WORKING WITH
MOTION CAPTURE
FILE FORMATS

## BY JEFF LANDER

oy am I glad that 3D acceleration hardware is here to stay. I'm sure you all feel as liberated as I do by not having to write all that basic polygon stuff. Clipping, sorting, and drawing pixel-by-pixel is about as dull as 3D programming gets. Now I have all this great hardware to do the mind-numbingly dull texture mapping and Z-buffering for me. I also have the render speed and horsepower to do some really interesting stuff. What am I going to do with all this spare time? Really cool real-time 3D characters!

Sure, we've all seen real-time 3D characters. We've even seen real-time 3D characters with a restricted use of animation. However, there have been so many limitations,

Jeff Lander is a Digital Evolutionist at Darwin 3D, where he crafts technology for the future of gaming, entertainment, and network communication. He can be reached at jeffl@darwin3d.com.
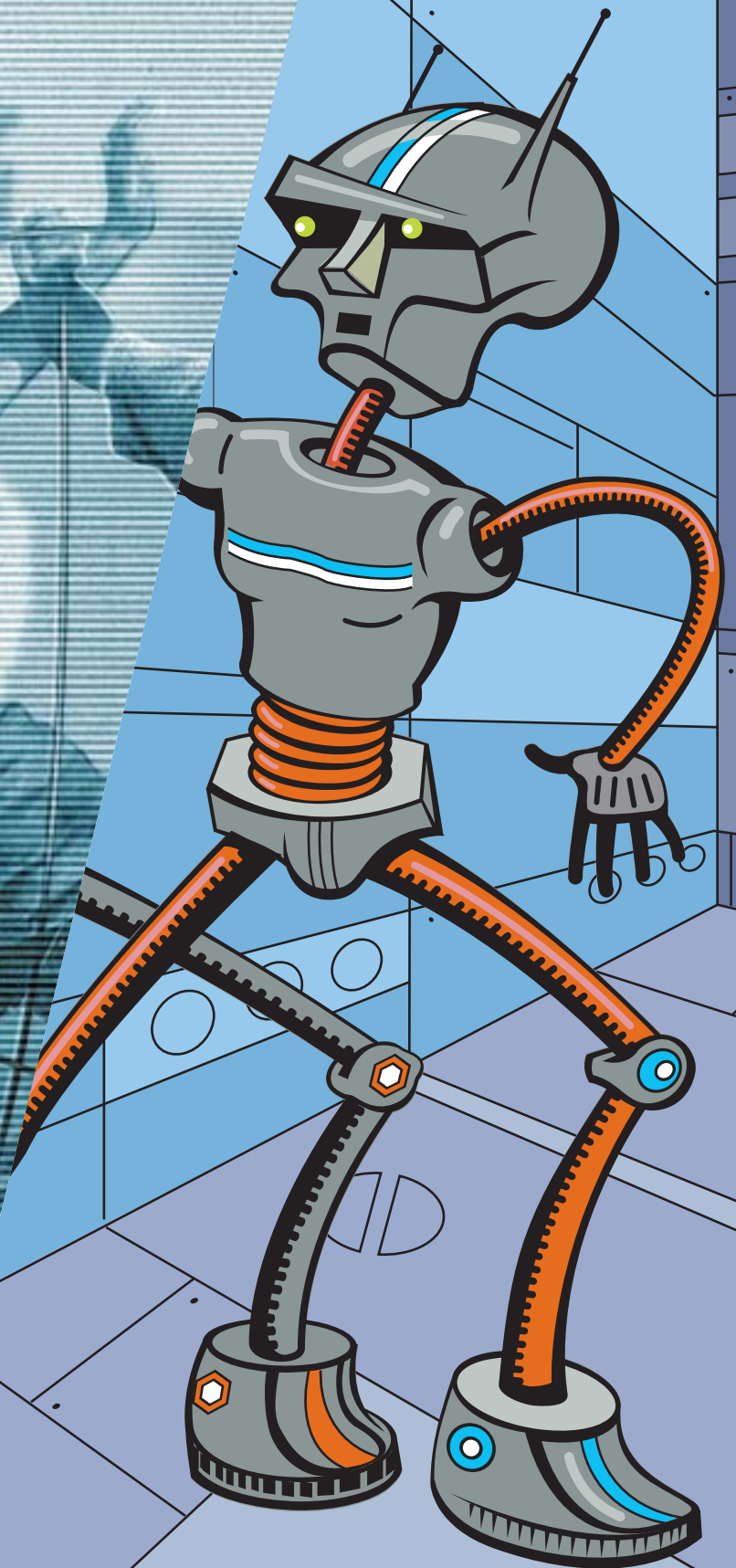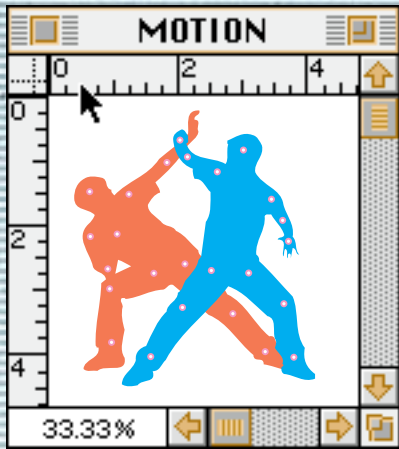
and we all want so much more. We want realism, but how do we go about fulfilling these sick desires? The answer: motion capture.

Motion Capture

Let's face it, motion capture is hot. In the last couple of years, motion capture has spread everywhere — from movies to television commercials, from sports titles to action games, even to click-and-explore adventures. Publishers are climbing all over each other trying to get the words "Motion-Captured 3D Characters" on their boxes. A lot of hype has been loaded onto those words, often to the player's disappointment. As usual, our expectations exceed what the technology can truly deliver. But we're getting so much closer; we have new ripping hardware and the experience from past-generation motion capture down-falls. Yet the desire for more keeps increasing.

The hype has gotten so over-the-top that for the last couple of years, I've been threatening to put out VIRTUA HANGMAN as a demo at E3. I can just see it, these realistic real-time 3D characters marching up to the gallows as you relentlessly guess letters. If we want to be cliché, we could even have a 3D character turning the letters. Now *that* would be an excessive use of technology.

While I wouldn't consider using motion capture for characters better suited to traditional keyframing, motion capture technology clearly has a place in game development. Luckily,

the techniques needed for programmers to apply motion capture data to real-time characters work equally well with any type of animation data, be it keyframed, motion captured, or animated through procedural dynamics.

The Need

Let's imagine a scenario in which your brilliant producers have assigned you, the programmer, to develop a real-time 3D character-based game. They have charged you with the tasks of designing the game engine and creating the production pathway. For a variety of design, budgetary, and staffing reasons, you've decided to use motion capture to supply the bulk of your animation data.

Your first task is to decide where you're going to get this data. It doesn't really matter whether you have your own capture setup or a service bureau is doing it for you — plan on plenty of cleanup time. Motion capture is not simple. The data needs quite a bit of massaging to get it ready for the game, and you can get in trouble by underestimating the amount of post-production work the data needs. You also need to be aware that motion capture data is specific to the hierarchy and body dimensions of the person captured. It's possible, but tricky, to scale this motion to other body types and sizes. However, I would recommend getting all your data from one session with one capture artist. This will make your life much easier in the long run.

Still, as an experienced production company, you won't be burdened with these details because your producers have budgeted the motion capture session correctly. Now you need to decide how you want this data to come to you. Other formats exist, but the Biovision (.BVA/.BVH) formats and the Acclaim Motion format are the big ones, and all the service bureaus and animation packages support these.

Your file format decision depends on your application and engine needs. You can bring these formats into a commercial animation package and export the data from there, but the formats are very compact and easy to use with your own tool set.

Definition of Terms

I'll refer to the character that you apply motion capture data to as a *skeleton*. The skeleton is made up of *bones*. To create the character's look, you attach geometry or weighted mesh vertices to these bones. The attributes that describe the position, orientation, and scale of a bone will be referred to as *channels*. By varying

---

LISTING 1. *Sample Biovision .BVA file.*

```
Segment:   Hips
Frames:    29
Frame Time: 0.033333
XTRAN      YTRAN      ZTRAN      XROT       YROT       ZROT       XSCALE  YSCALE  ZSCALE
INCHES     INCHES     INCHES     DEGREES    DEGREES    DEGREES    INCHES  INCHES  INCHES
0.000000   34.519684  0.000000   -14.988039 -12.240604 -3.481155  ...
0.102748   34.078739  3.159979   -15.337654 -14.320413 -3.983407  ...
0.260680   33.836613  6.487895   -16.308723 -15.090799 -3.861260  ...
...        REPEATS FOR A TOTAL OF 29 FRAMES
Segment:   Chest
Frames:    29
Frame Time: 0.033333
XTRAN      YTRAN      ZTRAN      XROT       YROT       ZROT       XSCALE  YSCALE  ZSCALE
INCHES     INCHES     INCHES     DEGREES    DEGREES    DEGREES    INCHES  INCHES  INCHES
0.272156   38.993561  -1.199981  -4.022753  -0.411088  1.354611   ...
0.413597   38.542671  1.932666   -4.371263  -0.591130  1.100887   ...
0.560568   38.279800  5.184929   -5.020082  -0.657020  0.768863   ...
...FOR THE REST OF THE SEGMENTS
```

the value in a channel over time, you get animation. These channels are combined into an *animation stream.* These streams can have a variable number of channels within them. Each slice of time is called a *frame.* In most applications, animation data has 30 frames per second, though that's not always the case.

BIOVISION's .BVA FORMAT. This is probably the easiest file format to handle. It's directly supported by most of the 3D animation packages. Let's take a look at a piece of a .BVA file (Listing 1).

This is as simple as animation data gets. For each bone in the skeleton (or what Biovision calls Segments), there are nine channels of animation. These represent the translation, rotation, and scale values for each bone for each frame. You'll also notice that there is no hierarchy definition. That's because each bone is described in its actual position (translation, rotation, and scale) for each frame. This can lead to problems, but it sure is easy to use.

Figure 1 shows the hierarchy of a sample .BVA file.

In Listing 1, we see that **Hips** as the first bone described. There are 29 frames of animation in the **Hips**. The frame time is described as 0.03333 seconds (per frame), which corresponds to 30 frames per second. Next comes a description of the channels and units used, then the actual channel data. There are 29 lines of nine values, followed by a segment block that describes the next bone, and so on, continuing to the end of the file. That's all there is to it.

BIOVISION's .BVH FORMAT. This format is similar to the .BVA format in many respects. In practice, I know of no off-the-shelf way to import this file format into Alias|Wavefront or Softimage, although Biovision's plug-in, Motion Manager for 3D Studio MAX, reads it.
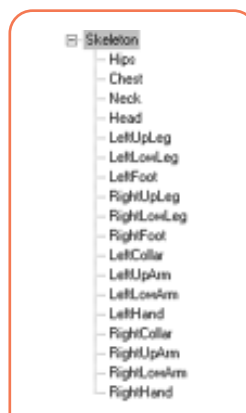

FIGURE 1. *.BVA file hierarchy.*

FIGURE 2. *.BVH file hierarchy.*

Still, it's an easy-to-read ASCII format that can be useful for importing and storing animation data. Obtaining data in this format should be easy because the format is supported by many motion capture devices and service bureaus.

The .BVH format differs from the .BVA format in several key areas, the most significant of which is that .BVH can store motion for a hierarchical skeleton. This means that the motion of the child bone is directly dependent on the motion of the parent bone. Figure 2 shows a sample .BVH format hierarchy.

In this sample, the bone **Hips** is the root of the skeleton. All other bones are children of the **Hips**. The rotation of the **LeftHip** is added to the rotation and translation of the **Hips**, and so on.

This hierarchy will certainly complicate the game engine's render loop. Why would you want to bother? You can do many more interesting things if your motion is in a hierarchy. Let's take the example of wanting to combine a "walk" motion with a "wave" motion. In the .BVA format, there is no relationship between the **LeftUpArm** and the **Hips**. If we were to apply a different motion to the different bones, nothing would stop them from separating. A motion hierarchy allows you to combine such motions fairly easily. Also, should we ever want to add inverse kinematics or dynamics to the game engine, a hierarchy would make this possible.

Listing 2 shows a fragment of a .BVH file. The word **HIERARCHY** in the first line signifies the start of the skeleton definition section. The first bone that is

---

LISTING 2. *Sample Biovision .BVH file.*

```
HIERARCHY
ROOT Hips
{
      OFFSET  0.00    0.00    0.00
      CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
      JOINT LeftHip
      {
            OFFSET    3.430000        0.000000    0.000000
            CHANNELS 3 Zrotation Xrotation Yrotation
            JOINT LeftKnee
            {
                  OFFSET  0.000000   -18.469999  0.000000
                  CHANNELS 3 Zrotation Xrotation Yrotation
                  JOINT LeftAnkle
                  {
                        OFFSET     0.000000   -17.950001  0.000000
                        CHANNELS 3 Zrotation Xrotation Yrotation
                        End Site
                        {
                                    OFFSET      0.000000   -3.119999   0.000000
                        }
                  }
            }
      }
...
}
MOTION
Frames:    20
Frame Time: 0.033333
  0.00  39.68   0.00    0.65 ...
...
```

defined is the `ROOT`. This bone is the parent to all other bones in the hierarchy. Each bone in this hierarchy is defined as a `JOINT`. Braces contain the root and each joint. All joints within a set of braces are the children of that parent joint.

Within each braced block is the `OFFSET` and `CHANNELS` definition for that bone (or `JOINT`). The `OFFSET` describes displacement of the root of the bone from its parent. This (x,y,z) coordinate is the world coordinate offset from the parent bone. In the example, the `Hips` bone is located at offset (0,0,0) and the `LeftHip` is 3.43 world units away from the `Hips` in the x axis.

The `CHANNELS` line defines which bone parameters will be animating in the file. The first parameter is the number of channels animated for this bone. Next is a data type for each of these channels. The possible types are: `Xposition`, `Yposition`, `Zposition`, `Xrotation`, `Yrotation`, and `Zrotation`. Note that the scale channels have been dropped in the .BVH format. Normally, only the root bone has any position data — the rest of the bones have only rotational data and rely on the root and the hierarchy for their position. The `CHANNELS` can be in any order. This order defines the sequence in which the operations need to be processed in the playback. For example, in the `LeftAnkle` joint, the order of channels is `Zrotation Xrotation Yrotation`, meaning that the bone is first rotated around the z axis, then the x axis, and finally the y axis. This becomes important when we try to display the data.
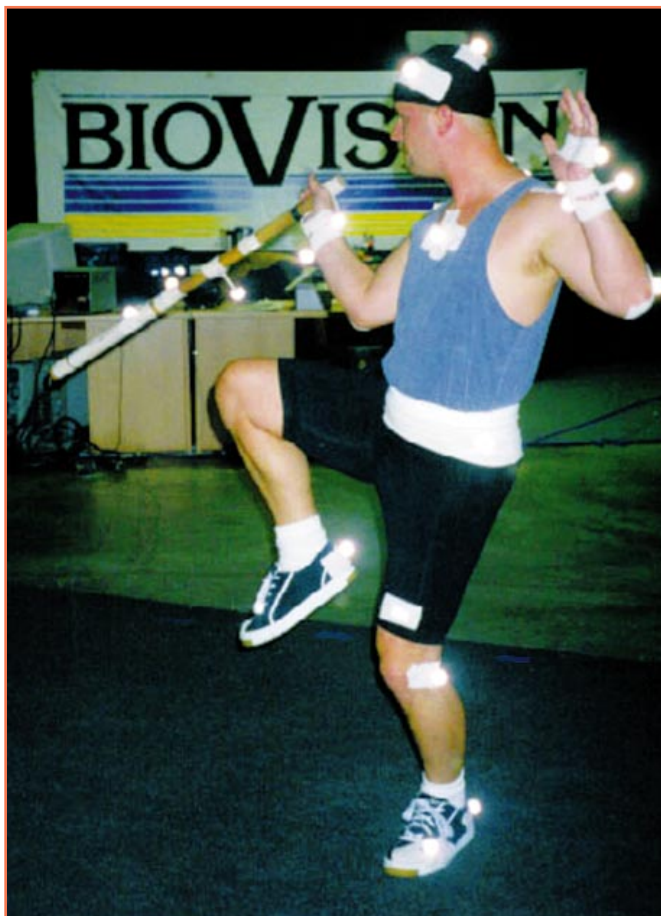
The branch of the hierarchy ends with the `End Site` joint. This joint is offset is only useful in determining the length of the last bone.

Following the `HIERARCHY` section is the `MOTION` section. This section actually describes the animation of each bone over time. As in the .BVA format, the first two lines of this section describe the number of frames and the time for each frame. However, unlike the .BVA format, the next lines describe the animation for all the bones at once. In each line in the rest of `MOTION` section, there is a value for every `CHANNEL` described in the `HIERARCHY` section. For example, if the `HIERARCHY` section describes 56 channels, there

will be 56 values on each line of the `MOTION` section. That continues for the total number of frames in the animation.

That's it for the .BVH format. While it's a bit more complex, it gives the programmer designing the engine greater flexibility.

ACCLAIM SKELETON FORMAT. This is the most complicated of the three file formats. It's also the most comprehensive, and supported by most of the 3D animation packages. An Acclaim motion capture file is actually made up of two files; the .ASF,

which describes the actual skeleton and its hierarchy, and the .AMC file, which contains the motion data. The separation of these two files has a nice benefit. In a single motion capture session, you can have one .ASF file that describes the skeleton and multiple .AMC motion files. The Acclaim format is such a technical and complex file format that this overview may not provide all the needed information. Documents describing the format in greater detail are available on the *Game Developer* web site (http://www.gdmag.com).

The .ASF file is similar to the `HIERARCHY` section of the .BVH file in many ways. Both files describe the joints and the hierarchy, but the .ASF file extends this a bit. Listing 3 displays a portion of an Acclaim .ASF file.

In this file format, lines beginning with a pound sign (`#`) are ignored. The .ASF file is divided into sections. Each section starts with a keyword preceded by a colon. The section continues until another keyword is reached. The `:version`, `:name`, and `:documentation` section are self-explanatory. The `:units` section describes a definition for all values and units of measure used.

The `:root` section describes the parent of the hierarchy. The `axis` and `order` elements describe the order of operations for the initial offset and root node transformation. The `position` element describes the root translation of the skeleton and the `orientation` element defines the rotation.

The `:bonedata` keyword starts a block that describes all of the remaining bones in the hierarchy. Each bone is delimited by `begin` and `end` statements.

FIGURE 3. .ASF file hierarchy.

35

```
:version 1.10
:name BioSkeleton
:units
   mass 1.0
   length 1.0
   angle deg
:documentation
   Data translated and provided by
   BioVision Motion Capture Studios
:root
   axis XYZ
   order TX TY TZ RZ RY RX
   position 0.0 0.0 0.0
   orientation 0.0 0.0 0.0
:bonedata
begin
   id 1
   name hips
   direction 0.000000 1.000000 0.000000
   length 0.000000
   axis 0.00000 0.00000 0.00000 XYZ
   dof rx ry rz
   limits (-180.0 180.0)
      (-180.0 180.0)
      (-180.0 180.0)
end
begin
   id 2
   name hips1
...
end
:hierarchy
begin
   root body_root1
   body_root1 hips
   hips hips1 hips2 hips3
...
end
```

```
:FULLY-SPECIFIED
:DEGREES
1
root -1.244205 36.710186 7.591899 0.958161 4.190043 -18.282991
hips 0.000000 0.000000 0.000000
chest 15.511776 -2.804996 -0.725314
neck 48.559605 0.000000 0.014236
head -38.332661 1.462782 -1.753684
leftcollar 0.000000 15.958783 0.921166
leftuparm -10.319685 -15.040003 63.091194
leftlowarm -27.769176 -15.856658 8.187016
lefthand 2.601753 -0.217064 -5.543770
rightcollar 0.000000 -8.470076 2.895008
rightuparm 6.496142 9.551583 -57.854118
rightlowarm -26.983490 11.338276 -5.716377
righthand -6.387745 -1.258509 5.876069
leftupleg 23.412262 -5.325913 12.099395
leftlowleg -6.933442 -6.276054 -1.363996
leftfoot -1.877641 4.455667 -6.275022
rightupleg 20.698696 3.189690 -8.377244
rightlowleg 3.445840 -6.717122 2.046032
rightfoot -8.162314 0.687809 9.000264
2
root -4.232432 36.723934 9.596100 -7.051147 1.678117 -7.711937
hips 0.000000 0.000000 0.000000
chest 31.863499 -19.017111 6.490547
...
```

This bone description section is what makes the Acclaim format very useful.

The **id** and **name** elements describe the bone by number or string. The initial rest position of the bone is described by the **direction** vector, and the **length** describes the physical length of the bone. The **axis** parameter describes the global orientation via an axis vector, and the token letters **xyz** describe the order of rotations. Not included in the sample are two optional elements: **bodymass**, which defines the mass of the bone, and **cofmass** which pinpoints the center of mass via a distance along the bone.

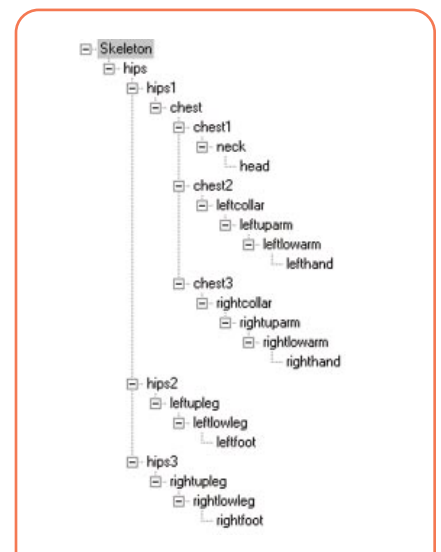The **dof** element describes the degrees of freedom possible in the bone. This is a list of tokens. The possible values are tx, ty, tz, rx, ry, rz, and l. The first of these six define freedom to translate and rotate around the three axes. The last **dof** defines the bone's ability to stretch in length over time. Each of these tokens represents a channel that will be present in the .AMC file in that order. The order of these channel tokens also describes the order of operations in the transformation of the bone.

The **limits** element is very interesting. It describes the limits of the degrees of freedom. It consists of value pairs of either floats or the keyword **inf**, meaning infinite. This information can be useful for setting up an inverse kinematic or dynamic 3D character.

The next section in the .ASF file is **:hierarchy**. Just as it sounds, it describes the hierarchy of the bones declared in the **:bonedata** section. It's a **begin...end** block in which each line is the parent bone followed by its children. From this information, the bones should be connected together in the proper hierarchy. Figure 3 displays the hierarchy in the sample .ASF file.

The .AMC file defines the actual channel animation. Listing 4 contains a sample .AMC fragment. Each frame of animation starts with a line declaring the frame number. Next is the bone animation data, which is comprised of the bone name and data for each channel defined for that bone. This information was defined in the **dof** section of each bone in the .ASF file. The frame sections in the file continue until the end of the file. After the complexity of the .ASF file, the .AMC looks pretty simple.
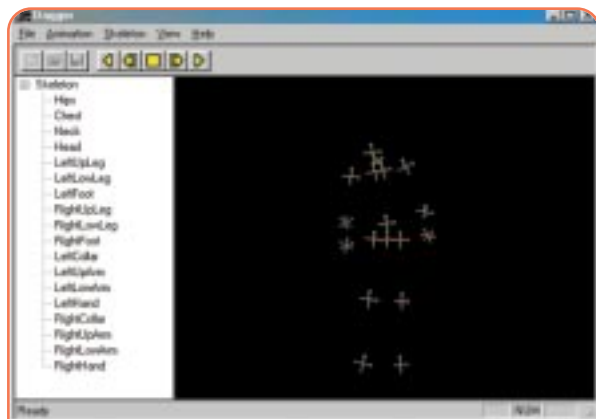

FIGURE 3. .ASF file hierarchy.

FIGURE 4. *A sample animation in OGLView.*

Pipeline," Casey Muratori, *Game Developer*, February/March 1997, pp.38-41). The order in which you perform these operations is critical to getting the expected result.

Because I wanted my motion capture viewer to support several different file formats, it was important to take operation order into account. I created a series of

You should be aware of one important aspect of the Acclaim and .BVH formats. While both formats can store rotations in arbitrary order, both Softimage and Alias|Wavefront expect the order of rotations to be tx, ty, tz, rx, ry, rz. This is important if you plan on going back and forth between the game engine and one of these packages.

Working with Data

Once you have your data in a format that you're happy with, it's time to start working on it. I've created an application that loads motion capture files of different formats and allows the user to play them back. You can download it from the *Game Developer* web site. When full production kicks in, such tools are very useful for file conversion and formatting. Also, it serves as a good test application to try out new ideas and benchmark code.

I decided to create the motion capture viewer as a MFC OpenGL application — I find it very quick and easy to create tools this way. If you're careful about how you design the tool, much of the code can be used directly in the game engine itself. Figure 4 shows a sample animation that has been loaded into the application.

DATA REPRESENTATION. As we saw from the different file formats, there are several ways to store the animation data from a motion capture session. The most important data is the order of rotations in each stream. You may remember from 3D matrix math that matrix multiplication is noncommutative (see "Inspecting the 3D

stream IDs that describe the order of channels in each stream.

Listing 5 shows the stream types that I've needed. These are not all the possibilities, but they are the ones that I've found useful. By creating separate stream types for single operations such as `STREAM_TYPE_TRANS` and `STREAM_TYPE_RXYZ`, I decrease stream size while animating. This operation isn't as important when the animation can fit in RAM, but it becomes critical when you have to stream animation off of a CD-ROM or the Internet.

Next, I created a data structure to represent a bone (Listing 6). I chose to store the transformation information as separate scale, translation, and rotation vectors instead of a global transformation matrix. This made it much easier to handle the different channel types. I also find the rotation values, called Euler angles, easy to understand while debugging. Conversion to and from quaternions for animation or a transformation matrix can also be done easily. Once the data format for a game engine is set, this can be optimized.

Looking at the `primStreamType`, `primStream`, and `primFrameCount` fields, it seems curious that I would want to have a motion stream for each bone. It would certainly be easier to have one animation stream that contains all the data for all the bones in the skeleton. However, this method allows the flexibility to have different stream types per bone. This can be useful because it allows me to attach completely different motions to the individual bones. Imagine a character in a walk cycle. The legs and hips are

LISTING 5. STREAM *definitions from* `Skeleton.H.`

```
/// STREAM Definitions //////////////////////////////////////////////////////
#define STREAM_TYPE_NONE            0           // NO STREAM APPLIED
#define STREAM_TYPE_SRT             1           // SCALE ROTATION AND TRANSLATION
#define STREAM_TYPE_TRANS           2           // STREAM HAS TRANSLATION (X Y Z) ORDER
#define STREAM_TYPE_RXYZ            4           // ROTATION (RX RY RZ) ORDER
#define STREAM_TYPE_RZXY            8           // ROTATION (RZ RX RY) ORDER
#define STREAM_TYPE_RYZX            16          // ROTATION (RY RZ RX) ORDER
#define STREAM_TYPE_RZYX            32          // ROTATION (RZ RY RX) ORDER
#define STREAM_TYPE_RXZY            64          // ROTATION (RX RZ RY) ORDER
#define STREAM_TYPE_RYXZ            128         // ROTATION (RY RX RZ) ORDER
#define STREAM_TYPE_S               256         // SCALE ONLY
#define STREAM_TYPE_T               512         // TRANSLATION ONLY (X Y Z) ORDER
#define STREAM_TYPE_INTERLEAVED     1024        // THIS DATA STREAM HAS MULTIPLE STREAMS
//////////////////////////////////////////////////////////////////////////////
```

affected by the **walk** stream. Suppose I then attach a **wave** stream to the right arm. Now I have a walking and waving character. We can also begin to plan for the possibility of blending animations together to create dynamic motions on the fly.

DISPLAY METHODS. Now that I have all this data loaded, I have to display it in a way that provides the most information possible. In my tool, I chose to represent each bone of the skeleton as an axis. The axes are colored red for x, green for y, and blue for z. An arrow indicates the positive direction in each axis. Since I was using OpenGL to create my motion capture tool, this seemed like a good opportunity to use display lists. Display lists are a method that OpenGL uses to optimize sequences of commands. Listing 7 contains the OpenGL commands that I used to create a simple colored axis.

I also created a hierarchy browser using the **CTreeCtl** class in MFC. This gives a nice visual representation of how the skeleton is laid out. From there, it's easy to add dialog boxes to edit bone settings, a more proper animation control window, and so on.

The animation is all handled via a Windows timer event. This isn't the fastest way to animate a Windows application, but it's plenty fast for this demonstration. There's a very good discussion on animating OpenGL Windows applications in Ron Fosner's book, *OpenGL Programming for Windows 95 and Windows NT*. I recommend this book and the *OpenGL Super Bible* by Wright and Sweet to any Windows OpenGL programmer.

The source code and executable for this application, along with sample motion files and two documents describing the Acclaim file format can be found on the *Game Developer* web site. ∎

Acknowledgements

LISTING 6. *Structure definition from* `Skeleton.h.`

```
struct t_Bone
{
    long        id;                 // BONE ID
    char        name[80];           // BONE NAME
    // HIERARCHY INFO
    t_Bone      *parent;            // POINTER TO PARENT BONE
    int         childCnt;           // COUNT OF CHILD BONES
    t_Bone      *children;          // POINTER TO CHILDREN
    // TRANSFORMATION INFO
    tVector     b_scale;            // BASE SCALE FACTORS
    tVector     b_rot;              // BASE ROTATION FACTORS
    tVector     b_trans;            // BASE TRANSLATION FACTORS
    tVector     scale;              // CURRENT SCALE FACTORS
    tVector     rot;                // CURRENT ROTATION FACTORS
    tVector     trans;              // CURRENT TRANSLATION FACTORS
    // ANIMATION INFO
    DWORD       primStreamType;     // WHAT TYPE OF PRIMARY STREAM IS ATTACHED
    float       *primStream;        // POINTER TO PRIMARY STREAM OF ANIMATION
    float       primFrameCount;     // FRAMES IN PRIMARY STREAM
    float       primCurFrame;       // CURRENT FRAME NUMBER IN STREAM
    ...
    // REST OF STRUCTURE DECLARATION
};
```

LISTING 7. *Display list code from OGLView.CPP*

```
// CREATE THE DISPLAY LIST FOR AN AXIS WITH ARROWS POINTING IN
// THE POSITIVE DIRECTION Red = X, Green = Y, Blue = Z
glNewList(OGL_AXIS_DLIST,GL_COMPILE);
    glBegin(GL_LINES);
            glColor3f(1.0f, 0.0f, 0.0f);        // X AXIS STARTS - COLOR RED
            glVertex3f(-0.2f,  0.0f, 0.0f);
            glVertex3f( 0.2f,  0.0f, 0.0f);
            glVertex3f( 0.2f,  0.0f, 0.0f);     // TOP PIECE OF ARROWHEAD
            glVertex3f( 0.15f, 0.04f, 0.0f);
            glVertex3f( 0.2f,  0.0f, 0.0f);     // BOTTOM PIECE OF ARROWHEAD
            glVertex3f( 0.15f, -0.04f, 0.0f);
            glColor3f(0.0f, 1.0f, 0.0f);        // Y AXIS STARTS - COLOR GREEN
            glVertex3f( 0.0f,  0.2f, 0.0f);
            glVertex3f( 0.0f, -0.2f, 0.0f);
            glVertex3f( 0.0f,  0.2f, 0.0f);     // TOP PIECE OF ARROWHEAD
            glVertex3f( 0.04f,  0.15f, 0.0f);
            glVertex3f( 0.0f,  0.2f, 0.0f);     // BOTTOM PIECE OF ARROWHEAD
            glVertex3f( -0.04f,  0.15f, 0.0f);
            glColor3f(0.0f, 0.0f, 1.0f);        // Z AXIS STARTS - COLOR BLUE
            glVertex3f( 0.0f,  0.0f,  0.2f);
            glVertex3f( 0.0f,  0.0f, -0.2f);
            glVertex3f( 0.0f,  0.0f, 0.2f);     // TOP PIECE OF ARROWHEAD
            glVertex3f( 0.0f,  0.04f, 0.15f);
            glVertex3f( 0.0f, 0.0f, 0.2f);      // BOTTOM PIECE OF ARROWHEAD
            glVertex3f( 0.0f, -0.04f, 0.15f);
        glEnd();
    glEndList();
```