CS229 Project 1: Motion Capture and Splines

Assignment Out:	Sept. 11th, 2000
Assignment Due:	Sept. 22nd, 2000 (11:59 pm)

---

1 Introduction

Motion capture is considered an essential tool in modern computer animation. Almost no videogame company today is without its own motion capture studio, and breathtaking scenes like the long camera fly-over across the deck of the Titanic would be impossible without a cast of virtual characters driven by motion capture data.

Characteristics of the video game and movie domains are that the character's motion can be tightly scripted, and that many animator hours can be invested in tailoring the captured data to the needs of the game or movie. Animators can create believable transitions between motion clips to be used in sequence in a video game, and motion capture data used in movies is painstakingly edited before it hits the screen. The results are convincing, but the process is laborious.

Much effort has been put into making motion capture data require less hand-tuning. There is speculation in the research community that motion capture data can be used as a database of examples from which a virtual character can draw to create a fluid and consistent personality. The canonical vision is to pipe a bunch of old Bogart movies into the computer overnight and come in the next morning to an animated Humphrey Bogart character with recognizable motion and mannerisms. [1]

In its raw format, however, motion capture data is voluminous and inflexible. Even simple modifications like changing the direction in which a character is looking are tedious to make by hand when working with the data in its original form. Motion capture ("mocap") data consists of a set of parameters sufficient to describe the pose of a skeleton at discrete points in time, typically 60 frames per second. The same amount of data is created regardless of the nature of the motion. One second of idle standing produces just as much data as one second of a complex ballet maneuver in *Swan Lake*. For this reason, editing raw motion capture data is just as painful as editing keyframes in traditional cel animation by erasing and recreating portions of each drawing (and here we have 60 "drawings" per second of animation). Every single frame of the motion that is affected must be edited to smoothly work in the new motion. Given the unwieldy heft of mocap data, it is no wonder that companies like Pixar shun motion capture – an artist modifying mocap data with a particular goal in mind might be better off starting from scratch!

---

[1] And possibly to a sit-in protest from a Screen Actor's Guild concerned that their jobs are about to become automated.

A typical first step to making motion capture data easier to work with is to create a continuous representation of the data, rather than working with the set of discrete snapshots that is provided as input. A continuous representation can easily be scaled in time if necessary and segmented into high frequency and low frequency components. Changing the direction of a gesture, for example, can then be achieved by editing the low frequency components, leaving the high frequencies alone. The low frequency components, being slow-changing, can often be adjusted effectively by modifying relatively few parameters (e.g. moving a few control points in a spline). The high frequency components, on the other hand, are often considered responsible for the perceived "naturalness" of a motion, so leaving those intact may lead to better results. Animation created wholly on computer from a set of keyframes often lacks these high frequency components, leading to motion that is uncannily smooth and obviously artificial. Accurately capturing these subtle details of real motion is part of the appeal of the motion capture process.

2 Assignment Overview

For this first assignment, you will create a continuous representation of a motion capture dataset by fitting a B-spline to that dataset and then modifying the motion slightly to produce a continuous, repeating motion. The data is of a human figure running a few strides across a room.

You are given the skeleton of a viewer, implemented using Java3D. To complete the assignment, you will need to:

- Use data from the skeleton file to display the skeleton.
- Use data from a motion file to play back the raw motion capture data.
- Fit splines to the motion capture data.
- Create a cycle from the data using the spline representation, and play back the resulting motion. The cycle created from the running data should cause the character to continuously run forward, not pop back to its starting point. You will have to specify the start frame and end frame for the cycle. The beginning and ending of a single stride mid-flight (the "ballistic phase") will probably give the best results.

Implementation details and file locations are at the end of the handout. If you choose not to use a B-Spline as a continuous representation for your motion, explain the motivation for your choice and describe briefly how you implemented your solution. By the way, you will probably want to reuse this code in future assignments, and it might also make a great base for a final project.

2.1 Questions

Please hand in the answers to the following questions in a README file along with your completed assignment.

1. Explain your approach to creating a repeatable cycle from the motion data. How did you modify how you created the continuous representation?

2. How would you implement a tracking camera that would follow the character and remain upright? Provide details about how you would compute the transform used to set the camera position at each frame.

3. Try your algorithm on datasets other than running (e.g. some of the kicking motions) and try to loop them. Pick one of the more disappointing results and describe how you might improve it.

4. Compare motion from the correct skeleton file (`body.asf`) to the that from the incorrect skeleton file (`bodyWrong.asf`). In particular, notice that the foot slides forward while in contact with the ground when the incorrect skeleton file is used. This artifact is not present in the original motion. How would you fix the problem? (In this case, the only change that has been made to the skeleton is to shrink the upper and lower legs to 60% of their original size.) Make your solution as general as possible (e.g. it should also work for a skeleton 10% the size of the original).

2.2 Bonus Project

If you read the Introduction carefully, you may be asking "so, what happened to this cool frequency decomposition idea?" You can do a version of frequency decomposition using a hieraracy of B-Splines instead of a single B-Spline to represent the motion. Impressive results for image blending have been achieved using frequency decomposition, and creating repeatable cycles using hierarchical B-Splines may similarly improve the appearance of your final motion. See Lee, Wolberg, and Shin '97 from the reading packet for information on hierarchical B-Splines, and see http://www-bcs.mit.edu/people/adelson/publications/abstracts/spline83.html for a blending application in vision (using a Laplacian pyramid rather than a hierarchical B-Spline approach). Definitely check out the images in the latter paper. The results are amazing.

Your handin should include some plots that convince us that your approach works. How did you decide when to stop (i.e. how did you decide how many levels of hierarchy were needed for each parameter)? Which joints required the largest and smallest numbers of B-Splines to adequately represent them?

This project is worth up to 25 bonus points for an exceptional implementation and report. (The project is scored out of 100 points.)

If you have other ideas for a worthy bonus project, run them by us!

3 Technical Background

One goal of this project is to give you a chance to work with the complex scene graphs that are used in animation. This section covers the technical background you will need to build the correct scene graph from the given skeleton and motion files.

3.1 Points and Transformation Matrices

One operation we will use constantly is that of transforming object geometry (e.g. vertices) from one coordinate frame to another. This section defines some notation for points and transformation matrices. For more detail on transformations and transformation hierarchies, refer to Watt and Watt or an introductory graphics text such as Foley, van Dam, Feiner, and Hughes.

In this document, points will be represented with an upper-left superscript indicating their current coordinate system. For example, point ${}^{O}\mathbf{p}$ is a point expressed in coordinate frame O. We use homogeneous coordinates:

$$\mathbf{p} = \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \\ 1 \end{bmatrix}, \qquad \mathbf{v} = \begin{bmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \\ 0 \end{bmatrix} \tag{1}$$

where $\mathbf{p}$ is a point, and $\mathbf{v}$ is a vector.

A matrix transforming points from one coordinate frame to another is expressed with a subscript indicating origin frame and a left superscript indicating destination frame. For example, the transform ${}^{W}M_O$ converts points in coordinate frame O to points in coordinate frame W:

$${}^{W}\mathbf{p} = {}^{W}M_O \; {}^{O}\mathbf{p} \tag{2}$$

Transformation matrices include both translations and rotations. It will often be convenient to separate out pure translations and rotations about coordinate axes. We will use the following notation for translations:

$$T(tx, ty, tz) = \begin{bmatrix} 0 & 0 & 0 & \mathrm{tx} \\ 0 & 0 & 0 & \mathrm{ty} \\ 0 & 0 & 0 & \mathrm{tz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Rotations about the x, y, and z axes respectively are:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5}$$

$$R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6}$$

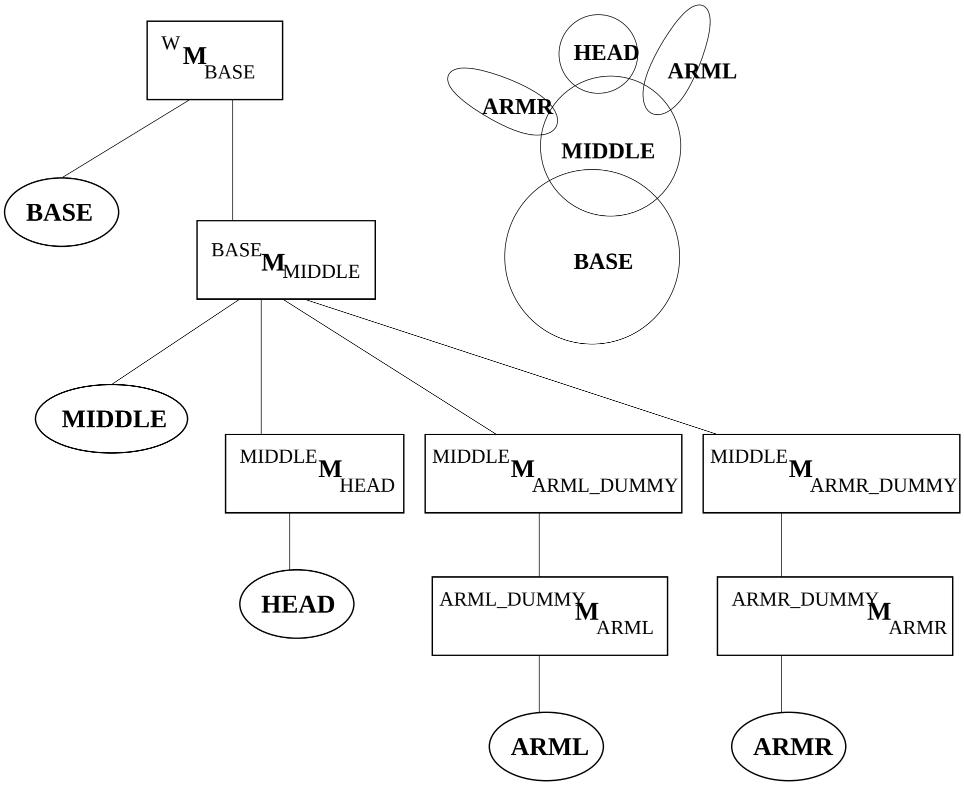

Figure 1: A simple character and scene graph.

Figure 1 shows a simple scene graph. Transformations are cumulative, so a point in local coordinate frame $HEAD$ can be converted to a point in world space as follows:

$$^{W}\mathbf{p} = {}^{W}M_{BASE}\ {}^{BASE}M_{MIDDLE}\ {}^{MIDDLE}M_{HEAD}\ \ {}^{HEAD}\mathbf{p} \tag{7}$$

For a point in local coordinate frame $MIDDLE$, we have:

$$^{W}\mathbf{p} = {}^{W}M_{BASE}\ {}^{BASE}M_{MIDDLE}\ {}^{MIDDLE}\mathbf{p} \tag{8}$$

The viewer we provide will display a scenegraph and allow you to move and rotate the camera to view the world. Your mission is to parse the ASF file and create a skeleton scene graph from the information contained in that file.

3.2 Interpreting the ASF File

ASF stands for "Acclaim Skeletal File." This file format was developed by Acclaim for use with its own motion capture process, and was later placed in the public domain. It is one of a handful of skeleton file formats commonly used to distribute motion capture data.

The information below will help you create a viewer that will read in all of the data we have available for the class. It will not be sufficiently general to allow you to read in all valid ASF files.[2]

3.2.1 The Hierarchy

At the end of the ASF file is the scene graph hierarchy. Each line of input lists either the root or a bone, followed by all of its immediate children. For the character in Figure 1, we would have:

```
:hierarchy
begin
    root BASE
    BASE MIDDLE
    MIDDLE HEAD ARML_DUMMY ARMR_DUMMY
    ARML_DUMMY ARML
    ARMR_DUMMY ARMR
end
```

The parent on a line must have been referred to as a child on a previous line. The only exception is the root, which must come first. Bodies ARML_DUMMY and ARMR_DUMMY are not displayed in Figure 1. We'll see why they are useful later.

[2] For hints on how to create a more general ASF file reader, see the documentation file `file:/course/cs229/info/Acclaim_Skeleton_Format.html`.

3.2.2 The Root

At the beginning of the file is a section beginning with keyword `:root`. This section contains offset information for the root of the system. The offset information is zero in all of our datasets, and so you can safely ignore it.

The `:root` section also states the order in which data to translate and rotate the root will be listed in any AMC motion file associated with this skeleton. The data order is the same in all of our datasets: x, y, and z translations followed by x, y, and z rotations.[3] You may assume this order in your code.

The bottom line is that you can completely ignore the `:root` section for all of our datasets.

3.2.3 The Bones

Following the `:root` section is a section labelled `:bonedata`. This section contains everything you need to know to draw the skeleton of the system in a default pose, and to interpret the data in an AMC motion file to drive each of the bones.

Direction and Length. Every bone of the system has a local coordinate frame (Figure 2). The `direction` and `length` keywords together define the vector from a bone's local coordinate origin to that of child bones — in other words, a vector to the far tip of the bone, where other bones can branch off. Each bone has only this single point for child bones to be attached. Suppose we have the following bones:

```
:bonedata
begin
    id 1
    name BASE
    direction 0 1 0
    length 3
    dof rx ry rz
    axis 0 0 0
end
begin
    id 2
    name MIDDLE
    direction 0 1 0
    length 3
    dof rx ry rz
    axis 0 0 0
end
begin
    id 3
```

[3] Each of these transformations occurs along fixed, global axes.

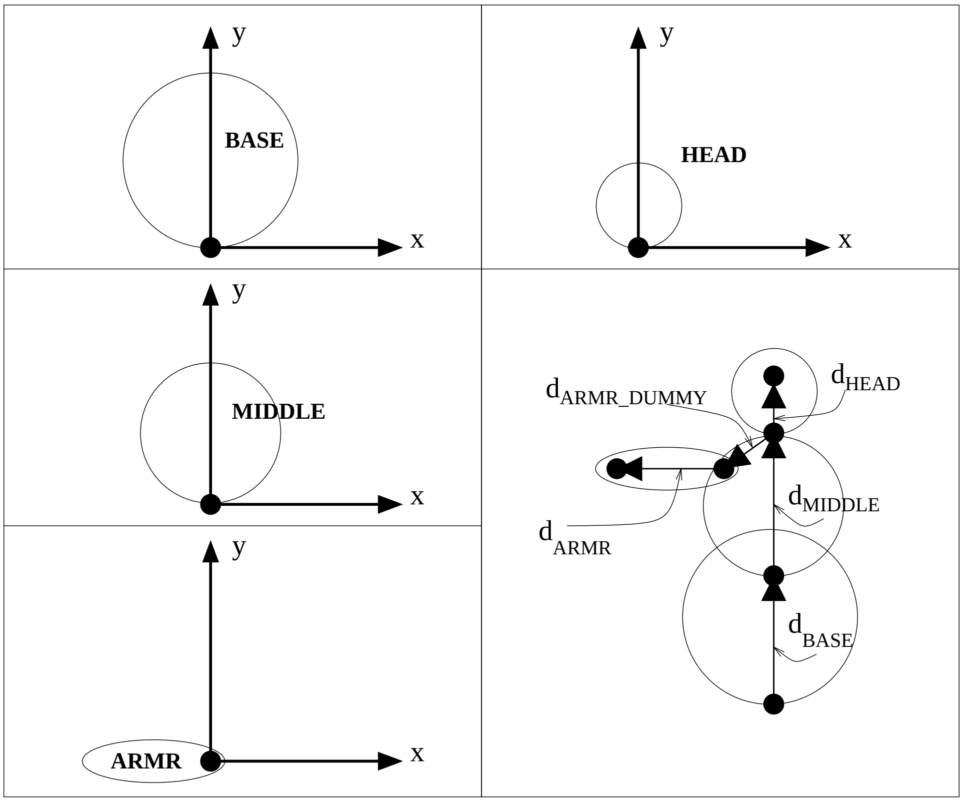

Figure 2: Bone placement: local coordinate frames and use of direction and length information to draw the 'home' or default-pose skeleton.

```
    name ARMR_DUMMY
    direction -0.7 -0.7 0
    length 1.5
end
begin
    id 4
    name ARMR
    direction -1 0 0
    length 3
    dof rx ry rz
    axis 0 0 90
end
begin
    id 5
    name HEAD
    direction 0 1 0
    length 1
    dof rx ry rz
    axis 0 90 0
end
```

From this data, we define the following vectors by scaling direction by length for each of the bones:

$$d_{BASE} = [\,0 \quad 3 \quad 0 \quad 0\,] \tag{9}$$
$$d_{MIDDLE} = [\,0 \quad 3 \quad 0 \quad 0\,] \tag{10}$$
$$d_{ARMR\_DUMMY} = [\,-1.05 \quad -1.05 \quad 0 \quad 0\,] \tag{11}$$
$$d_{ARMR} = [\,-3 \quad 0 \quad 0 \quad 0\,] \tag{12}$$
$$d_{HEAD} = [\,0 \quad 1 \quad 0 \quad 0\,] \tag{13}$$

These vectors allow us to draw the skeleton in its home or zero pose, as shown in Figure 2. Note that all direction vectors are specified in the global reference frame with rotations about all the joints set to zero. The reason for the ARMR_DUMMY and ARML_DUMMY bones should be clear from Figure 2. They allow the HEAD, ARMR, and ARML bones to appear to exit the MIDDLE bone at different positions rather than simply all at the tip, addressing the restriction of having a single attachment point for each bone.

DOF. The next keyword to notice is the `dof` (degrees of freedom) keyword. Bones that have this keyword can rotate about the origin of their local coordinate frame. Bones that do not have this keyword are rigidly connected to their parent. In the dataset above, only the ARMR_DUMMY bone is rigidly fixed to its parent, the MIDDLE bone.

The `dof` keyword also specifies the order in which joint rotations may be read from the AMC motion file. Any bone in our data set that is not rigidly fixed to its parent will have degrees of

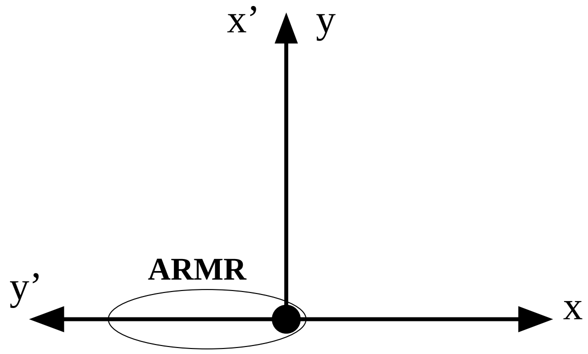

Figure 3: The axis entry for bone ARMR is "axis 0 0 90," indicating a rotation of 90 degrees about z. This rotation produces the local rotation axes x' and y' shown. (z' is identical to z and points out of the page.) Rotations in the AMC motion file specify rotations about these axes. For example, a rotation "ARMR 90 0 0" is a rotation of 90 degrees about x', and will cause the arm to point out of the page, toward the reader.

freedom in the following order: rx ry rz (or rotation about the x axis, followed by rotation about the y axis, followed by rotation about the z axis). You may assume this ordering in your code.

Axis. The last keyword of interest is the `axis` keyword. Rotations read from the AMC motion file represent rotations about local axes. The parameters specified on the axis line define an offset Euler rotation from a bone's coordinate frame to this local orientation, as described in Figure 3. Note that the `axis` offset *does not* affect the coordinate frame of child bones; `axis` is simply used for specifying the axes about which subsequent motion specified in the AMC file will occur.

3.2.4 Building the Scene Graph

Given the bone information outlined in the previous section, we need to define the transformation matrices used in the scene graph in Figure 1. In this section, we first break down one of the transformation matrices, ${}^{MIDDLE}M_{HEAD}$ and then show an expanded version of part of the scene graph.

Bone data for the MIDDLE and HEAD bones is given in the previous section. The steps involved in connecting the HEAD to its parent MIDDLE are shown in Figure 4. The example in Figure 4 assumes that the HEAD rotation read from the AMC file was a rotation of -90 degrees about x:

HEAD -90 0 0

Note that the observed rotation in Figure 4 is not -90 degrees about HEAD's x axis, but is instead -90 degrees about HEAD's x' axis. Axis x' is obtained by observing that HEAD's axis entry in the `:bonedata` section is '`axis 0 90 0`'. This entry specifies an offset rotation of 90 degrees about y. Creating a new coordinate frame with axes x', y', and z' by rotating the HEAD local frame 90 degrees about y places x' in the direction of $-z$. A -90 degree rotation about x' therefore amounts to a 90 degree rotation about z, as shown in the center picture in Figure 4.

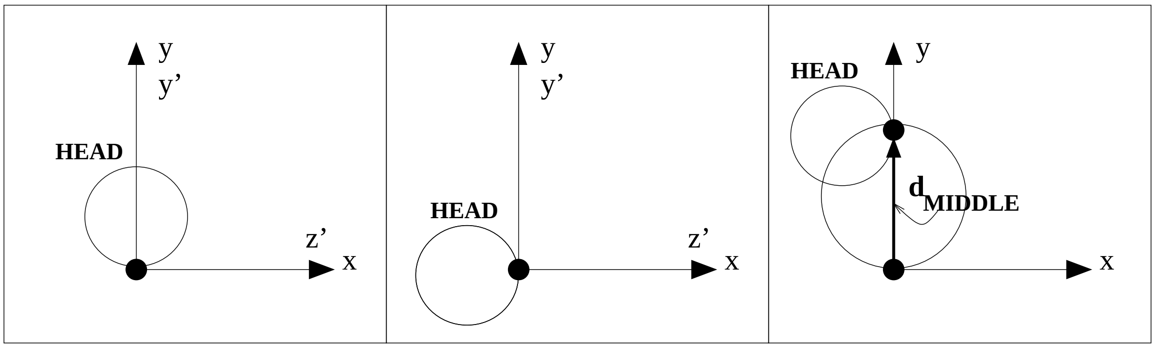

Figure 4: Steps in attaching the HEAD to its parent MIDDLE. (L) HEAD in its local coordinate frame. The `axis` keyword for head specifies a rotation of 90 degrees about y, resulting in the local rotation axes y' and z' shown. (C) Rotate a specified amount about the local axes. The given rotation is -90 degrees about the x' axis. Because x' points into the page, we rotate HEAD 90 degrees to the left. (R) Translate HEAD to attach it to its parent. The required translation is the direction vector of the parent, or d_{MIDDLE}.

In a more general situation, given local axis

`axis` $\alpha\ \beta\ \gamma$

and given the MIDDLE direction and length vectors

```
direction dx dy dz
length l
```

and given rotation from the AMC file

```
HEAD rx ry rz
```

we create a cumulative transform as follows:

$$L = R_z(\gamma) R_y(\beta) R_x(\alpha) \tag{14}$$

$$^{MIDDLE}M_{HEAD} = T(l * dx, l * dy, l * dz)\ L\ R_z(rz)\ R_y(ry)\ R_x(rx)\ L^{-1} \tag{15}$$

Note that the cumulative transform has the following steps (read right to left):

- Undo the local axis rotation L (defined based on the HEAD axis entry).
- Perform the rotation from the AMC file $R_z(rz)R_y(ry)R_x(rx)$.
- Redo the local axis rotation L.
- Translate HEAD $T(l*dx, l*dy, l*dz)$ (defined based on MIDDLE distance and length entries).

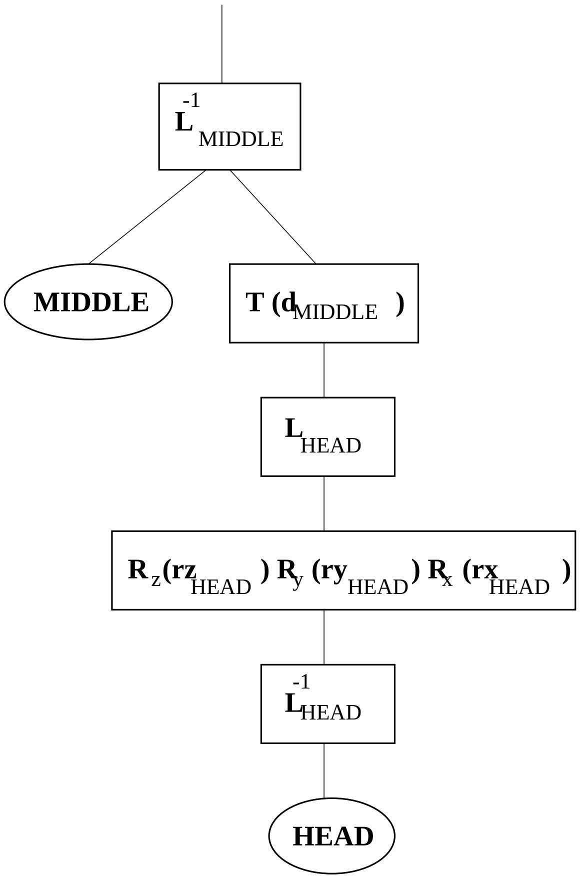

Figure 5: An expanded section of the scene graph depicting the portions of the transform from HEAD to MIDDLE. Only parameters rz_{HEAD}, ry_{HEAD}, and rx_{HEAD} are animated. All other transformations in this figure are fixed.

Depicted graphically, these transforms create the expanded portion of the scene graph shown in Figure 5. Note that most of the transforms are fixed. Transform $R_z(rz_{HEAD})R_y(ry_{HEAD})R_x(rx_{HEAD})$ is animated by varying parameters rx_{HEAD}, ry_{HEAD}, and rz_{HEAD} to rotate HEAD with respect to MIDDLE. Parameters rx_{HEAD}, ry_{HEAD}, and rz_{HEAD} are of course read frame by frame from the AMC file.

3.3 Interpreting the AMC File

Once you understand the skeleton format, the AMC file format should seem fairly straightfoward. The file provides data specifying the character pose at each frame, by providing values for each of the degrees of freedom for each bone. The data for each frame is preceeded by the frame number. There is a line for the root that contains the root translation (tx, ty, tz) followed by the root rotation (rx, ry, rz). There is also a line for each bone containing its local rotation (rx, ry, rz). Note that translations are in centimeters and rotations are in degrees.

4 Implementation

This being a CS class and all, the TA's have supplied you with a bit of support code to get you started off and to save you some drudgery. This code is available in `/course/cs229/asgn/mocap`, and implements the beginning of a mocap viewer in Java3D. It has a basic Swing-based GUI for viewing animations, a groundplane, a trackball camera interface, and basic lexical support for the ASF/AMC formats with stubs for you to insert your code. Given this start, you should be able to focus your energies on creating a scenegraph of the skeleton, playing back the keyframes, and building a continuous looping form from the data.

4.1 Java3D

As mentioned, the support code uses Java3D. Many of the ideas in Java3D come from existing packages (OpenGL, Inventor, Direct3D, World Toolkit), and will be familiar to anyone who has used one of these. Coming from an OpenGL background, the TA's experienced a "mostly pleasant" learning process, thanks largely to the excellent documentation. The support code takes care of much of the nitty-gritty, but there are two main aspects of Java3D (javax.media.j3d) which you'll need to get familiar with: the vector math package (javax.vecmath) and the process of constructing a scenegraph.

For a comprehensive introduction, the best place to get started is the *Java3D API Specification.* There are a few printed copies floating around the CIT, including one in the back of the Sun Lab and probably a few in Graphics. It is also available as HTML online — look for the link at the end of the course webpage.[4]

Additionally, there is the javadoc hypertext reference for Java3D [5] and for Java itself [6]. These

[4] http://java.sun.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dguide/index.html
[5] file:/course/cs229/info/j3d.docs/index.html
[6] file:/pro/java/java/docs/api/index.html

are invaluable references while coding. Both of these links are available on the course webpage as well.

4.1.1 javax.vecmath

The vector math package has a comprehensive array of datatypes in varying precisions. If you've not seen it before, start off by taking a look at the Vector3d class in the javadoc reference. All the standard vector operations are here (or inherited), including useful things like linear interpolation and in-place multiply-add. For efficiency reasons, many of the vecmath methods operate on function parameters (including `this`), rather than returning temporary objects. The javadocs have full details on call semantics. Feel free to post to the 229 newsgroup if you are having difficulty achieving some result.

You should be able to stick with keeping everything as doubles rather than floats, which will save your having to convert things back and forth.

4.2 Scenegraphs in Java3D

You should only need a few Java3D node types in creating your scenegraph: `TransformGroup`s, which apply a Transform3D to their children, and `Shape3D`s, which represent instances of geometry.

Adding scenegraph nodes is straightforward: just create the object and call parent.addChild(child). Say you have a node in your scenegraph, let's call it root. The following code creates a sphere and attaches it to the root, translated by the Vector3d "where":

```
import com.sun.j3d.utils.geometry.*;  // various useful geometry primitives
...
Vector3d where = new Vector3d( 2, 3, 5 );
TransformGroup transGroup = new TransformGroup();
Transform3D trans = new Transform3D();
trans.setTranslation( where );
transGroup.setTransform(trans);
transGroup.addChild( new Sphere(4) );
root.addChild( transGroup );
```

In creating your skeleton, feel free to use the primitives in com.sun.j3d.utils.geometry.*, or create your own geometry. The demo uses a combination of spheres at joint positions and cylinders oriented along the bone vectors.

One important thing to note: the support code, after calling to create the skeleton, calls compile() on the entire scenegraph, to maximize display speed. After the scenegraph has been compiled, Java3D will balk and throw an exception if you try to modify any of the TransformGroup nodes in the graph — unless you have marked them as modifiable:

`setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);`

Look at the Axes and Groundplane classes for more example code of creating Java3D scenegraphs.

4.3 Debugging Hints

If something goes awry and Java throws an exception and dumps a stack trace, you may see references to "compiled code" where you would normally hope to see line numbers. All is not lost. This is just an artifact of the just-in-time compiling going on under the hood. Go back and

```
setenv JAVA_COMPILER none
```

and re-run your program. It should give you a fully-detailed stack trace.

4.4 Otherness

The trackball camera interface uses the right mouse button. Start a drag near the border of the window to rotate the view. Start a horizontal drag near the center to pan in the film plane; start a vertical drag near the center to dolly in/out.

All data files for this assignment are in `/course/cs229/data/mocap`. The skeleton is in `body.asf`, and the motion files are `*.amc`. The skeleton file for question 4 is `bodyWrong.asf`. An index to the motion files is in `MotionIndex.txt`.

The files 'compile' and 'run' are scripts to compile/run the program with the proper binaries and CLASSPATH. Feel free to switch to a Makefile or what-have-you.

A demo viewer for the mocap data is in `/course/cs229/demos/mocap`. Go to that directory and execute 'run'.

Finally, while there are many ways to approach the assignment, one possible course of actions is as follows:

1. Display a simple primitive in the viewer.
2. Apply transformations to that primitive.
3. Read in the skeleton file. (Note that the `load` method of the `Skeleton` class already does most of the work of reading in the ASF file.)
4. Create the scenegraph and display the skeleton.
5. Read in the motion file. (Note that the `load` method of the `SkeletonAnimation` class already does most of the work of reading in the AMC file.)
6. Apply some frame of the motion directly to the skeleton.
7. Animate the skeleton directly with the raw motion.
8. Represent the motion with B-splines and create the repeating run cycle.

5 Handing in

Run "`/course/cs229/bin/cs229_handin mocap`" from your project directory.