Animating human model in OpenGL using data from Vicon system

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

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

Abstract

This paper explains how to animate 3D human model in OpenGL, using skeleton-driven deformation technique, similar to Linear Blend Skinning, that computes deformed weighted vertex position, which is calculated using quaternions. This helps to avoid some common animation problems, that occurs while dealing with various human body rigid parts transformations. The motion for this animation was captured using motion capture system and after processing it and converting to appropriate BVH format, was ready to use for animating 3D human model in lowest programming C++/OpenGL level.

Keywords: *Human animation, Vicon motion capture system, OpenGL, C++, 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, what we call skin, it becomes more 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 don't consider many other human body deformations, like stretching or bulging muscles.

1.1. Overview

We present a motion data driven technique of animating a human body model in OpenGL. We have two main issues in this work: getting motion data and using it correctly to animate skeleton and then solving the human model mesh deformation problems. Using data captured by motion capture system and correctly defined joint structure, we achieve realistic body movements for our model. Any motion data, that was captured using same joint structure scheme as ours, can be used for animating that human body model.

After introducing the techniques that we were using in this project (Sections: *2* and *3*), explaining about motion data formats and their differences (Section *6.1*), we give details on how to prepare 3D human body model mesh for our program (Section *7*) and how we managed to animate both skeleton and its underlying layer (Section *8*).

1.2. Previous works

Linear blend skinning is 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 [15], 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 [5, 6] and the artifacts of this technique were discovered very soon [14]. The technique that dealt with some of those issues and set background for linear blend skinning type transformations was introduced in [7] 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 with different shapes, where a one-to-one vertex correspondence must exist between each pair of examples.

Many other used techniques only extends the previous

one. In [9] 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 [15] 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 [3] 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

Linear blend skinning technique is widely used for interactive applications. It goes by many different names, such as Skeleton Subspace Deformation or SSD or "smooth skinning" in Maya.

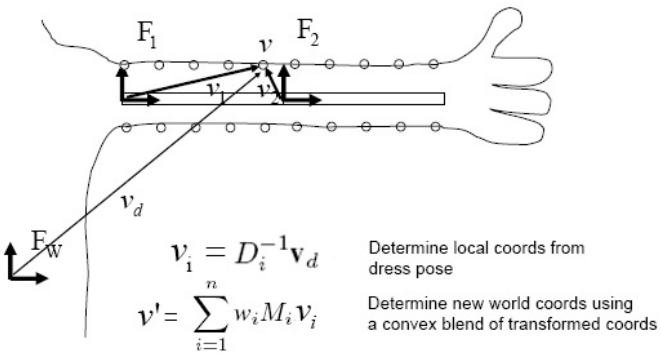
The linear blend skinning algorithm works by first placing a hierarchical skeleton inside a static model mesh of a character in some neutral pose (usually in the da Vinci posture or so called "dress pose"). Then, each vertex is assigned a set of influencing joints and a blending weight for each influence. Computing the deformation in some pose involves rigidly transforming each dress pose vertex by all of its influencing joints. Then the blending weights are used to combine these rigidly transformed positions.

The deformed 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 dress-pose location 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 dress-pose matrix associated with the *i*th influence. Taken together, $D_i^{-1}\mathbf{v}_d$ represents the location of $\mathbf{v}_d$ in the local coordinate frame of the *i*th influence.

Figure 1.



Linear blend skinning though has two primary failings. First, the method is incapable of expressing complex deformations. Artifacts such as the "candy-wrapper", collapse effect on wrists and collapsing around bending joints. They occur because vertices are transformed by linearly interpolated matrices. If the interpolated matrices are dissimilar as in a rotation of nearly 180 degrees, the interpolated transformation is degenerate, so the geometry must collapse. Second, authoring linear blend skins is difficult and frustrating for users.

Despite its failings, this skinning algorithm is very fast and widely supported by commercial applications so it remains popular especially in games and virtual environments.

3. Quaternions

A quaternion is a mathematical object, consisting of 4 scalars. Quaternions were first 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 Euler angles are known to have the disadvantage of being susceptible to "Gimbal lock" where attempts to rotate an object fail to appear as expected, due to the order in which the rotations are performed. Quaternions are a solution to this problem. Instead of rotating an object through a series of successive rotations, quaternions provide the ability to rotate an object through an arbitrary rotation axis and angle. Though the rotation is still performed by using 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 desired rotation matrix.

As mentioned earlier, a quaternion is four numbers, representing one real dimension and 3 imaginary dimensions. Each of it's 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:

$$q = a + i*b + j*c + k*d$$

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. All other operations can be found in the references page of the report.

In order to perform a quaternion multiplication, a quaternion's 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 :

$$(a + i*b + j*c + k*d) * (e + i*f + j*g + k*h)$$

$$\Updownarrow$$

$$a * e + i*a*f + j*a*g + k*a*h + i*b*e - b*f + k*b*g - j*b*h + j*e*c - k*c*f - c*g + i*c*h + k*e*d + j*d*f - i*d*g - d*h$$

$$\Updownarrow$$

$$(a*e - b*f - c*g - d*h) + i*(a*f + b*e + c*h - d*g) + j*(a*g - b*h + e*c + d*f) + k*(a*h + b*g - c*f + e*d)$$

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

$$q1 * q2 \neq q2 * q1$$

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 quarternion's 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:

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

4. Parametric representation of lines in 3D space

A line is a geometric object, defined by two points A and B. It is infinite in length, passing through the points and extending forever in both directions. A line segment (or just segment) is also defined by two points, its endpoints, but extends only from one endpoint to the other. 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 the 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:

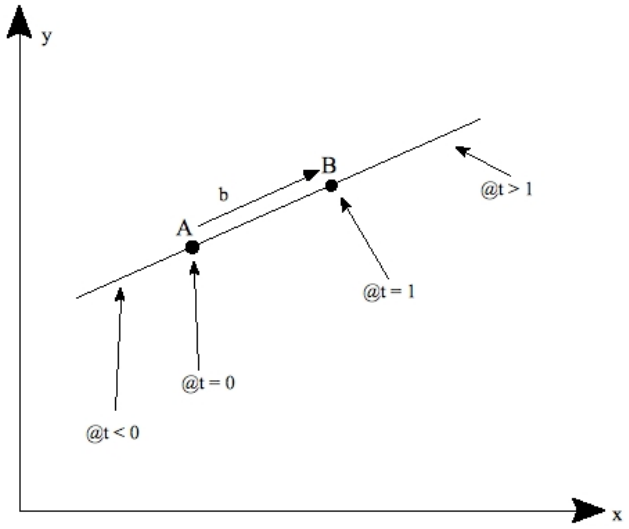
$$L(t) = A + b * t$$

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 name and compute any point along the line L. This is done using a parameter t, that distinguishes one point on the line from another. As t varies, so does the position of L(t) along the line. If $t = 0$, $L(0)$ evaluates to A, so at $t = 0$ we are at point A. At $t = 1$ then $L(1) = A + (B - A) = B$. As t varies, we add a longer or shorter version of b to the point A, resulting in a new point along the line. If t is larger than 1, this point lies somewhere on the opposite side of B from A, and when t is less than 0,

it 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 is that *L(t)* lies a fraction *t* of the way between *A* and *B* when *t* lies between 0 and 1. For instance, when t = 0.5 the point L(0.5) is the midpoint between *A* and *B* and when t = 0.3, the point L(0.3) is 30% of the way from *A* to *B*. This is clear from equation $L(t) = A + b * t$ since $| L(t) - A | = | b | * | t |$ and $| B - A | = | b |$, so the value of $| t |$ is the ratio of the distances $| L(t) - A |$ *to* $| B - A |$.

5. Data formats

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 it's 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. For example "# this is a comment".
- "g" - a group line. Lines, starting with "g" (group) token, determine the start of a group. "g" token is followed by the group name. For example "$gleft_arm$". In our project each group name corresponded to some body part, described by vertices, textures, normals and polygons.
- "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. For example "v -0.756447 0.702621 0.047024" is a vertex with coordinates (-0.756447, 0.702621, 0.047024).
- "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). For example "vt 0.487840 0.942165 0.000000".
- "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. For example "vn 0.149280 -0.186998 -0.240511".
- "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. For example "f 6/9/6 2/5/2 1/1/1" is a triangle polygon, with vertices 6, 2 and 1, textures 9, 5 and 1 and normals 6, 2 and 1.

The above described tokens were of importance in our project, but they are not the only ones that OBJ format allows. The OBJ format specification is much broader and covers such abilities as surface encoding, connectivity between free-form surfaces, rendering attributes. The OBJ specification can be found in the document's reference page.

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. Details about Vicon motion capture system, how we managed to capture and process motion data can be found on the appendix **A**.

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.

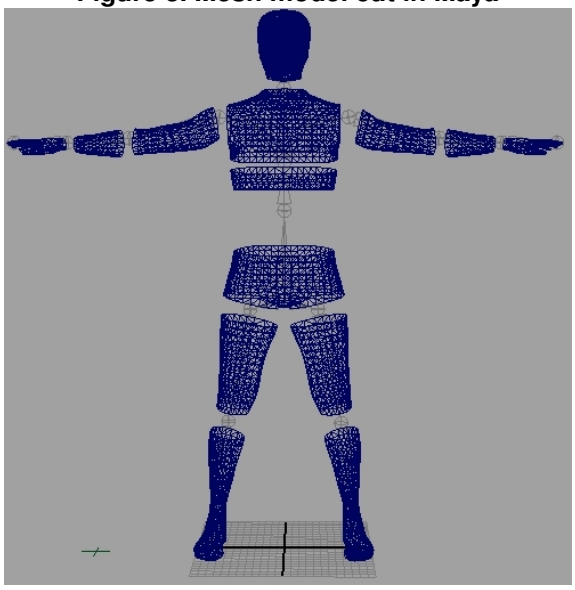
7. Human body mesh model

Human body mesh model must be selected not only by it's 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 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.

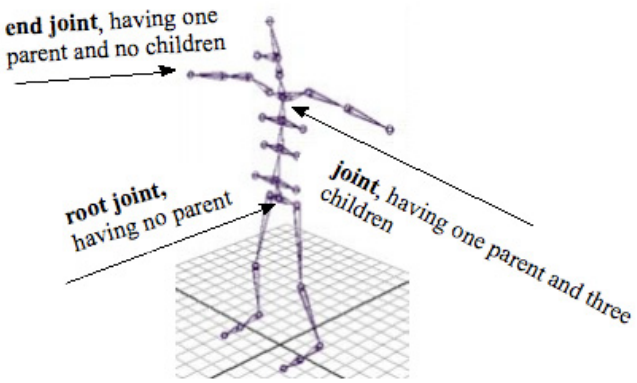
Figure 3. Mesh model cut in Maya



8. Animating human body

The base of any character animation is its skeleton. A skeleton is a hierarchical structure that let you pose and animate your model. Skeletons are composed of series of joints and bones that form joint hierarchies. 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.

Figure 4. Human body skeleton



To animate a character, first of all, the animation data has to be applied to characters 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's bone). 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:

$$M = T * R * S$$

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 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 move individual joints to pose and animate your joint chains. Moving a joint affect's that joint and any joints below it in the hierarchy. For example, if you want a joint chain to reach for a particular location in space, you have to rotate each joint individually so that the joint chain can

reach the location. To do this, you rotate and translate the joint chain's parent joint, then the next joint, and so on down the joint chain.

From a more detailed point of view, each joint has a local transformation that describes its orientation within its local coordinate system, which in turn is a subject to its parent's local orientations. To obtain a global matrix transform for a given joint, the local transform needs to be pre-multiplied by it's parents global transform, which itself is derived by multiplying its local transform with its parent's 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's 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:

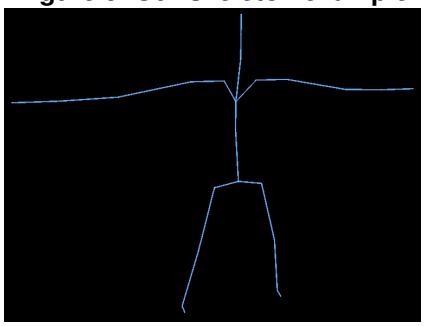
$$M_{global\_left\_ankle} = M_{local\_root} * M_{local\_left\_hip} * M_{local\_left\_knee} * M_{local\_left\_ankle}$$

8.1. Our approach for animating human model

8.2. 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

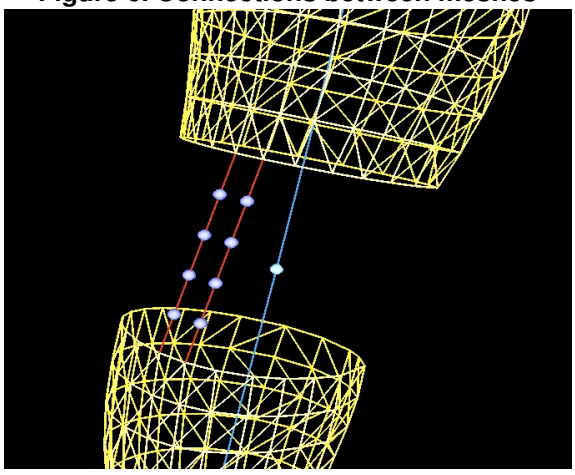


The joints, mentioned above, allow our model to be correctly animated in 3D scene, although much detail is not achieved as the movement of fingers and toes is not animated. Each joint has a mesh applied to it, which is not affected by the animation data, except being rotated or translated in the scene. 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 model's 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 (let's mark it's 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

$$t = n * 1/N$$

Now to calculate the position of subvertices after rotation, we apply the linear blend skinning technique, which means that each subvertex has some weight (let's 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.

Let's 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 it's position on the connecting line:

$$w = P * 1/N$$

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.

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

Animating human skeleton, initial pose is not important, but when we have to deal with human body mesh, it's all they way around. As our human model body mesh was cut in different parts, in order to connect them, we had to have appropriate initial pose. In our case, this pose must be the T-pose, as only in it we can find the shortest distance between two mesh vertices and then connect mesh parts correctly. The detailed way how we connect these vertices is described in section 8.2.

The problem was, that after processing motion data and exporting it to BVH, it wasn't in initial T-pose. The same problem occurred after trying a number of examples from the internet. The initial pose always was with model hands straight down. Due to BVH structure, fixing this problem requires file to be manually edited:

1. Skeleton structure and joint offsets in BVH file has to be changed to match T pose. Basically we had to use forward kinematics from the elbow joint.

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

We managed to automate these two steps, by using third-party application. After changing initial pose using applications' graphical user interface, it has automatically recalculated frame rotations.

9.2. "Exploding knee" problem

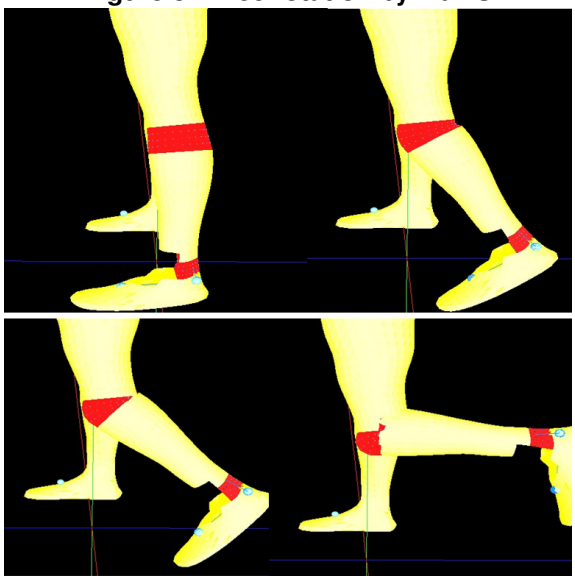
Problem occurred during matrix multiplications, when the difference between frames got very different. Sometimes our BVH file format data gave to frames in a row with Z rotations one 179 other -179. This artifact can be seen in figure 7, where is a motion sequence taken frame by frame.

Figure 7. Knee explosion sequence

9.3. Mesh connections collapsing on complex deformations

While implementing and testing our linear blend skinning algorithm on human model knee with only simple deformations, that only made joint rotation on one of the 3D axis, we were getting really promising results. It can be seen on figure 8.

Figure 8. Knee rotation by 1 axis



Though after introducing full human body animation and taking data from BVH motion data file, we got joint rotations on all 3 axis, what caused totally different final results for our linear blend skinning algorithm. Results can be seen on figure .

Figure 9. Knee rotation by 3 axis

The solution for this problem could be trying different weight distribution for vertices than that we're using now.

10. Conclusion

Conclusion - can be combined with future work

10.1. Future work

What could be implemented to improve this project? Live streaming to our animating program (straight from cameras to animation in OpenGL); other skin deformations

A Details of using Vicon motion system

A.1. Vicon MX motion capture system

Vicon MX motion capture system is a passive optical system that uses markers coated with a retroreflective material to reflect light back that is generated near the cameras lens. The camera's threshold can be adjusted so only the bright reflective markers will be sampled, ignoring skin and fabric.

An object with markers attached at known positions is used to calibrate the cameras and obtain their positions and the lens distortion of each camera is measured. Providing two calibrated cameras see a marker, a 3 dimensional fix can be obtained.

Our system used for this project mainly consists of:

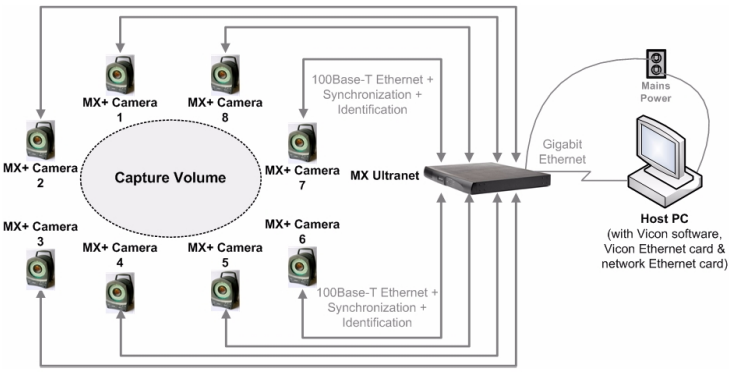
- 8 Vicon MX3+ cameras, fitted with sensitive solid-state sensors, with stringent checks for linearity, sensitivity, and absence of jitter. Each MX camera is programmed with firmware to control its operation and enable it to perform its own onboard grayscale processing. Cameras and their relevant characteristics are recognized immediately when they are plugged in to Vicon MX system.

- MX Ultranet, that supplies power, synchronization, and communications for all eight connected MX cameras. Only MX Ultranet is connected to the Host PC and routes all communication to and from the host PC, and timing/synchronization signals to and from the MX cameras connected to it.

- Host PC, with seperate Ethernet port (preferably Gigabit ethernet card) to enable communications between the Vicon software installed on this *host pc* (in our case *Vicon IQ 2.5*) and *MX Ultranet*. No other special requirements for Host PC, but processor and memory are vital for being able to achieve better results. Our *host pc* had 2.49GB of RAM and 1.60GHz Intel Xeon 8 core processor. *Vicon IQ* software on a *host pc* is used to collect and process the raw video data. It takes the two-dimensional data from each camera, combining it with calibration data to reconstruct the equivalent digital motion in three dimensions. This can be viewed as a virtual three-dimensional motion. After this reconstruction the data may be passed to other applications.

A.2. Motion system preparations

Vicon motion capture system high-resolution cameras are situated in a circle, above the maximum height of any



marker that would be captured, that forms a motion capture space. Then the capture volume is created. The capture volume is three-dimensional, but in practice is more readily viewed by its boundaries marked out on the floor. This gives you a guide for the positioning of the cameras, and your subjects a guide as to where they can move. The area should be as central to your capture space as possible to maximize the volume you are able to capture. Initially you are looking to see whether the camera is positioned and oriented for the maximum view. Then every single camera can be adjusted separately by changing software settings, to achieve best result.

After setting up cameras, the vital thing is system calibration. It allows the software to calculate the relative location and orientation of all the cameras. When your movements have been recorded the reconstruction process uses these measurements to calculate the accurate movements of the markers through space.

This is done by static and dynamic calibrations:

- Static calibration - calculates the origin or center of the capture volume and determines the orientation of the 3D Workspace.

- Dynamic calibration - involves movement/waving of a calibration wand throughout the whole volume and allows the system to calculate the relative positions and orientations of the cameras. It also linearizes the cameras.

A.3. Capturing motion data

Having the whole motion system ready for use, the next first step is placing all the retroreflective markers on special suit. The special suit is used to achieve the most accurate results in motion, that motion system gets by tracking those reflective markers and it should be as as tight as possible. Though the suit is not always used while capturing motion data, as for example capturing face motion data, special reflective markers are used, that are put directly on subjects

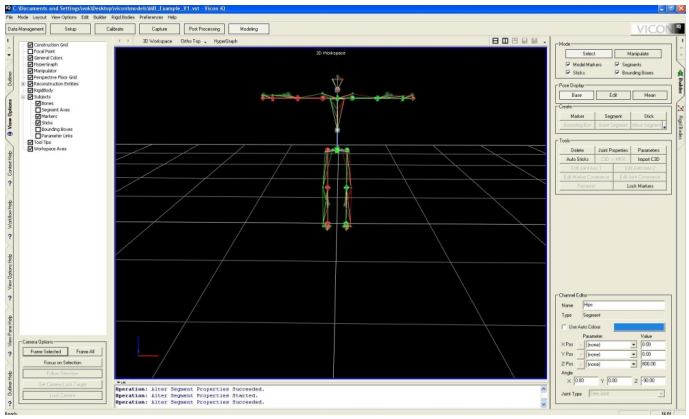
face. Placing markers is vital that the animating template is already selected. In this case, skeleton template, that will be assigned with motion data points. These templates describes the bone structure, joint positions and markers positions, how these markers relate to the bone structure and joints.

When markers are ready, it's time to start capturing data. Motion capture subject should stand in the middle of the circle in the capture area, with his back straight, arms straight out with palms facing down, and feet forwards. This is called a T-pose, and every capture session should begin and end every in it. This pose is common in animating human bodies, as it allows meshes of the skin to be connected correctly. Also it usually allows the motion capture system to determine all the markers positions right from the frame 1, as all of them are in its visibility area. While capturing, you should be aware, that during whole capture process, you should be in cameras capture area.

A.4. Captured data post-processing

After capturing motion data, in order to proceed further, post-processing must be done. Firstly it involves labeling/matching each marker from captured data to our used animation template (skeleton template) at the first frame. Only by labeling these markers, we are getting logical motion data in further frames, and not some meaningless points in the 3D world. This is the stage, when it's necessary that those reflective markers were situated correctly on the motion capture subject.

Though labeling does really big part in getting correct motion data, it's very important to do some manual tweaking of the labeled points and track unlabeled ones. They can be unlabeled either of some motion capture failure at some point or there can be a "ghost" points that needs to be deleted. Ghost points are quite common while using passive optical motion capture system with retroreflective markers.



Without manual fixing, special pipelines are used to automate various processes, for example using kinematic fitting to match our animation template to our labeled markers. Various pipelines can be used to significantly shorten the post-processing time.

References

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

[2] F. S. Hill and S. M.Kelley. *Computer Graphics Using OpenGL*.

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

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

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

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

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

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

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

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

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

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

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

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

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