Playing Movies in a Java 3D World (Part 1)

Andrew Davison

Dept. of Computer Engineering
Prince of Songkla University
Hat Yai, Songkhla 90112
Thailand

ad@fivedots.coe.psu.ac.th

May 2005

The ability to play a movie clip inside a Java 3D scene opens up opportunities for richer, more interesting 3D content. A movie can display more believable backgrounds, such as moving clouds, a busy city street, or the view out of a window. Movies can be employed in help screens, or as transitions between game levels.

This article, which is split into two parts, describes how I implemented a Java 3D movie screen. In this part, I'll explain how I utilized the Java Media Framework (JMF), more specifically the JMF Performance Pack for Windows v.2.1.1e (`http://java.sun.com/products/java-media/jmf/`). The other tools in my arsenal were J2SE 5.0 and Java 3D 1.3.2. In part two, I'll discuss another version of the movie screen, using Quicktime for Java.

Figure 1 shows two screenshots of the JMF Movie3D application, taken at different times: the one on the right is a view of the screen from the back.

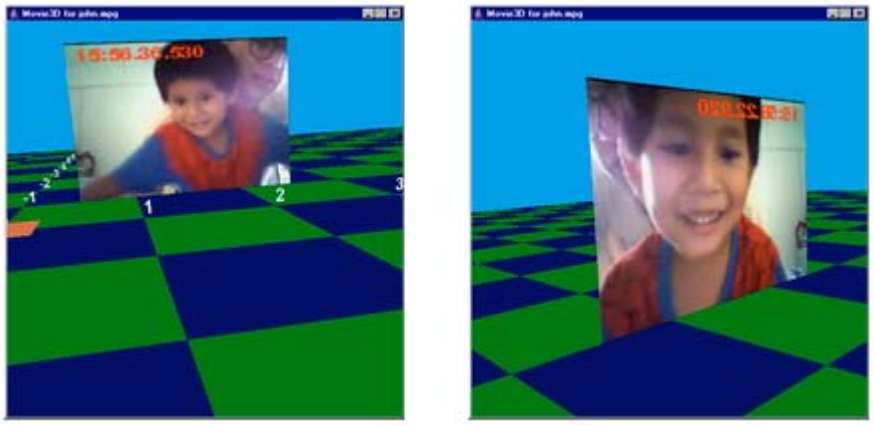

Figure 1. Two Views of the Movie3D Application

The important elements of this application are:

- An integration of JMF and Java 3D. There can be multiple screens in an application, of any size. Since a screen is a subclass of Java 3D's Shape3D class, it can be easily integrated into different Java 3D scenes.
- The implementation uses the *Model-View-Controller* design pattern. The screen is the view element, represented by the JMFMovieScreen class. The movie is the model part, and is managed by the JMFSnapper class. A Java 3D Behavior class, TimeBehavior, is the controller, triggering periodic updates of the movie. All the JMF code is localized in the JMFSnapper class, making it easier to test and

changes. Part two of this article essentially replaces JMFSnapper by a Quicktime for Java version called QTSnapper.

- The use of Java 3D performance tricks to speed up rendering. The result is a movie which runs at 25 frames/second without any difficulty.
- A discussion of the problems I had with JMF, problems which meant that my preferred solution wouldn't work. JMF has the potential to be a great API, but beneath its gleaming surface there are some poorly implemented features lying in wait

1. I'm Sitting on a Mountain

Actually, no, I'm sitting on a chair in a very cold office with a thermostat that's out of reach. What I really mean is that this article rests on top of a lot of background knowledge about Java 3D and JMF.

I'm not going to explain the Java 3D elements in much detail since they're covered in my O'Reilly book, *Killer Game Programming in Java* (henceforward known as KGPJ). For example, the checkerboard scene shown in Figure 1 is a slightly modified version of the Checkers3D example in Chapter 15. I've reused the code for creating the checkerboard floor, the blue sky, the lighting, and for allowing the user to move the viewpoint around the scene.

If you don't want to buy the book, then early drafts of all the chapters, and all the code, can be found at the book's website:

```
http://fivedots.coe.psu.ac.th/~ad/jg/
```

In this article, I'll explain the JMF techniques I've used for extracting frames from the movie. I won't be talking about streaming media, capture, or transcoding.

2. Two Overviews of the Application

The movie is loaded and played by the JMFSnapper class, and plays in a continuous loop until told to stop.

The movie screen is created by JMFMovieScreen, which manages a Java 3D quadrilateral (a *quad*) resting on the checkerboard floor.

One way of visualizing these classes is to look at the application's scene graph in Figure 2. (A scene graph shows how the Java 3D nodes in a scene are linked together.)

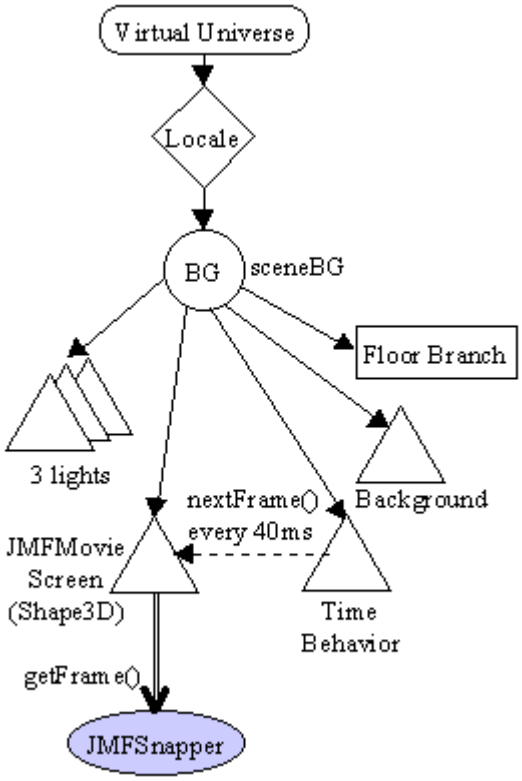

Figure 2. Scene Graph for Movie3D

A lot of the detail in Figure 2 can be ignored, but the graph bears a striking resemblance to the one for the Checkers3D example in Chapter 15 of KGPJ. Only the movie-specific nodes are new.

The JMFMovieScreen and TimeBehavior objects are shown as triangles since they're nodes in the scene graph. The JMFSnapper object isn't part of the graph, but is called by JMFMovieScreen.

Every 40 ms, the TimeBehavior object calls the nextFrame() method in JMFMovieScreen. That in turn calls getFrame() in JMFSnapper to get the current frame in the playing movie, which is then laid over the quad managed by JMFMovieScreen.

TimeBehavior is a subclass of Java 3D's Behavior class, and is the Java 3D way of implementing a timer. It's very similar to the TimeBehavior class used in the 3D sprites example of Chapter 18 of KGPJ.

Another way of gaining some insight about the application is to look at its UML class diagrams, given in Figure 3. Only the public methods of the classes are shown.

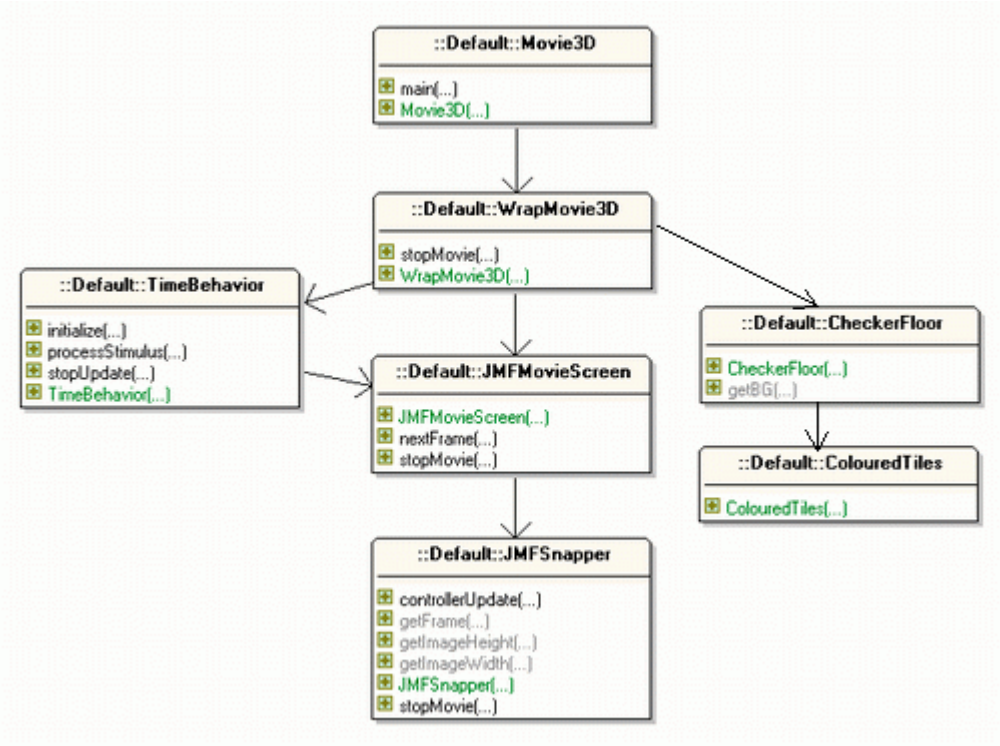

Figure 3. Class Diagrams for Movie3D

Movie3D subclasses JFrame, while WrapMovie3D is a subclass of JPanel. WrapMovie3D constructs the scene graph shown in Figure 2, and renders it into the application's JPanel. It uses the CheckerFloor and ColouredTiles classes to build the checkerboard floor.

JMFMovieScreen creates the movie screen, adds it to the scene, and starts the movie by creating a JMFSnapper object. TimeBehavior calls JMFMovieScreen's nextFrame () method every 40ms. nextFrame() calls getFrame() in JMFSnapper to retrieve the current frame.

All the code for this example, as well as an early version of this article, can be found at the KGPJ website, `http://fivedots.coe.psu.ac.th/~ad/jg/`.

3. Going to the Movies

The movie, it's screen, and the TimeBehavior object for updating the screen, are set up by the addMovieScreen () method in WrapMovie3D:

```
// globals
private BranchGroup sceneBG;
private JMFMovieScreen ms;    // the movie screen
private TimeBehavior timer;   // for updating the screen
```

```
private void addMovieScreen(String fnm)
{
  // put the movie in fnm onto a movie screen
  ms = new JMFMovieScreen( new Point3f(1.5f, 0, -1), 2.0f, fnm);
  sceneBG.addChild(ms);

  // set up the timer for animating the movie
  timer = new TimeBehavior(40, ms);
           // update the movie every 40ms (== 25 frames/sec)
  timer.setSchedulingBounds(bounds);
  sceneBG.addChild(timer);
}
```

The two Java 3D addChild() calls link the JMFMovieScreen and TimeBehavior nodes into the scene graph. The setSchedulingBounds() call activates the TimeBehavior node (i.e. starts it ticking).

4. Creating the Movie Screen

JMFMovieScreen is a subclass of Java 3D's Shape3D class, so must specify a geometry and appearance for its shape.

The geometry is a quadrilateral (quad) with sides proportional to the movie's image size, but with a maximum dimension (width or height) specified as an argument to the constructor. The quad is upright, facing along the +z axis, and can be positioned anywhere on the floor.

The quad's appearance is two-sided, allowing the movie to be seen on the screen's front and back. The texture is smoothed using bilinear interpolation, which greatly reduces the pixellation of the movie image when viewed up close.

Most of this functionality is copied from the ImageCsSeries class used in the First Person Shooter (FPS) example in Chapter 24 of KGPJ. ImageCsSeries displays a series of GIF images on a quad. For the sake of brevity, I'll only describe the features of JMFMovieScreen that differ from ImageCsSeries.

Rendering the Image Efficiently

A frame from the movie is laid over the quad by being converted to a texture; this is done in two steps: first the supplied BufferedImage is passed to a Java 3D ImageComponent2D object, and then to a Java 3D Texture2D.

The updating of the quad's texture occurs quite rapidly: there are 25 frame updates per second, requiring 25 changes to the texture. It's therefore quite important that the texturing be carried out efficiently. This is possible by ensuring that certain formats are utilized for the BufferedImage and ImageComponent2D objects.

The ImageComponent2D object used by JMFMovieScreen is declared like so:

```
ImageComponent2D ic = new ImageComponent2D(
           ImageComponent2D.FORMAT_RGB,
```

```
            FORMAT_SIZE, FORMAT_SIZE, true, true);
```

The last two arguments of the constructor specify that it uses the "by reference" and "Y-up" modes. These modes reduce the memory needed to store the texture image, since Java 3D will avoid copying the image from application space into graphics memory.

In a Windows OS environment, using OpenGL as the underlying rendering engine in Java 3D, the ImageComponent2D format should be `ImageComponent2D.FORMAT_RGB` (as shown above), and the BufferedImage format should be `BufferedImage.TYPE_3BYTE_BGR`. The BufferedImage format is fixed in JMFSnapper.

More details on this technique, and other performance tips, can be found at j3d.org, `http://www.j3d.org/tutorials/quick_fix/perf_guide_1_3.html`.

Linking a Texture to the Quad

The usual way of tying a texture (image) to a quad is to link the lower left corner of the texture to the lower left corner of the quad, and specify the other connections in a counter-clockwise direction. This approach is illustrated by Figure 4.

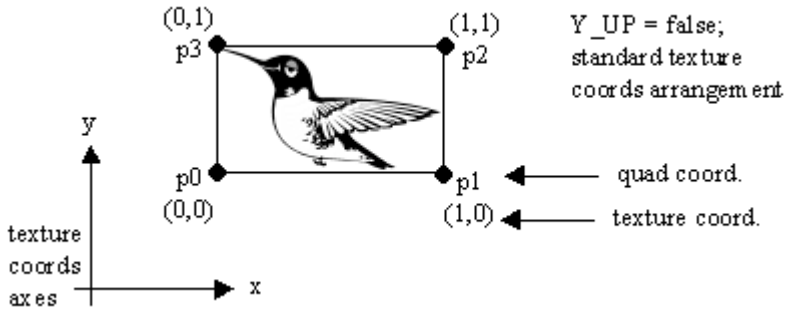

Figure 4. The Standard Linkage between Texture and Quad

The texture coordinates range between 0 and 1 along the x- and y- axes, with the y-axis pointing upwards. For example, the lower left corner of the texture uses the coordinate (0,0), and the top-right corner is at (1,1).

When the “Y-up” mode is employed, the y-axis of the texture coordinates is reversed, to point downwards. This means that the texture coordinate (0,0) refers to the *top* left of the texture, while (1,1) refers to the *bottom* right.

With the "Y-up" mode set, the texture coordinates must be assigned to different points on the quad in order to obtain the same orientation for the image. This new configuration is shown in Figure 5.

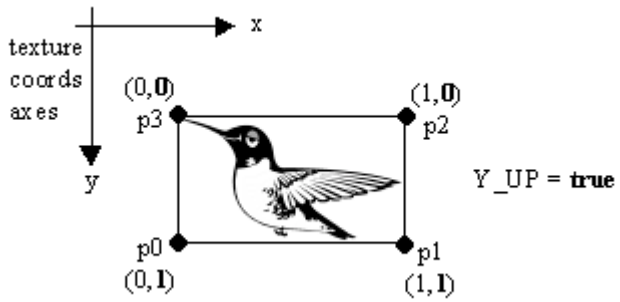

Figure 5. The Linkage between Texture and Quad when "Y-up" Mode is Used

The JMFMovieScreen code which connects the quad points and the texture coordinates is:

```
TexCoord2f q = new TexCoord2f();
q.set(0.0f, 0.0f);
plane.setTextureCoordinate(0, 3, q);
                  // (0,0) tex coord --> top left quad point (03)
q.set(1.0f, 0.0f);
plane.setTextureCoordinate(0, 2, q);  // (1,0) --> top right (p2)
q.set(1.0f, 1.0f);
plane.setTextureCoordinate(0, 1, q);  // (1,1) --> bottom right (p1)
q.set(0.0f, 1.0f);
plane.setTextureCoordinate(0, 0, q);  // (0,1) --> bottom left (p0)
```

The plane object represents the quad.

Updating the Image

As explained earlier, a TimeBehavior object is set to call JMFMovieScreen's nextFrame() method every 40 ms. nextFrame() calls getFrame() in the JMFSnapper object to retrieve the current movie frame as a BufferedImage object. This is assigned to an ImageComponent2D object, and then to the quad's texture. nextFrame() is:

```
// globals
private Texture2D texture;     // used by the quad
private ImageComponent2D ic;

private JMFSnapper snapper;    // to take snaps of the movie
private boolean isStopped = false;   // is the movie stopped?


public void nextFrame()
{ if (isStopped)    // the movie has been stopped
    return;

  BufferedImage im = snapper.getFrame();  // get current frame
  if (im != null) {
    ic.set(im);              // assign frame to ImageComponent2D
```

```
    texture.setImage(0,ic);  // make it the shape's texture
  }
  else
    System.out.println("Null BufferedImage");
}
```

snapper, the JMFSnapper object, is created in JMFMovieScreen's constructor:

```
// load and play the movie
snapper = new JMFSnapper(movieFnm);
```

JMFSnapper's simple interface hides the complexity of the JMF code required to play the movie and extract frames from it. In part two of this article, JMFSnapper is replaced by a version using Quicktime for Java, with minimal changes required to JMFMovieScreen.

5. Managing the Movie

JMF offers a high-level way of accessing specific movie frames. The code fragment below illustrates the main elements. I've left out error checking and exception handling.

```
// create a movie player, in a 'realized' state
URL url = new URL("file:" + movieFnm);
Player p = Manager.createRealizedPlayer(url);

// create a frame positioner
FramePositioningControl fpc = (FramePositioningControl)
    p.getControl("javax.media.control.FramePositioningControl");

// create a frame grabber
FrameGrabbingControl fg = (FrameGrabbingControl)
    p.getControl("javax.media.control.FrameGrabbingControl");

// request that the player changes to a 'prefetched' state
p.prefetch();

// wait until the player is in that state...

// move to a particular frame, e.g. frame 100
fpc.seek(100);

 // take a snap of the current frame
Buffer buf = fg.grabFrame();

// get its video format details
VideoFormat vf = (VideoFormat) buf.getFormat();

// initialize BufferToImage with the video format
BufferToImage bufferToImage = new BufferToImage(vf);

// convert the buffer to an image
Image im = bufferToImage.createImage(buf);
```

```
// specify the format of the desired BufferedImage object
BufferedImage formatImg =
      new BufferedImage(FORMAT_SIZE, FORMAT_SIZE,
                              BufferedImage.TYPE_3BYTE_BGR);

// convert the image to a BufferedImage
Graphics g = formatImg.getGraphics();
g.drawImage(im, 0, 0, FORMAT_SIZE, FORMAT_SIZE, null);
g.dispose();
```

A media player passes through six states between being created and started. A player in the *realized* state knows how to render its data, so can provide visual components and controls when asked. I require two controls: FramePositioningControl and FrameGrabbingControl. FramePositioningControl offers methods like seek() and skip () for moving about inside a movie to examine a particular frame. FrameGrabbingControl supplies grabFrame(), which pulls the current frame from the video track of the movie.

For these controls to work, the player must be moved from its *realized* state into a *prefetched* state. This prepares the player for playing the media, and the media data is loaded.

The call to prefetch() is asynchronous, which means that my code must include a waiting period until the state transition is finished. The standard JMF coding solution is to implement a waitForState() method which causes execution to pause until a state change event wakes it up.

The desired frame can be located in the track with seek(), then grabbed with grabFrame(). The code must go through several translation steps to convert the grabbed Buffer object into the BufferedImage object required by JMFMovieScreen. Note that the BufferedImage object uses the `TYPE_3BYTE_BGR` format, which is necessary for the Java 3D parts of the program to employ texturing by reference.

Sun's JMF website contains a useful collection of small examples (`http://java.sun.com/products/java-media/jmf/2.1.1/solutions/`), one of which, Seek.java, shows how to use FramePositioningControl to step through a movie.

Hacking in Three Steps

Unfortunately, the code outlined above fails, at least in the JMF Performance Pack for Windows v.2.1.1e. I went through several rewrites to get to a working version of JMFSnapper.

Hack 1. The two controls, FramePositioningControl and FrameGrabbingControl, are unavailable in the default player module used in JMF. (The Solaris and Win32 performance packs each support two different MPEG players.) The 'native modular' player is required, which is selected by calling:

```
Manager.setHint(Manager.PLUGIN_PLAYER, new Boolean(true));
```

This player is a heavy-weight component, which interacts poorly with light-weight Swing GUIs such as JFrame and JPanel. However, I don't need to display the player. A more serious consequence of using the native modular player is a much longer

loading time for the media, and erratic playing (e.g. varying play rates and dropped frames).

Hack 2. After pondering for a while, I decided the best way to speed up the player was to give it less work to do. I stripped the audio tracks out of the MPEG files, and made sure the files were saved in the (relatively) simple MPEG-1 format. Any number of video editing tools are available to do these tasks. I used two freeware utilities: MPEG Properties (`http://www.medialab.se/mpgprop.html`) and FlasKMPEG (`http://www.flaskmpeg.net/`). The former is a simple utility that supplies movie format information, while the latter is a decent editor.

The stripped down movies play promptly, their frame rates are constant, and no frames are lost.

Nevertheless, the FramePositioningControl class is unreliable. On my WinXP machine, seek() almost always failed, and skip() worked correctly perhaps 4 times out of 5.

Hack 3. I bid a tearful farewell to FramePositioningControl. My frame grabbing algorithm relies on calling FrameGrabbingControl's grabFrame() method at regular intervals while the player is running the movie.

I now have code which reliably catches frames from video-only MPEG-1 files. It also works fairly well with files that have video and audio tracks, but the player is slow to start. Also, the erratic playing causes frames to be grabbed erratically.

I added some 'waiting' code at the start of JMFSnapper to deal with video+audio movies. The JMFSnapper object waits for a player to start (i.e. to enter its *started* state), *and* also waits for the first movie frame to become available.

Waiting for the First Frame

The JMFSnapper constructor calls a waitForBufferToImage() method which repeatedly calls hasBufferToImage() until it detects the first video frame.

hasBufferToImage() calls FrameGrabbingControl's grabFrame(), and checks if the returned Buffer object contains video format data. It uses this data to initialize a BufferToImage object, which is employed subsequently to translate each grabbed frame into an image.

```
// globals
private FrameGrabbingControl fg;    // the frame grabber
private BufferToImage bufferToImage = null;
private int width, height;          // frame dimensions


private boolean hasBufferToImage()
{
  Buffer buf = fg.grabFrame();     // take a snap
  if (buf == null) {
    System.out.println("No grabbed frame");
    return false;
  }

  // there is a buffer, but check if it's empty or not
```

```
  VideoFormat vf = (VideoFormat) buf.getFormat();
  if (vf == null) {
    System.out.println("No video format");
    return false;
  }

  System.out.println("Video format: " + vf);
  width = vf.getSize().width;     // extract the image's dimensions
  height = vf.getSize().height;

  // initialize bufferToImage with the video format info.
  bufferToImage = new BufferToImage(vf);
  return true;
}
```

A minor drawback of this coding approach is that the first video frame (which causes hasBufferToImage() to return true) is discarded after the BufferToImage object is initialized. The frame isn't made available as a BufferedImage to JMFMovieScreen.

Taking a Snap

The most important public method of JMFSnapper is getFrame(), which is called periodically to get the current frame in the running movie.

```
// global
private BufferedImage formatImg;    // for the frame image

synchronized public BufferedImage getFrame()
{
  // grab the current frame as a buffer object
  Buffer buf = fg.grabFrame();
  if (buf == null) {
    System.out.println("No grabbed buffer");
    return null;
  }

  // convert buffer to image
  Image im = bufferToImage.createImage(buf);
  if (im == null) {
    System.out.println("No grabbed image");
    return null;
  }

  // convert the image to a BufferedImage
  Graphics g = formatImg.getGraphics();
  g.drawImage(im, 0, 0, FORMAT_SIZE, FORMAT_SIZE, null);

  // Overlay current time on top of the image
  g.setColor(Color.RED);
  g.setFont(new Font("Helvetica", Font.BOLD, 12));
  g.drawString(timeNow(), 5, 14);

  g.dispose();

  return formatImg;
}  // end of getFrame()
```

The methods getFrame() and closeMovie() are both synchronized in JMFSnapper. closeMovie() terminates the player, and may be called at any time. The synchronized keywords ensure that the player can't be closed while a frame is being extracted from it.

The formatImg BufferedImage object is initialized in JMFSnapper's constructor:

```
formatImg = new BufferedImage(FORMAT_SIZE, FORMAT_SIZE,
                                  BufferedImage.TYPE_3BYTE_BGR);
```

6. Other Approaches to Frame Grabbing

Sun's JMF examples website (`http://java.sun.com/products/java-media/jmf/2.1.1/solutions/`) offers two other ways of grabbing frames from a movie.

The VideoRenderer

The DemoJMFJ3D example (`http://java.sun.com/products/java-media/jmf/2.1.1/solutions/DemoJMFJ3D.html`) is a combined Java 3D and JMF application, which shows how to wrap a video around a cylinder.

The Java 3D part is virtually identical to what I've discussed – a BufferedImage using the `BufferedImage.TYPE_3BYTE_BGR` format is passed to an ImageComponent2D object, and then becomes the cylinder's texture. The image can also use the `BufferedImage.TYPE_4BYTE_ABGR` format, which is required by Solaris in order to support texturing by reference.

The JMF side of the program is quite different from mine. An implementation of JMF's VideoRenderer interface is attached to the TrackControl object for the video track of the movie. Once the TrackControl object is started, the process() method of VideoRenderer is automatically called for each frame encountered in the video. process()'s input argument is the Buffer object (i.e. the grabbed frame). Rather than use the Buffer-to-BufferedImage translation steps I've outlined, DemoJMFJ3D builds the BufferedImage by carrying out a low-level byte array copy between the Buffer's raw data and a pixel map for the BufferedImage.

A lot of the code in DemoJMFJ3D is used in a 3D chat room example in:

> *Java Media APIs: Cross-Platform Imaging, Media and Visualization*
> A. Terrazas, J. Ostuni, and M. Barlow
> Sams, 2002
> `http://www.samspublishing.com/title/0672320940`

I recommend this book as a good introduction to JMF, and it also has several very interesting chapters on Java 3D.

A Processor Codec Plug-in

The FrameAccess example (`http://java.sun.com/products/java-media/jmf/2.1.1/solutions/FrameAccess.html`) utilizes more advanced elements of JMF, centered around a Processor codec plug-in.

The Processor class is an extended version of Player, which offers more capabilities for processing media data. A codec plug-in (an implementation of the JMF interface Codec) is capable of reading frames from a track, processing them in arbitrary ways, then writing them back to the track. In particular, Codec's process() method is called each time a frame in encountered in the track. It's supplied with a Buffer object holding the input frame, and an empty Buffer object for the output

FrameAccess attaches a Codec plug-in to the movie's video track, and uses the input frame Buffer object passed to process() to generate some basic statistics about the video. This example could easily be modified to convert the Buffer object into a BufferedImage, either using my approach or the byte array technique of DemoJMFJ3D.

Unfortunately, the Processor class isn't required to support plug-ins; as a consequence, plug-ins don't work in JMF 1.0, and in some 2.0-based versions.

It’s a good idea to search the jmf-interest mailing list (`http://archives.java.sun.com/archives/jmf-interest.html`) before utilizing Sun’s JMF examples, since many of the programs have problems in different versions of JMF.