# Animating human model in OpenGL using data from motion capture system

Algirdas Beinaravicius
Aalborg University Copenhagen
Computer Vision and Graphics
algirdux@gmail.com

Gediminas Mazrimas
Aalborg University Copenhagen
Computer Vision and Graphics
g.mazrimas@gmail.com

## Abstract

*This paper explains how to animate a 3D human model in OpenGL using motion data. Human model here is a skeleton and its underlying layer - skin. Motion data processing for correct interpretation in animating program is vital. For the final visual result, skin deformations are considered the most important part of the animation, so our own implementation of the linear blend skinning technique is used. Such a technique allows to avoid some common animation problems, especially when dealing with more complex transformations. As the main concern is a full body animation, detailed body part deformations were not considered.*

***Keywords:*** *Human animation, Motion capturing data, Linear Blend Skinning*

## 1. Introduction

Our animation focuses on the most common and partly simple human body animation technique that uses joint based structure to animate human model. Joint structure, given their position and orientation, can be thought as human body skeleton. Animation of the skeleton, when talking about its complexity, is pretty simple, as it includes only rigid body parts and requires rotations of bones at the joint position. But when taking in account underlying layer of the skeleton - skin, it becomes complicated. This is when linear blend skinning technique comes in use, associating joints to vertices and introducing weights to them. Due to very fast computation speeds, this technique is the most popular in animation production. On the other hand, while using this simple shape blending technique to deal with complex transformations various skin deformation problems occur. Typical ones are collapsing elbow, candy-wrapper effect when the arm turns 180 degrees. Also such a technique doesn't consider many other human body deformations, like stretching or bulging muscles.

### 1.1. Overview

We present a motion data driven technique for animating human body model in a program written in a C++ programming language and working in an OpenGL environment. Main issues in this work are to get motion data, interpret it correctly for animation and then solve skin deformation problems that are caused by the body transformations. Realistic character movements in animation are achieved by using data that was captured using motion capture system. Of course motion data at first must be in appropriate format and then processed correctly, in order to use in our animation program. While dealing with skin deformation, our implementation of linear blend skinning algorithm is used combined with mathematical objects - quaternions in calculations.

In further sections the techniques used in this project are introduced (*2*, *3*, *4*), data and data formats that we have to deal with described in detail (*5*, *6*). Then our approaches in preparing 3D human model body mesh for our program (*7*) and in animating both skeleton and its underlying layer (*8*) are explained. Finally, problems that appeared during this project work are provided (*9*).

### 1.2. Previous works

Linear blend skinning is a very widely used technique, but due to its disability to deal with more complex skin deformations, it has many slightly different approaches and modifications for animating human body model. Furthermore, for animating human body model totally different methods are also used, such as anatomy-based ones [17], using different layers to imitate muscle structures, but we won't be going to details of such a technique.

Basic principles of linear blend skinning were described by the game development community [7, 8, 5] and the artifacts of this technique were discovered very soon [16]. The technique that dealt with some of those issues and gave more detailed overview for linear blend skinning type transformations was introduced in [9] and was named as "skeleton subspace deformation". It consider skin deformation as an interpolation problem and radial basis functions are used to interpolate between example skins.

Many other used techniques only extend the previous one. In [11] so called "extended linear blend skinning" is introduced. Approaching 180 degrees, the linearly blended matrix becomes degenerate and collapses the skin geometry. This method avoids blending transformations that are so dissimilar by adding extra transformations that properly interpolates without collapsing. In other paper [17] authors introduces non-linear technique to avoid unpleasant defects that are caused by linear shape blending. They achieve that by direct assignment of weights to the vertices according to its position around the joint. While in [5] authors suggest interpolating transformations itself instead of transformed vertex positions. Considering transformations consisting of a translation and rotation, they suggest using a quaternion representation.

To conclude, despite many techniques correcting the problems of linear blend skinning, none of them can fully avoid them. As a result, the traditional linear blend skinning is still widely used in many applications.

## 2. Linear blend skinning

In our project linear blend skinning technique was used to smoothly animate human body skin. This technique with or without modifications is widely used for various applications and goes by many different names, such as Skeleton Subspace Deformation or SSD or "smooth skinning" in Maya.

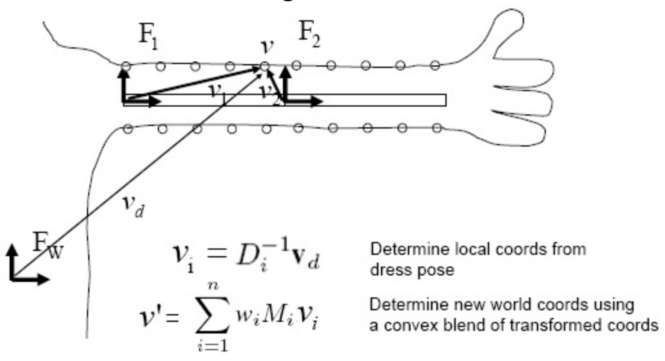
Basic principle of linear blend skinning algorithm is placing mesh model on the skeleton in some neutral pose, usually in T-pose or so called "dress pose". Then mesh vertices are assigned an influencing joints with predefined weights for these influences. In this section we look into the most common linear blend skinning method example. Our exact algorithm, which by the way is always influenced by only one joint, will be explained in further sections.

Formula for calculating new vertex position $\overline{\mathbf{v}}$ is

$$\overline{\mathbf{v}} = \sum_{i=1}^{n} w_i M_i D_i^{-1} \mathbf{v}_d$$

where $w_i$ are the influence weights, $v_d$ is the initial position of a particular vertex $\mathbf{v}$, $M_i$ is the transformation matrix associated with the $i$th influence, and $D_i^{-1}$ is the inverse of the initial pose matrix associated with the $i$th influence. Then $D_i^{-1}\mathbf{v}_d$ represents the position of $\mathbf{v}_d$ in the local coordinate system of the $i$th influence.

Figure 1.


$F_1$
$v$
$F_2$
$v_1$
$v_2$
$v_d$
$F_W$
$v_i = D_i^{-1}\mathbf{v}_d$
Determine local coords from dress pose
$v' = \sum_{i=1}^{n} w_i M_i v_i$
Determine new world coords using a convex blend of transformed coords

Linear blend skinning has its drawbacks also. Various artifacts, such as "candy-wrapper" effect and collapsing around bending joints, occur on complex deformations. Though different approaches of this technique may fix some of the most common problems.

## 3. Quaternions

In our project we introduce the mathematics of quaternions. The power of quaternions was being used to process the motion data as well as solving some problems, which occurred before the usage of this technique. Motion data, contained in BVH file (section 5.2), usually describes the transformation of each entity in 3D scene as three separate rotations around Z, Y and X axes. The usage of quaternions enabled us to find a single axis and a single angle equivalent to the rotations around Z, Y and X axes. This lead us to a much simpler implementation of the overall character animation as well as the solution for the "exploding knee" problem (Section 7). This section describes quaternions from a mathematical point of view, providing enough information to implement and use them.

A quaternion is a mathematical object, consisting of 4 scalars. Quaternions were discovered in 19th century by Irish mathematician Sir William Rowan Hamilton during his search for a way to represent points in space. Soon after their introduction, quaternions appeared at most of universities as an advanced mathematics subject. Nowadays quaternions find their applications in signal processing, physics, bioinformatics, orbital mechanics as well as computer graphics.

In computer graphics the usage of Euler angles is pretty natural and easily implemented. But Euler angles have their disadvantages as well. One of them is the well known "Gimbal lock" problem. The "Gimbal lock" occurs, when an attempt to rotate an object ends up with strange unsuspected results. The problem shows up due to the order in which the rotations are being performed, as in particular situations rotation around one axe might ”lock” the rotation around another axis. The usage of quaternions is the solution to the ”Gimbal lock” problem. Instead of rotating an object through a number of different axes, quaternions provide the ability to calculate a single axis and an angle realizing a number of separate independent rotations. Although the rotation is still performed by traditional matrix mathematics, instead of multiplying matrices together, quaternions, representing the axis of rotation, are multiplied together. The final resulting quaternion is then converted to the proper rotation matrix.

As mentioned earlier, a quaternion is four numbers, representing one real dimension and 3 imaginary dimensions. Each of its imaginary dimensions has a unit value of square root of -1 (a complex number). In addition to that, imaginary dimensions are all perpendicular to each other and can be noted as i, j, k. So a quaternion can be represented as follows:

$$\text{q} = a + \text{i}*\text{b} + \text{j}*\text{c} + \text{k}*\text{d}$$

where $a$ is a real dimension representation and b, c, d are just scalars.

Since quaternions are mathematical objects, they have a certain algebra with all the addition, subtraction, multiplication and division operations applied to them. In this section of the report we are going to consider just the quaternion multiplication operation, since this operation is necessary to represent a rotation. For all other operations refer to [2].

In order to perform a quaternion multiplication, a quaternions imaginary dimension multiplication has to be described first:

*i ∗ i = j ∗ j = k ∗ k = -1*

*i ∗ j = k*

*j ∗ i = -k*

*j ∗ k = i*

*k ∗ j = -i*

*k ∗ i = j*

*i ∗ k = -j*

The multiplication of two quaternions is :

(a + i∗b + j∗c + k∗d) ∗ (e + i∗f + j∗g + k∗h)
=
a ∗ e + i∗a∗f + j∗a∗g + k∗a∗h + i∗b∗e - b∗f + k∗b∗g - j∗ b∗h + j∗e∗c - k∗c∗f - c∗g + i∗c∗h + k∗e∗d + j∗d∗f - i∗d∗g - d∗h
=
(a∗e - b∗f - c∗g - d∗h) + i∗(a∗f + b∗e + c∗h - d∗g) + j∗(a∗g - b∗h + e∗c + d∗f) + k∗(a∗h + b∗g - c∗f + e∗d)

The important thing to notice is that the multiplication of two quaternions is not commutative, so that

$$\text{q1} * \text{q2} \neq \text{q2} * \text{q1}$$

As mentioned above, the multiplication operation, provides us with the ability to rotate one quaternion by another quaternion. So the rotation of quaternion q1 by quaternion q2 would result in another quaternion q = q2 ∗ q1;

Since the representation of a quaternion is quite complex and hard to imagine, it can be interpreted in another way. The x, y, z components of a quaternion can be treated as a representation of rotation axis and a component - as a representation of rotation angle. The relation between actual values and their representations as quaternion components can be described as:

- a = $cos(angle/2)$
- b = $axis_x * sin(angle/2)$
- c = $axis_y * sin(angle/2)$
- d = $axis_z * sin(angle/2)$

where $axis_x$, $axis_y$ and $axis_z$ is a normalized vector, representing the rotation axis.

In order to rotate a 3D point by a quaternion, a rotation matrix $R$ has to be produced:

$$R = \begin{pmatrix} a^2 - b^2 - c^2 - d^2 & 2*b*c - 2*a*d & 2*b*d + 2*a*c & 0 \\ 2*b*c + 2*a*d & a^2 - b^2 + c^2 - d^2 & 2*c*d - 2*a*b & 0 \\ 2*b*d - 2*a*c & 2*c*d - 2*a*b & a^2 - b^2c^2 + d^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Instead of calculating the direct rotation matrix, another approach is to calculate back the rotation vector and Eulerian angle from their quaternion representation. The angle can be calculated as:

$$angle = arccos(a) * 2 \tag{1}$$

The vector can be calculated as:

$\text{sin}_a = \sqrt(1 - a * a);$

$$vector_x = b/sin_a \tag{2}$$

$$vector_y = c/sin_a \tag{3}$$

$$vector_z = d/sin_a \tag{4}$$

## 3.1. Comparison between Eulerian angels and quaternions

During our project work, we faced many problems concerning the usage of Eulerian angles. For example the necessity of doing three separate rotations instead of just one. This lead to the problem of ”knee explosion” (section 9.2), which is closely related to the previously mentioned problem of ”gimbal lock”.

In the preceding section a detailed overview of quaternions was provided. Here we compare the mathematical calculations of quaternions with the Eulerian angles and reveal the drawbacks of the latter technique. Computer graphics mostly use the calculations of Eulerian angles as the representation of various transformations in 3D scene. This section only copes with rotations.

Any 3D affine transformation can be represented as a 4x4 matrix R:

$$R = \begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

A rotation in 3D scene is just a certain type of affine transformation that can be noted as a matrix R$_u(\delta)$:

$$\begin{pmatrix} c+(1-c)u_x^2 & (1-c)u_y u_x - s u_z & (1-c)u_z u_x + s u_y & 0 \\ (1-c)u_x u_y + s u_z & c+(1-c)u_y^2 & (1-c)u_z u_y + s u_x & 0 \\ (1-c)u_x u_y + s u_y & (1-c)u_y u_z + s u_x & c+(1-c)u_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where $\delta$ is the rotation angle, c = cos($\delta$), s = sin($\delta$) and (u$_x$, u$_y$, u$_z$) represent the rotation vector.

Worth thing to notice is the fact, that we can calculate back the rotation axis and the rotation angle from the rotation matrix. First of all, the calculation of rotation angle consists of summing up the four diagonal elements:

3c + (1-c)(u$_x^2$+u$_y^2$+u$_x^2$) + 1=

3c + 1 - c + 1 =

2c + 2 = 2 + 2cos($\delta$),

where (u$_x$, u$_y$, u$_z$) is the unit vector.

So, 2 + 2cos($\delta$) = a$_{11}$ + a$_{22}$ + a$_{33}$ + 1.

$$cos(\delta) = \frac{1}{2}(a_{11} + a_{22} + a_{33} - 1) \tag{5}$$

Similarly, we calculate back the rotation axis:

$$u_x = \frac{a_{32} - a_{23}}{2sin(\delta)} \tag{6}$$

$$u_y = \frac{a_{13} - a_{31}}{2sin(\delta)} \tag{7}$$

$$u_z = \frac{a_{21} - a_{12}}{2sin(\delta)} \tag{8}$$

In order to reveal the problems of Eulerian angles, we find it useful to define separate matrices of X, Y and Z rotations. The rotation around the X axis by the angle of $\alpha$ can be represented by the matrix R$_x(\alpha)$:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) & 0 \\ 0 & sin(\alpha) & cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

R$_y(\beta)$ matrix illustrates the rotation around the Y axis.

$$\begin{pmatrix} cos(\beta) & 0 & sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -sin(\beta) & 0 & cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The last rotation around Z axis is defined by the matrix R$_z(\gamma)$.

$$\begin{pmatrix} cos(\gamma) & -sin(\gamma) & 0 & 0 \\ sin(\gamma) & cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Since the animation part in our project contained of Z, Y, X rotations, this sequence is going to be used as the basis to illustrate the problems of Eulerian angles. In order to perform the Z, Y, X rotation sequence, the corresponding matrices have to be multiplied in the opposite order (matrix multiplication is not commutative). The overall rotation R = R$_x(\alpha)$ $*$ R$_y(\beta)$ $*$ R$_z(\gamma)$. The resulting matrix R is represented below.

$$\begin{pmatrix} cos(\beta)cos(\gamma) & -cos(\beta)sin(\gamma) & sin(\beta) & 0 \\ sin(\alpha)sin(\beta)cos(\gamma) + cos(\alpha)sin(\gamma) & -sin(\alpha)sin(\beta)sin(\gamma) + cos(\alpha)cos(\gamma) & -sin(\alpha)cos(\beta) & 0 \\ -sin(\beta)cos(\alpha)cos(\gamma) + sin(\alpha)sin(\gamma) & sin(\beta)cos(\alpha)sin(\gamma) + sin(\alpha)cos(\gamma) & cos(\alpha)cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{9}$$

Now, to illustrate the problem that appears using the Eulerian angles, let us consider the rotation around Z axis of 90 degrees, 90 degrees rotation around Y axis and the rotation around X axis of 90 degrees as well. Using the (9) matrix and substituting the corresponding values, we get the matrix, illustrating the latter rotations:

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

Using the (5) formula, we calculate back the total rotation angle:

$$\cos(\delta) = \tfrac{1}{2}(0-1+0-1) = -1$$
$$\Updownarrow$$
$$\delta = \arccos(-1) = 180^\circ$$

Now, if we wanted to calculate back the rotation axis using the (6), (7) and (8) formulas, we would see, that it is impossible, since sin(180°) = 0. That is, where the "gimbal lock" problem occurs.

To solve the "gimbal lock" problem, the calculations of quaternions were introduced. Quaternions q1, q2 and q3 represent the rotations around X, Y and Z axes respectively.

q1 = $\cos(\frac{\alpha}{2}) + i*\sin(\frac{\alpha}{2})$

q2 = $\cos(\frac{\beta}{2}) + j*\sin(\frac{\beta}{2})$

q3 = $\cos(\frac{\gamma}{2}) + k*\sin(\frac{\gamma}{2})$

The corresponding multiplication of these quaternions (section about quaternions) provides quaternion q, which represents the sequence of rotations around Z, Y and X axes:

$$q = q1 * q2 * q3 = \cos(\tfrac{\alpha}{2})\cos(\tfrac{\beta}{2})\cos(\tfrac{\gamma}{2}) - \sin(\tfrac{\alpha}{2})\sin(\tfrac{\beta}{2})\sin(\tfrac{\gamma}{2}) + i * (\sin(\tfrac{\alpha}{2})\cos(\tfrac{\beta}{2})\cos(\tfrac{\gamma}{2}) + \cos(\tfrac{\alpha}{2})\sin(\tfrac{\beta}{2})\sin(\tfrac{\gamma}{2})) + j * (-\sin(\tfrac{\alpha}{2})\cos(\tfrac{\beta}{2})\sin(\tfrac{\gamma}{2}) + \cos(\tfrac{\gamma}{2})\cos(\tfrac{\alpha}{2})\sin(\tfrac{\beta}{2})) + k * (\cos(\tfrac{\alpha}{2})\cos(\tfrac{\beta}{2})\sin(\tfrac{\gamma}{2}) + \cos(\tfrac{\gamma}{2})\sin(\tfrac{\gamma}{2})\sin(\tfrac{\beta}{2})) \quad (10)$$

Taking the values $\alpha = 90°$, $\beta = 90°$, $\gamma = 90°$ and substituting them in 10 formula, we get:

$q = 0 + i * \frac{\sqrt{2}}{2} + j * 0 + k * \frac{\sqrt{2}}{2}$

To calculate back the rotation angle and rotation vector, we use formulas (1), (2), (3) and (4).

$\delta = \arccos(0) * 2 = 180°$

$\sin_a = \sqrt{1 - 0 * 0} = 1$

$\text{axis}_x = \frac{\sqrt{2}}{2} / 1 = \frac{\sqrt{2}}{2}$

$\text{axis}_y = \frac{0}{1} = 0$

$\text{axis}_z = \frac{\sqrt{2}}{2} / 1 = \frac{\sqrt{2}}{2}$

So, as you can see the calculations of particular rotations using quaternions still represent correct results, whereas Eulerian angles fail.

## 4. Parametric representation of lines in 3D space

The approach of animating the character in our project, first of all, consisted of cutting the model in appropriate meshes (Figure 3 shows the picture of our character being cut in separate meshes). After the model had been cut, it was necessary to connect its separate parts so that the character is complete and full-scale. Apart from that, the connected area (what we further call the mesh connection) is the area, where we implement our approach to linear blend skinning. So in order to connect separate meshes, our project took advantage of parametric representation of lines in 3D space.

A line is a geometric object, which can be defined by two points, for example *A* and *B*. A line segment (or just segment) can be also defined by two points. The main difference between a line and a segment is that the latter one has a finite length. The segment only extends from point A to point B (usually points A and B are called segment endpoints). Although these objects, lines and segments, are very familiar, it is important to describe their parametric representation, as it was one of the solutions, which helped us to implement our approach to linear blend skinning technique.

In order for the linear blend skinning to work, the algorithm, implementing it, has to know exact coordinates of each vertex, which is going to be affected during the animation of the character. The parametric representation of a line/segment enabled us to find the required vertices, as well as their positions in 3D space.

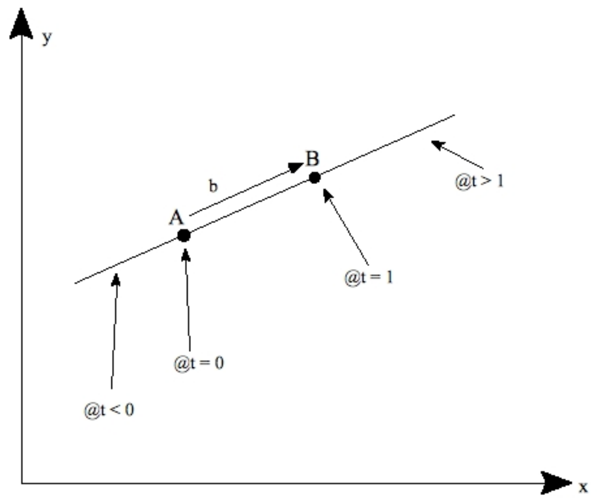
The parametric representation of a line *L* can be written as:

$$L(t) = A + b * t$$

where *b* is a vector: *b* = *B* - *A*, *A* is a starting point of a segment and *B* is an ending point of a segment.

This construction gives us a way to calculate any point along the line *L*. This is done using a parameter *t*, which can be imagined as a representation of time (the formula, realizing the parametric representation of a line, has a corresponding formula in the introductory physics of mechanics). In our case, parameter *t* can also be viewed as a number that sets apart one point on the line from another. As *t* changes its values, so does the point L(t) along the line. If *t = 0, L(0)* is the same position as *A*, so at *t = 0* we are at point *A*. At *t = 1, L(1) = A + (B - A) = B*. So as you can see, as *t* varies, we add a vector *b* (which is lengthened or shortened by the value of *t*) to the point *A*. As a result, we get a new point along the line. Taking a closer look, if *t* is larger than 1, this point lies somewhere on the opposite side of *B* from *A*. If *t* is less than 0, *L(t)* lies on the opposite side of *A* from *B*. Below is a picture illustrating these facts in a graphical 2D view, although 3D version uses the same ideas.

**Figure 2. Line in space**



A very useful fact to note are the values of parameter *t*, which lies between 0 and 1. These values proportionally scale the line/segment. For example, when *t = 0.3*, the point *L(t)* lies 30% of the way from point A. When *t = 0.5*, the point *L(t)* lies exactly in the middle of the line/segment. This fact can be proven by considering equation *L(t) = A + b * t*. Since | *B - A* | = | *b* |, then | *L(t) - A* | = | *b* | * | *t* |. So the value of | *t* | corresponds to the scale of the distance of | *B - A* |, since | *t* | = | *L(t) - A* | / | *B - A* |

## 5. Data formats

In this project we have to deal with various file formats. *C3D* file format stores motion data from motion capture system and *BVH* stores hierarchical skeleton structure and its rotations/translations adopted from C3D data. The *BVH* data is used to define the skeleton with its joint structure as well as all the animation frames, while the *OBJ* files are used for storing meshes of different human body parts, which are loaded on to the skeleton. Our project program required the direct parsing of *BVH* and *OBJ* files, so the corresponding parsers had to be implemented.

### 5.1. Description of Vicon C3D format

This section is meant to give just a brief introduction to the C3D file format, describing it in an abstract way. A reference to a detailed manual, describing the format, is given in the reference part of this document.

The C3D (Coordinate 3D) file format is a binary data file format originally developed for the AMASS photogrammetry software system, capable of storing 3D data as well as analog data together with all associated parameters for a single measurement trial. Since only the 3D data is of importance, we will not consider analog data in this document. The main advantage of C3D format over other motion capture data formats is that it is able to encapsulate the motion data as well as parameters, describing the motion data, in a single file. Apart from that, C3D is freely licensed and well documented.

Being a binary file, the C3D file consists of a number of 512-byte blocks. Logically C3D format can be divided into 3 basic sections, each having one or more 512-byte blocks:

- Header section is the first section of a file. The main purpose of this section is holding a pointer to the start of parameters section. Other parts of this section, is of no particular importance, as usually it consists data, copied from parameters section.
- The parameters section usually starts at block number 2, although this is not fixed and should not be assumed to be the case for every C3D file. This section contains information about the 3D data stored in the file. The section is extensible, meaning that user can define its own parameters without violating the format specification.
- Data section containing the 3D point coordinates is usually located after the parameters section. This section simply contains sequential frames data. In the case of 3D points (the data is X, Y and Z coordinates).

### 5.2. Description of Biovision BVH format

The BVH format is an updated version of BioVisions BVA data format, with the addition of a hierarchical data structure representing the bones of the skeleton. The BVH file is an ASCII file and consists of two parts:

- *Hierarchy* is for storing joint hierarchy and initial pose of the skeleton, basically joint-to-joint connections and offsets.
- *Motion* describes the channel motion data for each frame, that is the movement of individual joints.

The *hierarchy* section starts with *root* joint and contains the definition of a joint hierarchy within nested braces like source code written in the C programming language. Each joint in a *hierarchy* has an *offset* field and a *channels* field. The *offset* field stores initial *offset* values for each joint with respect to its parent joint. *Channels* field defines which channels of transformation (translation and/or rotation) exist for the joint in the *motion* data section of the file. The *channels* field also defines the order of transformation. A *channel* is either x-, y-, or z-translation or local x-, y-, or z-rotation. All the segments are assumed to be rigid and scaling is not available. The *end site* field is also available in order to determine body segment end.

In the *motion* section the total number of frames in the animation and the frame speed in frames-per-second is defined. Every next row then contains data values for all *channels* which were specified in the *hierarchy* section. The listing order of *motion* values in each row in is assumed to match their listed order from the *hierarchy* section (top down).

There are few drawbacks of the BVH format. One is that it lacks a full definition of the initial pose. Another drawback is that the format has only translational offsets of children segments from their parents. No rotational offset can be defined. Moreover, the BVH format is often implemented differently in different applications, that is, one BVH format that works well in one application may not be interpreted in another. All the same the format is very flexible and it is relatively easy to edit BVH files.

### 5.3. Description of OBJ format

OBJ is a geometry definition file format, first developed by Wavefront technologies. The file format is open and has been adopted by most of 3D graphics application vendors. It can be imported/exported from most of 3D modeling tools such as Autodesk's Maya, 3ds Max, Newtek's Lightwave, Blender, etc. This section of the document provides a thorough explanation of the most important parts of an object file, with more details provided in the reference page.

An object file can be stored in ASCII (using the ".obj" file extension) or in binary format (using the .MOD extension). The binary format is proprietary and undocumented, so only the ASCII format is described here.

The OBJ file format supports lines, polygons, and free-form curves and surfaces. Lines and polygons are described in terms of their points, while curves and surfaces are defined with control points and other information depending on the type of curve. Since, in our project, lines and polygons were sufficient to represent the model appropriately, only the parameters, concerning, these entities are described below.

The OBJ file is composed of lines of text, each of them starting with a token, which describes the type of the entity being recorded in that line. Below are listed various tokens, which were of importance in our project:

- "#" - a comment line. Lines, starting with "#" token, are simply skipped by OBJ file readers.

- "g" - a group line. Lines, starting with "g" (group) token, determine the start of a group. "g" token is followed by the group name.

- "v" - a vertex line. Lines, starting with "v" (vertex) token, provide the information, concerning vertices. This token is followed by x, y and z coordinates of the vertex.

- "vt" - a texture line. Lines, starting with "vt" (vertex texture) token, are recorded with information, concerning textures. "vt" token is followed by x, y and z coordinates of the texture, although only the x and y coordinates are of importance (z coordinate is 0.0).

- "vn" - a vertex normal line. Lines, starting with "vn" (vertex normal) token, contain information, concerning the normal of a vertex. "vn" token is followed by x, y and z coordinates of the normal.

- "f" - a line, describing face. Lines, starting with "f" (face) token, provide the information, concerning polygons. "f" token is followed by a number of triplets, which is equal to the number of vertices, the polygon has. Each triplet is of a form "int/int/int", where the first "int" is a vertex position in the file, the second "int" is "texture" position in the file and the third "int" is a normal position in the file.

The above described tokens were of importance in our project, but they are not the only ones that OBJ format supports. The OBJ format specification is much broader and covers such abilities as surface encoding, connectivity between free-form surfaces, rendering attributes. For a full OBJ specification refer to [1].

## 6. Motion capture

Motion capture (*mocap*) is sampling and recording motion of humans, animals and other various objects as 3D data. The data can be used to study motion or to animate 3D computer models. During the motion capture process not only the capturing stage using *mocap* equipment is very important, as equally important are the preparation and post processing processes. The whole system must be well calibrated, adjusted and set up, furthermore, after capturing data needs to be cleaned, edited, and applied to a 3D model.

During this project, for capturing motion data, we used Vicon Motion System, combined with Vicon IQ 2.5 software running on our systems host pc.

### 6.1. Motion data

There are various *motion data* storing formats, that mostly depend on what kind of program they are being used on. Our aim on this project, considering motion data, was to select the most appropriate data format. After reviewing some examples, the decision was made, that the best choice for our model animation would be using *BVH* motion data format. In *BVH* file whole model joint structure is described and motion data is represented as rotations and translations of these joints. This feature allows us to easily animate our skeleton, without any additional calculations. Only drawback is that our motion capturing is being done using Vicon IQ software, which is not able to directly export data to BVH file format. At first we need get Vicons' C3D data file and then convert it to BVH using other 3rd party application.

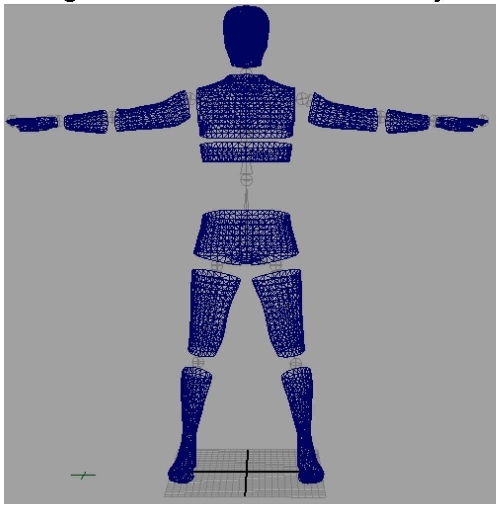
## 7. Human body mesh model

Human body mesh model must be selected not only by its looks and model details, but also by its position. The model must be in such a position, that in animating stage you could connect models body parts with your skeleton and make it move. In this project simple human body mesh was selected with initial T-pose.

### 7.1. Mesh model preparations

Mesh model is a vital object in animation, looking from the users (viewers) side. After using all the mathematical

**Figure 3. Mesh model cut in Maya**



theories and different approaches, the final result highly depends on the mesh being used. For our project we decided to use whole human body mesh cut in different body parts, with empty space between these rigid parts, that are later connected to one mesh. The main issue using this kind of approach, is that you lose details of your body mesh, so the better solution is using the whole uncut body.

## 8. Animating human body

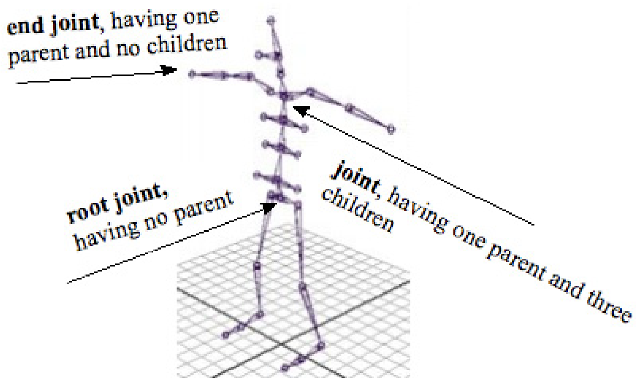
The base of any character animation is its skeleton. Skeletons are composed of a hierarchical structure of joints and bones that let you pose and animate your model. As our skeleton is composed of joints (bones are just segments between corresponding joints), most of the time only the joints will be considered as the base of characters' movements.

So, each joint has a number of children joints and one parent. Only the root joint does not have a parent and end-joints don't have children. From a programmers point of view joint hierarchy can be treated as a tree data structure, having a root, nodes and leaves.

To animate a character, first of all, the animation data has to be applied to character skeleton (every joint that the skeleton consists of). Generally, the motion of an individual joint consists of translation, rotation and scale components (scaling is usually applied to character bones). All these components can be merged together to give an overall transform using homogenous coordinates. If the translation, rotation and scaling is being applied, the overall transformation can be defined as multiplication of corresponding matrices:

$$M = T * R * S$$

**Figure 4. Human body skeleton**


**end joint**, having one parent and no children
**root joint**, having no parent
**Joint**, having one parent and three children

where $M$ is an overall transformation matrix, $T$ is translation matrix, $R$ is rotation matrix and $S$ is scale matrix.

In our case, only the root joint had the translational data, applied to it, whereas all other joints were being applied with rotational data. No scaling data was introduced in our motion capturing data.

For the animation to look correctly, a forward kinematics technique has to be applied. With forward kinematics, you rotate or translate individual joints to animate your skeleton. Transforming a joint in 3D scene affects that joint and any joints below it in the tree hierarchy. In other words, if you rotate any joint, you have to rotate all of its children as well. For example, the shoulder joint rotation by an angle $O$ would affect the elbow joint and wrist joint. The elbow should be rotated by $O$ degrees, as well as the wrist should be rotated by $O$ degrees.

From a more detailed point of view, each joint has a local transformation that describes its orientation within its local coordinate system, which is dependent on its parents' local transformation. To calculate a global matrix transform for a given joint, the local transform needs to be pre-multiplied by its parents' global transform, which itself is derived by multiplying its local transform with its parents' global transform and so on, until you reach the root joint. For the root joint, the local and the global transforms are the same. The equation below outlines this combination sequence, where $n$ is the current joint, whose parent joint is $n$ - $1$, $n = 0$ is the root joint and $M$ is the joint transformation matrix.

$$M^n_{global} = \prod_{i=0}^{n} M^i_{local}$$

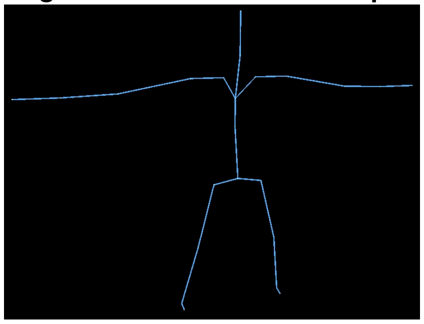
So, for example to get the global transformation of the left ankle joint, the overall transformation sequence might look like this:

$$M_{global\_left\_ankle} = M_{local\_root} * M_{local\_left\_hip} * M_{local\_left\_knee} * M_{local\_left\_ankle}$$

## 8.1. Preparing the character for animation

Our model consists of 18 joints, defined in the BVH data file: Hips (root joint), LeftHip, LeftKnee, LeftAnkle, RightHip, RightKnee, RightAnkle, Chest, Chest2, LeftCollar, LeftShoulder, LeftElbow, LeftWrist, RightCollar, RightShoulder, RightElbow, RightWrist and Neck. All of these joints should be positioned in such a way, that the skeleton is drawn in a T-pose. Below is a picture 5 of our skeleton.

**Figure 5. Our skeleton example**

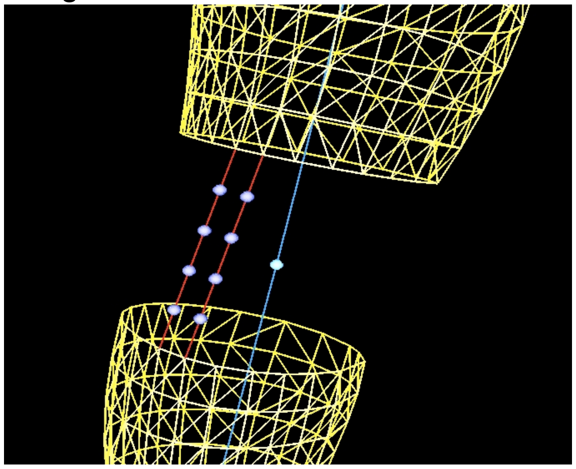


The joints, mentioned above, allow our model to be correctly animated in 3D scene, although much detail is not achieved as the movement of fingers and toes is not animated. Each joint has a mesh applied to it, which is not affected by the animation data, except being rotated or translated in the scene. The connection between the meshes is being done automatically during the construction of the character. The algorithm for doing that is:

1. Find two vertices, belonging to the mesh and its parent mesh, which have the shortest distance of all the vertices.
2. Determine some threshold value, which can be just user defined (for example 5%). The threshold value determines the longest distance between vertices, belonging to the mesh and its parent.
3. Find all other vertices, which are necessary to connect, by calculating the distance between each of them. If the distance is between the shortest distance and the distance, calculated from the threshold value, then those two vertices have to be connected.

Each connection between the mesh and its parent has subvertices. This is necessary for every joint rotation to look smooth. The more subvertices we have, the smoother the rotation looks. To find positions of subvertices on the connecting line, we apply the parametric representation of a line. To give a better view, of how subvertices are introduced in our model, the image of models right knee is provided. In the image 6 you can see two meshes being connected by two connecting lines, each of which has four subvertices. Of course, this picture is provided just for explanatory purposes, the full connection between the meshes should have much more connecting lines.

**Figure 6. Connections between meshes**



Any line function can be written as: $L(t) = A + b*t$, where $A$ is the starting point of a line, $b$ is a vector, determining the direction of a line and $t$ is a parameter value. A more detailed explanation of the parametric representation of the line can be found in documents section 4.

Since the above mentioned algorithm finds the positions of the first vertex, necessary to connect, (which is letter $A$ in the parametric representation of the line) and the second one (lets mark its position as letter $B$), we can find the direction vector $b = B - A$. Now taking t from 0 to 1 we get the position of the subvertex, which is between $A$ and $B$. In fact if we took $t = 0$, from the parametric representation of the line we get the position of vertex $A$, and if we took $t = 1$, we get the position of vertex $B$. So to determine the position of the first subvertex on the line, we need to set $t = 1/N$, where $N$ is the total number of subvertices. To determine the position of the second subvertex on the line, set $t = 2 * 1/N$. In general, to find the $n$'th subvertex position on the line, we can set

$$t = n * 1/N$$

## 8.2. Animating the character

Our character is being animated by the data, provided by the Vicon motion capturing system. Motion data is encoded as BVH format file, consisting of 1289 frames (a more detailed explanation about Biovision BVH format is provided in section 5.2). Each frame holds rotation values for every joint as well as translation values for root joint in 3D space.

Since rotation values are provided as three separate rotations around *X*, *Y* and *Z* axis, quaternions are being used to calculate rotation axis and rotation angle.

For example in the first frame, the root joint is being rotated by -0.66 degrees around *Z* axis, 0.80 degrees around *Y* axis and 171.21 degrees around *X* axis. By applying the mathematics of quaternions, the overall rotation vector is (0.00631242, 0.999959, 0.00653654) and the rotation angle is 171.209 (quaternions are being explained in section 3). Since BVH format file provides only local rotations for each joint, to get the global rotation in 3D space, each joints' parent rotation should be taken into account. Considering the previous example, not only the root joint has to be rotated by 171.209 degrees around the rotation axis (0.00631242, 0.999959, 0.00653654), but all of its children as well.

Since each joint has a mesh applied to it, during the animation, meshes are transformed in 3D scene the same way the joints are translated and rotated. Only the mesh connection has a different approach. In order for the mesh connection to rotate correctly, our approach of linear blend skinning technique is applied, which is explained in more detail in section 8.3.

### 8.3. Our approach to linear blend skinning

To calculate the position of subvertices after rotation, we apply our approach to the linear blend skinning technique, which means that each subvertex has some weight (lets mark it as *w*), which influences the angle it has to be rotated. Of course, each subvertex weight could be determined manually (for example, user defined), but we found that it is pretty easy to determine it automatically to get the satisfying overall result.

Lets consider that we have *N* vertices on the connecting line. The joint is being rotated by angle *A*. Each vertex weight *w* is being determined by its position on the connecting line:

$$w = \mathrm{P} * 1/\mathrm{N}$$

where P is the position on the connecting line.

So the first vertex on the line has the weight $w = 1 * 1/N$, while the last vertex on the line has the weight $w = N * 1/N = 1$.

Now, considering the angle each vertex has to be rotated, the first one is rotated by $A * 1 * 1/N$ degrees, while the last one is being rotated by $A * N * 1/N = A$ degrees. After rotating all the subvertices, they are being connected to polygons.

### 8.4. Relation between our approach to linear blend skinning technique and the original implementation

From the section 2, the traditional implementation of linear blend skinning is defined by the formula:

$$\bar{\mathbf{v}} = \sum_{i=1}^{n} w_i M_i D_i^{-1} \mathbf{v}_d$$

where $w_i$ are the influence weights, $v_d$ is the initial pose location of a particular vertex **v**, $M_i$ is the transformation matrix associated with the *i*th influence, and $D_i^{-1}$ is the inverse of the initial pose matrix associated with the *i*th influence.

Since our implementation of linear blend skinning technique has only one influence, the necessity of summing up the influences disappears. The weight $w$ = P $*$ 1/N as described in previous section 8.3. The transformation matrix *M* is calculated by applying the quaternions technique (section 3). The $D_i^{-1}\mathbf{v}_d$ part of the original formula is the same in our linear blend skinning approach, representing the position of $\mathbf{v}_d$ in the local coordinate frame. Below is the overall formula, illustrating our implementation of linear blend skinning technique.

$$\bar{\mathbf{v}} = P/NMD^{-1}\mathbf{v}_d$$

## 9. Problems

In this section we give some examples of the problems that occurred during the project. Some of these problems were solved during the work, while others still need to be seen into.

### 9.1. Initial BVH pose

During the animation of human skeleton, the initial pose is not important, but when the mesh is considered, it is all they way around. As our human model was cut into individual parts, we had to have appropriate initial pose to correctly connect separate meshes. In our case, the appropriate pose was the T-pose. The T-pose was the only pose, which enabled us to find the shortest distance between two meshes and connect them correctly. The detailed way how the meshes are connected is described in section 8.1.

The problem was, that after processing motion data and exporting it to BVH file, the character was not in initial T-pose. The same problem occurred after trying a number of examples downloaded from the internet. The initial pose was the I-pose instead of the T-pose. Due to BVH structure, fixing this problem requires the file to be manually edited:

1. Skeleton structure and joint offsets in BVH file has to be changed to match T pose.

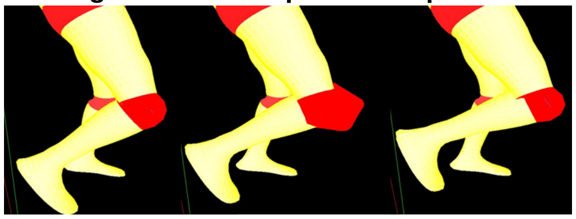
2. All the frame rotations has to be recalculated, as the rotations in BVH file are given from the initial position.

We managed to automate these two steps by using third-party application. After changing the initial pose, the application automatically recalculated frame rotations.

## 9.2. "Exploding knee" problem

The "exploding knee" problem occurred during the animation, when the overall rotation is being applied as three separate rotations around Z, Y and X axes. This artifact can be seen in figure 7. During the rotations around Z, Y and X axes in a row, the linear blend skinning algorithm, which we apply to animate connections between meshes, provides faulty results. This kind of situation is called "gimbal lock", which is described in detail in section 3.1. The solution for this problem was the introduction of quaternions, which enabled us to calculate a unique rotation vector, as well as rotation angle, representing three separate rotations around Z, Y and X axes.

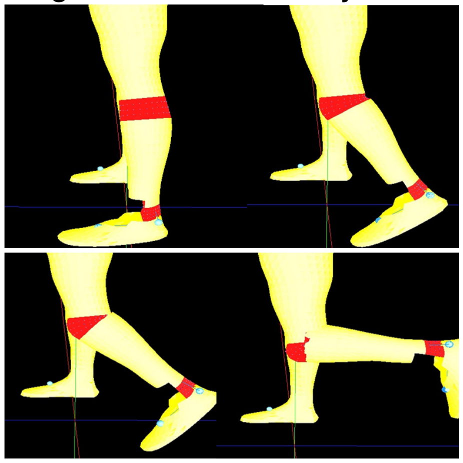
**Figure 7. Knee explosion sequence**



## 9.3. Mesh connections collapsing on complex deformations

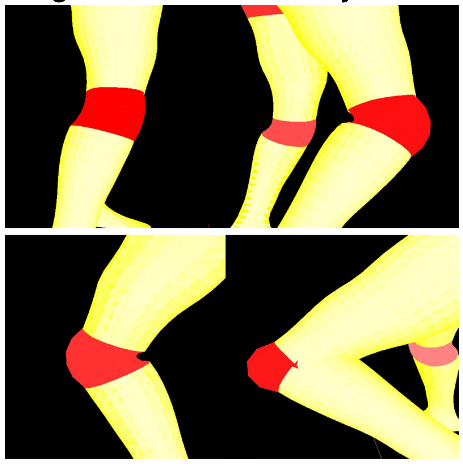
During the implementation and testing of our linear blend skinning algorithm, only one-axis rotations were considered, which provided really promising results. It can be seen in figure 8.

**Figure 8. Knee rotation by 1 axis**



After introducing a full human body animation and taking data from BVH motion data file, we were provided with rotations around all three axes. Although the application of quaternions enabled us to calculate the rotation vector and angle, representing the transformations around all three axes, the final results were not so smooth as in the testing phase. The results can be seen in figure 9.

**Figure 9. Knee rotation by 3 axis**



The solution for this problem could be trying different weight distribution for vertices than those we were using. Other solution, that could give positive results, is cutting out smaller bits of the model.

## 10. Conclusion

The aim of this project was to animate human model using data from the Vicon motion capturing system. For the animation to look smooth, the linear blend skinning technique was chosen. To realize the linear blend skinning algorithm correctly, merits of quaternions were applied as well. Although linear blend skinning is very popular among game developers and other 3D graphics communities, the technique is not perfect. The linear blend skinning implementation in our project showed up with many problems. Usually most of them are solved by manually assigning different weights for each vertex, but this way requires user interruption, whereas our idea was to automatically determine the correct weights.

All in all, our project succeeded in animating the human model in 3D scene. The main goals were reached, though improvements are necessary at some areas.

## 10.1. Future work

Possible further improvements:

1. More detailed animation (moving fingers sand toes);

2. Detailed facial animation;

3. Improved animation around joint areas. Reconsidering the implementation of linear blend skinning technique;

4. Live streaming to our animating program (straight from motion capture system to animation in OpenGL);

5. Other skin deformations (stretching/bulging muscles).

## References

[1] *OBJ specification*. http://local.wasp.uwa.edu.au/ pbourke/dataformats/obj/.

[2] M. J. Baker. *Quaternion algebra*. http://www.euclideanspace.com/maths/algebra/realNormedAlgebra/quaternions/arithmetic/index.htm.

[3] J. Bloomenthal. Medial-based vertex deformation.

[4] F. S. Hill and S. M.Kelley. *Computer Graphics Using OpenGL*. Prentice Hall, third edition, 2007.

[5] L. Kavan and J. Zara. Spherical blend skinning: A real-time deformation of articulated models.

[6] M. Kitagawa and B. Windsor. *MoCap for Artists - Workflow and Techniques for Motion Capture*. Focal Press.

[7] J. LANDER. Skin them bones: Game programming for the web generation. *Game Developer Magazine*, pages 11–16, May 1998.

[8] J. LANDER. Over my dead, polygonal body. *Game Developer Magazine*, pages 17–22, October 1999.

[9] J. P. Lewis, M. Cordner, and N. Fong. Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation.

[10] M. Meredith and S.Maddock. Motion capture file formats explained. 2003.

[11] A. Mohr and M. Gleicher. Building efficient, accurate character skins from example. 2003.

[12] A. Mohr, L. Tokheim, and M. Gleicher. Direct manpulation of interactive character skins. 2003.

[13] Motion Lab Systems, http://www.c3d.org/HTML/default.htm. *C3D Reference*.

[14] S. I. Park and J. K. Hodgins. Capturing and animating skin deformation in human motion.

[15] Vicon Motion Systems Ltd. *Vicon system reference*.

[16] J. Weber. Run-time skin deformation. 2000.

[17] X. S. Yang and J. J. Zhang. Realistic skeleton driven skin deformation.