attacking weapon when entering the parry position (holding the right mouse button). If the opponent is approaching, it is recommended to perform a step back prior to the actual defense (sometimes, it happens that the opponent does not reach – this is a perfect situation for counter attack).

Appendix B

Fencing Terminology

There is a lot of fencing styles in the world, as summarized in section 1.2. Each of them has slightly different terminology, although the basic principles are often the same. For the purposes of this work, we have selected one of the most popular terms, which are explained below.

- cut – an attack by the blade of the weapon (swept volume of the weapon is a plane)
- guard – basic posture of the body; an entry point for all legs movements
- hit by the point – an attack by the point of the weapon (swept volume of the weapon is a line)
- lunge – the legs routine used to quickly advance towards the opponent
- parry – defense action
- prime – type of a parry, see Fig. 1.1
- quarte – type of a parry, see Fig. 1.1
- quinte – type of a parry, see Fig. 1.1
- second – type of a parry, see Fig. 1.1
- sixte – type of a parry, see Fig. 1.1
- stand – a non-combat position of the body postured before the guard
- step forward – a step beginning and ending in guard
- step back – a step beginning and ending in guard
- tierce – type of a parry, see Fig. 1.1

Appendix C

Structure of the CD

- **DATA** – key frame positions and orientations of the weapon, discussed in section 4.2.1
- **EXE** – the application itself
  - **DATA** – data needed by the application: the weapon models, legs animations, textures etc.
  - **gl1.exe** – the main program
- **SOFTWARE**
  - **CFX_Setup.exe** – CharacterFX, a shareware virtual humanoid animation software
  - **MS3D_Setup.exe** – MilkShape3D, another shareware program for virtual humanoid design
- **SRC** – source code. Each source file describes itself in the header.
- **readme.txt** – brief info about contents of the CD
- **vf.pdf** – the text of the diploma thesis

Bibliography

[1] Paolo Baerlocher. *Inverse Kinematics Techniques for the Interactive Posture Control of Articulated Figures*. PhD thesis, EPFL, 2001.

[2] David Baraff. Non-penetrating rigid body simulation. *Eurographics 93 State of the Art Reports*, September 1993.

[3] David Baraff. Fast contact force computation for nonpenetrating rigid bodies. *SIGGRAPH*, July 1994.

[4] David Baraff. An introduction to physically based modeling: Rigid body simulation I - unconstrained rigid body dynamics. *SIGGRAPH Course Notes*, 1997.

[5] David Baraff. An introduction to physically based modeling: Rigid body simulation II - nonpenetration constraints. *SIGGRAPH Course Notes*, 1997.

[6] Jules Bloomenthal. Medial-based vertex deformation. *ACM SIGGRAPH Symposium on Computer Animation*, 2002.

[7] Gareth Bradshaw and Carol O'Sullivan. Sphere-tree construction using dynamic medial axis approximation. *ACM SIGGRAPH Symposium on Computer Animation*, 2002.

[8] Matthias Buck and Elmar Schömer. Interactive rigid body manipulation with obstacle contacts. *The Journal of Visualization and Computer Animation*, 9(4):243–257, 1998.

[9] David Eberly. *3D game engine design: a practical approach to real-time computer graphics*. Morgan Kaufmann Publishers Inc., 2001.

[10] Jens Eckstein and Elmar Schömer. Dynamic collision detection in virtual reality applications. In V. Skala, editor, *WSCG'99 Conference Proceedings*, 1999.

[11] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[12] Chris Hecker. Physics, part 3: Collision response. *Game Developer Magazine*, pages 11–18, March 1997.

[13] P. Jiménez, F. Thomas, and C. Torras. 3D Collision Detection: A Survey. *Computers and Graphics*, 25(2):269–285, April 2001.

[14] Ladislav Kavan. Rigid body collision response. In *Proceedings of the 7th Central European Seminar on Computer Graphics*, 2003. Also availaible in `http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2003`.

[15] Ladislav Kavan and Jiří Žára. Real-time skin deformation with bones blending. In *WSCG Short Papers Proceedings*, 2003. Also availaible as `http://wscg.zcu.cz/wscg2003/Papers_2003/G61.pdf`.

[16] Young J. Kim, Miguel A. Otaduy, Ming C. Lin, and Dinesh Manocha. Fast penetration depth computation for physically-based animation. *ACM SIGGRAPH Symposium on Computer Animation*, 2002.

[17] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, /1998.

[18] James Thomas Klosowski. *Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments*. PhD thesis, State University of New York, 1998.

[19] Jeff Lander. Making kine more flexible. *Game Developer Magazine*, November 1998.

[20] Jeff Lander. Oh my god, I inverted kine! *Game Developer Magazine*, September 1998.

[21] Jeff Lander. Skin them bones: Game programming for the web generation. *Game Developer Magazine*, pages 11–16, May 1998.

[22] Jeff Lander. Collision response: Bouncy, trouncy, fun. *Game Developer Magazine*, pages 15–19, March 1999.

[23] Jeff Lander. Over my dead, polygonal body. *Game Developer Magazine*, pages 17–22, October 1999.

[24] Jiří Matoušek. *Lectures on Discrete Geometry*. Springer, April 2002.

[25] Matrix and Quaternion FAQ. `http://skal.planet-d.net/demo/matrixfaq.htm`.

[26] Victor J. Milenkovic and Harald Schmidl. Optimization-based animation. *ACM SIGGRAPH*, pages 37–46, August 2001.

[27] Brian Mirtich. Fast and accurate computation of polyhedral mass properties. *Journal of Graphics Tools: JGT*, 1(2):31–50, 1996.

[28] Brian Mirtich and John F. Canny. Impulse-based simulation of rigid bodies. In *Symposium on Interactive 3D Graphics*, pages 181–188, 217, 1995.

[29] Tom Molet, Amaury Aubel, Tolga Capin, Stephane Carion, Elwin Lee, Nadia Magnenat Thalmann, Hansrudi Noser, Igor Pandzic, Gael Sannier, and Daniel Thalmann. Anyone for tennis? *Presence*, 8(2):140–156, 1999.

[30] Pavel Plch. *Historický šerm*. in Czech only.

[31] Brett Porter. PortaLib3D. `http://rsn.gamedev.net`, 2001.

[32] Vladimír Štěpán. Teaching tennis in virtual environment. In *SSCG'02 Conference Proceedings*, 2002. Also availaible as `http://cmp.felk.cvut.cz/~stepanv/files/sccg/StepanVladimirL.pdf`.

[33] Vladimír Štěpán. *Výuka tenisu ve virtuálním prostoru*. Diploma thesis (in Czech only), 2002.

[34] Deepak Tolani, Ambarish Goswami, and Norman I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models 62*, pages 353–388, 2000.

[35] Jiří Tůma. Linear algebra course notes (in Czech only). `http://adela.karlin.mff.cuni.cz/~tuma/st_linalg.htm`, 1997 – 2002.

[36] Gino van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools: JGT*, 2(4):1–14, 1997.

[37] Xiaohuan Corina Wang and Cary Phillips. Multi-weight enveloping: Least-squares approximation techniques for skin animation. *ACM SIGGRAPH Symposium on Computer Animation*, 2002.

[38] Jason Weber. Run-time skin deformation. Intel Architecture Labs, 2000.

[39] Chris Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master's thesis, Simon Fraser university, 1993.

[40] Wojciech Zablocki. Analysis of fencing movements. `http://www.kismeta.com/diGrasse/zablocki_SabreFencing.htm`.

[41] Gabriel Zachmann. Rapid collision detection by dynamically aligned DOP-trees. In *Proceedings of IEEE, VRAIS'98 Atlanta*, March 1998.

[42] Jianmin Zhao and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, pages 313–336, October 1994.