Playing Movies in a Java 3D World (Part 2)

Andrew Davison

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

ad@fivedots.coe.psu.ac.th

June 2nd 2005 (version 2)

In part 1, I described how to play a movie clip inside a Java 3D scene with the help of the Java Media Framework (JMF). The implementation uses the Model-View-Controller design pattern:

- the movie screen is the view element, represented by a JMFMovieScreen class;
- the movie is the model part, managed by a JMFSnapper class;
- a Java 3D Behavior class, TimeBehavior, is the controller, triggering periodic frame retrievals from the movie, which are drawn onto the screen.

In this part, I'll revisit the movie component, re-implementing it using *QuickTime for Java* (QTJ). QTJ provides an object-based Java layer over the QuickTime API, making it possible to play QuickTime movies, edit and create them, capture audio and video, and perform 2D and 3D animations. QuickTime is available for the Mac and Windows. Details about QTJ's installation, documentation, and examples can be found at `http://developer.apple.com/quicktime/qtjava/`.

As a consequence of the design pattern, the replacement of JMF by QTJ affects the application very little – only the movie class, JMFSnapper, departs, replaced by a QuickTime for Java version called QTSnapper.

Figure 1 shows two screenshots of the QTJ version of the Movie3D application: the picture on the right is a view of the screen from the back.

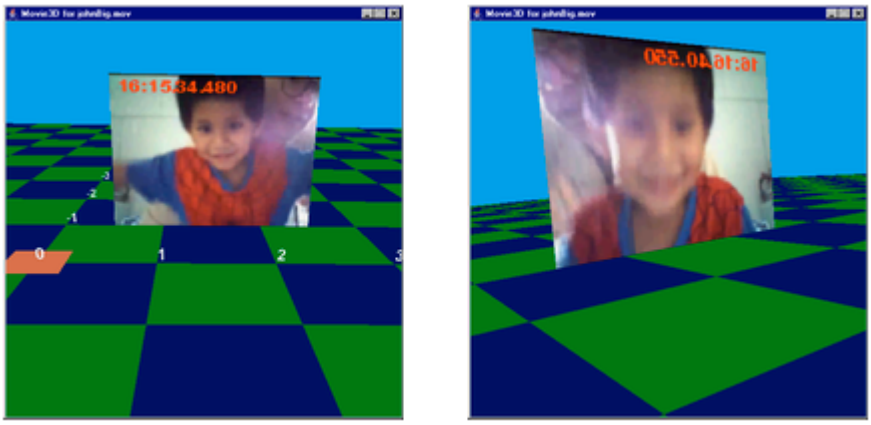

Figure 1. Two Views of the QTJ Movie3D Application.

A quick glance back at Figure 1 of Part 1 shows no obvious differences between the QTJ-based application and the JMF one.

However, a closer comparison of the executing programs reveals two changes: the QTJ movie is slightly more pixelated, *and* plays more slowly. The pixelation was introduced when the original movie was translated from MPEG to QuickTime's MOV format, and could be remedied with the help of a better conversion tool. The speed issue is more fundamental: it relates to the underlying implementation of QTSnapper.

The important elements of this article are:

- A discussion of the two main ways of implementing QTSnapper. One approach *renders every frame* of the movie onto the screen, while the other *selects a frame based on the current time*. This latter approach means that frames may be skipped, making the movie jitter, but the skipping permits the movie to play faster.
- The introduction of some simple frame/second measures (FPS), which I'll use to judge the relative speeds of the different approaches, and to detect skipped frames.

1. I'm Still on a Mountain, but a Different One

As in part 1, the code here utilizes two large APIs, which I don't have time to describe in any detail. I'm using Java 3D again, but switching media APIs from JMF to QTJ.

You'll find plenty of information about Java 3D in my O'Reilly book, *Killer Game Programming in Java* (KGPJ), including all the code for the checkerboard scene in Figure 1 where the movie screen is standing.

I won't be explaining the movie screen or the movie updating behaviour, since they're unchanged from Part 1.

What I will be doing is focussing on the QTJ techniques I employ in QTSnapper for extracting frames from the movie.

2. Two Overviews of the Application

The application's scene graph is shown in Figure 2.

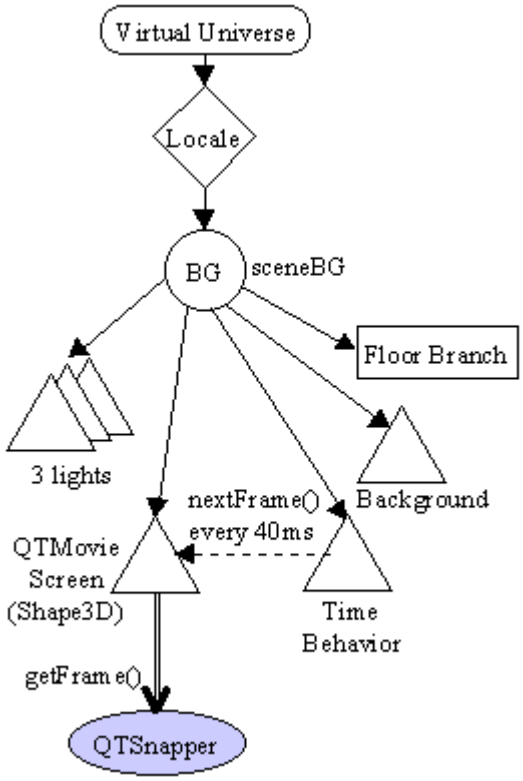

Figure 2. Scene Graph for Movie3D

This graph is almost identical to the one in Part 1.

The QuickTime movie is loaded by the QTSnapper class. The movie screen is created by QTMovieScreen, which manages a Java 3D quadrilateral (a *quad*) resting on the checkerboard floor. Every 40 ms, the TimeBehavior object calls the nextFrame() method in QTMovieScreen. That in turn calls getFrame() in QTSnapper to get a frame in the movie, which is laid over the quad managed by QTMovieScreen.

There's an important difference between JMFSnapper and QTSnapper. JMFSnapper returns the *current frame for the playing movie*, while QTSnapper returns the *current frame in the movie based on an incrementing index*.

For example, as getFrame() is called repeatedly in JMFSnapper, it may retrieve frames 1, 3, 6, 9, 11, etc., depending on when the method is called and the movie's playing speed. When getFrame() is called in QTSnapper, it will return frames 1, 2, 3, 4, etc.

The UML class diagrams for the application are given in Figure 3. Only the public methods of the classes are shown.

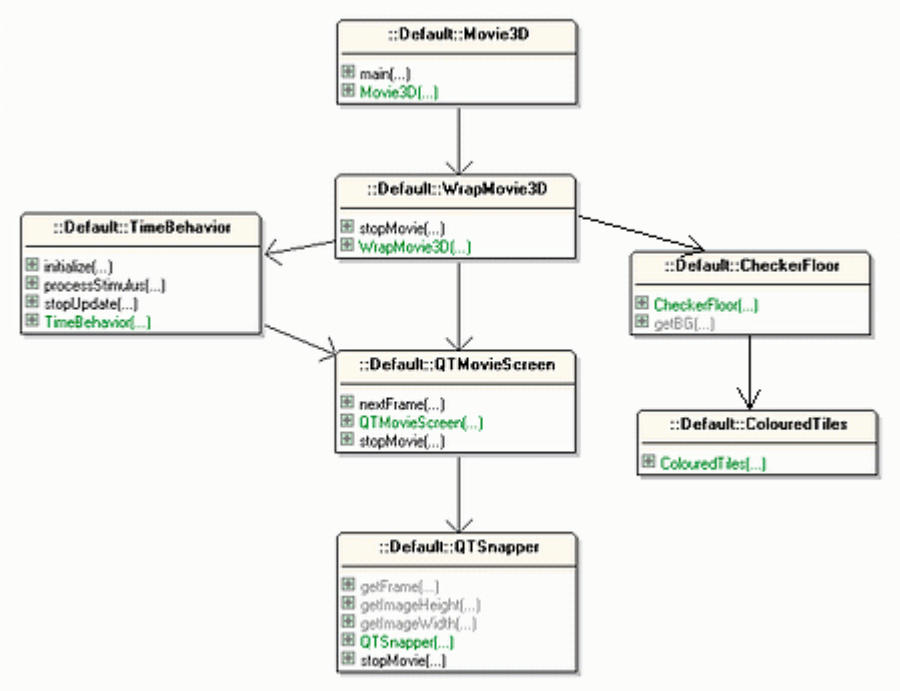

Figure 3. Class Diagrams for Movie3D

Aside from the names of the movie screen and movie classes (QTMovieScreen and QTSnapper), there's no difference between this figure and the class diagrams for the application in Part 1. In fact, only the internal implementation of the 'Snapper' class has altered.

To migrate between the JMF Movie3D application and this QTJ-based version requires the 'Snapper' class to be replaced. It's also necessary to change two lines in the movie screen class, where the 'Snapper' class is declared and instantiated:

```
// global variable
private QTSnapper snapper;     // was JMFSnapper

// in the constructor, load the movie in fnm
snapper = new QTSnapper(fnm);
```

These two changes are the only reason for renaming JMFMovieScreen to QTMovieScreen.

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. A Frame-By-Frame Movie

To understand the inner workings of QTSnapper, it helps to have a general idea of the structure of a QuickTime movie. Each movie may be composed of multiple audio and video tracks, overlapping in time. The general idea is illustrated by Figure 4.

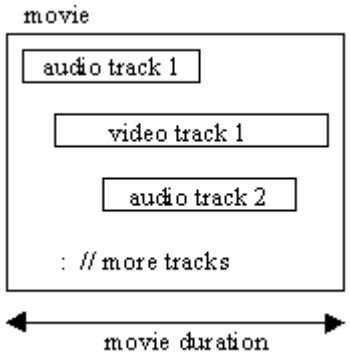

Figure 4. The Internals of a QuickTime Movie

Each track manages its own data, such as the type of media it contains, and the media itself. The media container has its own data structures, including its duration and playing rate (i.e. how many samples should be shown per second). The media is a series of samples (or frames), the first one starting at time 0 (with respect to media time). The samples are indexed, with the first sample at position 1 (not 0).

Simplified views of QuickTime's track and media structures are given in Figure 5.

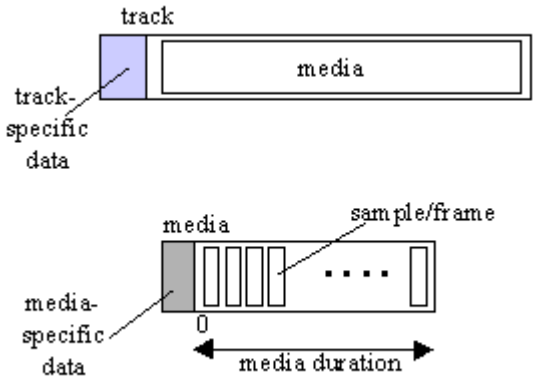

Figure 5. The Internals of a QuickTime Track and Media

For more comprehensive information, have a look at the movie section of the QuickTime tutorial at
`http://developer.apple.com/quicktime/qttutorial/movies.html`.

Accessing the Movie's Video Media

The QTSnapper constructor opens the movie:

```
// globals
```

```
private boolean isSessionOpen = false;
private OpenMovieFile movieFile;
private Movie movie;


// in the constructor,
// start a QuickTime session
QTSession.open();
isSessionOpen = true;

// open the movie
movieFile = OpenMovieFile.asRead( new QTFile(fnm) );
movie = Movie.fromFile(movieFile);
```

The call to QTSession.open() initializes QuickTime prior to its use. There should be a corresponding call to QTSession.close() at termination time.

The video track is located (if one exists) and its media accessed:

```
// more globals
private Track videoTrack;
private Media vidMedia;


// in the constructor,
// extract the video track from the movie
videoTrack = movie.getIndTrackType(1,
                 StdQTConstants.videoMediaType,
                 StdQTConstants.movieTrackMediaType);
if (videoTrack == null) {
  System.out.println("Sorry, not a video");
  System.exit(0);
}

// get the media used by the video track
vidMedia = videoTrack.getMedia();
```

Once the media is exposed, various information is extracted from it:

```
// more globals
private MediaSample mediaSample;
private int numSamples;    // number of samples in the movie
private int sampIdx;       // current sample index
private int width;         // frame width
private int height;        // frame height


// in the constructor
numSamples = vidMedia.getSampleCount();

sampIdx = 1;   // get the first sample in the track
mediaSample = vidMedia.getSample(0,
      vidMedia.sampleNumToMediaTime(sampIdx).time, 1);

// store the width and height of the image in the media sample
ImageDescription imgDesc =
      ImageDescription) mediaSample.description;
width = imgDesc.getWidth();
```

```
height = imgDesc.getHeight();
```

sampIdx is a counter which will iterate through the samples (the first sample starts at position 1).

The movie image's width and height are obtained by examining the first sample, under the assumption that all the samples use the same dimensions.

Measuring FPS

The number of frames returned per second (FPS) by QTSnapper will be used later to compare different implementation strategies for the class. The necessary elements are initialized in the constructor:

```
// frame rate globals
private long startTime;
private long numFramesMade;


// initialize them in the constructor
startTime = System.currentTimeMillis();
numFramesMade = 0;
```

Finishing Off

As the application is about to terminate, stopMovie() is called in QTSnapper. It reports the FPS, and shuts down QuickTime.

```
// globals
private DecimalFormat frameDf = new DecimalFormat("0.#"); // 1 dp


synchronized public void stopMovie()
{
  if (isSessionOpen) {
    // report frame rate
    long duration = System.currentTimeMillis() - startTime;
    double frameRate =
        ((double) numFramesMade * 1000.0) / duration;
    System.out.println("FPS: " + frameDf.format(frameRate));

    QTSession.close();  // close down QuickTime
    isSessionOpen = false;
  }
}
```

stopMovie() and getFrame() are synchronized so that it's impossible to terminate the QuickTime session while a frame is being copied from the movie.

Catching a Frame

getFrame() returns a single sample (a frame) from the movie as a BufferedImage object. The frame is selected using the index number stored in sampIdx (which goes from 1 to numSamples, then repeats).

```
// globals
private BufferedImage img, formatImg;


synchronized public BufferedImage getFrame()
{
  if (!isSessionOpen)
    return null;
  if (sampIdx > numSamples)   // start back with the first sample
    sampIdx = 1;

  try {
    // get the sample starting at the specified index time
    TimeInfo ti = vidMedia.sampleNumToMediaTime(sampIdx);
    mediaSample = vidMedia.getSample(0, ti.time, 1);
    sampIdx++;

    writeToBufferedImage(mediaSample, img);

    // resize img, writing it to formatImg
    Graphics g = formatImg.getGraphics();
    g.drawImage(img, 0, 0, FORMAT_SIZE, FORMAT_SIZE, null);

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

    numFramesMade++;   // count another frame generated
  }
  catch (Exception e) {
    System.out.println(e);
    formatImg = null;
  }

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

The sample is readily obtained by calling the getSample() method from QTJ's Media class. Unfortunately, there's still the tricky question of converting the sample into a BufferedImage, which I've hidden away inside my writeToBufferedImage() method.

The method performs some fancy translation dance steps, which I lifted from Chris W. Johnson's MovieFrameExtractor.java example, available from the quicktime-java mailing list (`http://lists.apple.com/archives/quicktime-java/2002/Feb/msg00062.html`).

The gory details, copiously commented, can be studied in the code. A 'raw' image is extracted from the sample, then decompressed as it's written into a QuickTime version of a Graphics object. The uncompressed data in the Graphics object is copied into

another 'raw' image, then into a pixel array (a PixMap). Finally this array is written into the DataBuffer part of an empty BufferedImage.

Does the Application Work? Does it Work Well?

Yes, Movie3D displays a movie, but large movies play slowly. This is due to getFrame()'s slowness in supplying frames, which can be quantified by looking at the FPS numbers.

For the movie in Figure 1, the reported FPS on a slow Windows 98 machine is in the range 15-17 frames/second. However, the TimeBehavior object is requesting an update every 40 ms, which should translate into frames appearing at nearer to 25 FPS.

getFrame() is slow because of the time-consuming conversion of the sample to a BufferedImage. As the current call to getFrame() gets bogged down converting a frame, further requests are delayed until the current one is finished.

I'll look at two ways of attacking this problem: permit getFrame() to skip frames when it finally gets to process a request, and try a different conversion strategy in getFrame (). I'll look at each of these in turn, starting with frame skipping

4. A Movie that Skips Frames

The new 'Snapper' class, QTSnapper1, still returns a frame when its getFrame() method is called. It differs from QTSnapper is that it supplies a frame corresponding to the current running time of the movie.

For example, getFrame() may retrieve frames 1, 2, 5, 8, 12, etc., depending on the when the method is called. Consequently, the movie progresses at a good speed, but the lack of frames may cause the movie to jitter.

In comparison, QTSnapper will return all the frames (1, 2, 3, 4, etc) but the delay between the getFrame() calls will make the movie run slowly. However, there won't be any jitter, since no frames are dropped.

The crucial element in QTSnapper1 is the notion of 'current running time' for the movie. My approach is to calculate the current running time *for the QTSnapper object* when getFrame() is called, then convert it to a *movie* running time, and finally to a sample index value.

QTSnapper1 has the same public methods as QTSnapper, so can be utilized in QTMovieScreen with minimal changes. The difference only becomes apparent when a movie is played – the movie rattles along at a good speed. Measurements, detailed later, put the 'apparent' frame rate for the example in Figure 1 at about 31 FPS, compared to 16 FPS for QTSnapper.

Accessing the Movie's Video Media

QTSnapper1 follows the same steps as QTSnapper to access the movie's video. Once the video is available, several media values are stored as globals, to be used later by getFrame():

```
// globals
private Media vidMedia;
private int numSamples;
private int timeScale;   // media's time scale
private int duration;    // duration of the media


// in the constructor,
// get the media used by the video track
vidMedia = videoTrack.getMedia();

// store media details for later
numSamples = vidMedia.getSampleCount();
timeScale = vidMedia.getTimeScale();
duration = vidMedia.getDuration();
```

Getting a Frame Now!

The new element of getFrame() is how it calculates the index value used to access a particular sample. The rest of the method, the call to writeToBufferedImage() and the code for writing the current time on the image, is the same as in QTSnapper.

```
// globals
private MediaSample mediaSample;
private BufferedImage img, formatImg;

private int prevSampNum;
private int sampNum = 0;
private int numCycles = 0;
private int numSkips = 0;


// inside getFrame(),
// get the time in seconds since the start of QTSnapper1
double currTime =
  ((double)(System.currentTimeMillis() - startTime))/1000.0;

// use the video's time scale
int videoCurrTime = ((int)(currTime * timeScale))% duration;

try {
  // backup the previous sample number
  prevSampNum = sampNum;

  // calculate the new sample number
  sampNum = vidMedia.timeToSampleNum(videoCurrTime).sampleNum;

  if (sampNum == prevSampNum)   // no sample change, so no need
    return formatImg;           // to generate a new image

  if (sampNum < prevSampNum)    // movie has just started over
```

```
    numCycles++;

  // record the number of frames skipped
  int skipSize = sampNum - (prevSampNum+1);
  if (skipSize > 0)  // skipped frame(s)
    numSkips += skipSize;

  // get a single sample starting at the sample number's time
  TimeInfo ti = vidMedia.sampleNumToMediaTime(sampNum);
  mediaSample = vidMedia.getSample(0, ti.time, 1);
```

getFrame() calculates the current time in seconds, measured from when QTSnapper1 started:

```
double currTime =
  ((double)(System.currentTimeMillis() - startTime))/1000.0;
```

Each QuickTime media has its own time scale, ts, such that one unit of time is 1/ts seconds. The time scale constant must be multiplied to currTime to get the current time in movie time units:

```
int videoCurrTime = ((int)(currTime * timeScale))% duration;
```

The time is corrected modulo the media duration, to allow the movie to repeat when the current time passes the end of the movie.

The time is mapped to a sample index number by calling Media's timeToSampleNum () method:

```
sampNum = vidMedia.timeToSampleNum(videoCurrTime).sampleNum;
```

The previously used sample number is stored in prevSampNum, to allow a number of tests and calculations to be carried out.

If the 'new' sample number is the same as the previous one, then there's no need to go through the lengthy process of converting the sample to a BufferedImage; getFrame() can return the existing formatImg reference.

If the new sample number is less than the previous one, this means that the movie has looped, and a frame from the start of the movie is about to be shown. This looping is registered by incrementing numCycles.

If the new sample number is greater than the previous number plus one, then the number of skipped samples is recorded.

Finishing Off

stopMovie() prints the FPS and closes the QuickTime session, in a similar way to the stopMovie() method in QTSnapper. It also reports additional information:

```
long totalFrames = (numCycles * numSamples) + sampNum;

// report percentage of skipped frames
```

```
double skipPerCent = (double)(numSkips * 100) / totalFrames;
System.out.println("Percentage frames skipped: " +
                          frameDf.format(skipPerCent) + "%");  // 1 dp

// 'apparent' FPS (AFPS)
double appFrameRate = ((double) totalFrames * 1000.0) / duration;
System.out.println("AFPS: " + frameDf.format(appFrameRate));  // 1 dp
```

appFrameRate represents the 'apparent' frame rate, which is the total number of samples that have been iterated over since QTSnapper1 began. It's 'apparent' in the sense that not all of those samples will have necessarily been rendered to the screen.

Does the Application Work? Does it Work Well?

With QTSnapper1 instead of QTSnapper, slow movies (such as the one in Figure 1) play much faster. The numbers reported at termination time put the apparent frame rate at about 31 FPS, the actual frame rate at around 16 FPS, and the percentage of skipped frames near to 50%. Surprisingly, this high level of missed frames is not that apparent on screen.

For other, smaller, movies, the speed-up is less noticeable; the percentage of skipped frames is around 5-10%.

Unfortunately, there are two problems: scrambled pixels when frames are skipped, and a lack of control over the number of frames that may be skipped.

Scrambled Pixels

Whenever QTSnapper1 skips a movie frame, the next retrieved frame will contain some scrambled pixels. This effect can be seen in Figure 6. The incorrect pixels use values from an earlier stage in the video.

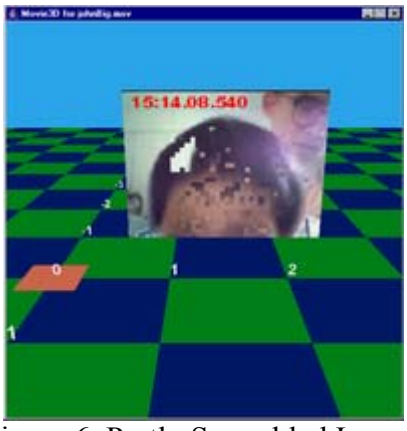

Figure 6. Partly Scrambled Image

I figured out a solution with the help of people from the quicktime-java mailing list (`http://lists.apple.com/archives/quicktime-java//2005/Jun/msg00005.html`). My special thanks to George Birbilis and Dean Perry.

The problem is that all my movie examples use *temporal compression*, a compression scheme which takes advantage of similarities between successive video frames. If two successive frames have the same background, then there's no need to store the background again. Instead, only the differences between the two frames are stored.

This technique, which is employed by almost all the popular video formats, means that the extraction of an image from a frame will depend on that frame *and* potentially several previous ones as well.

Temporal decompression is dealt with by a QuickTime DSequence object, which is employed by my writeToBufferedImage() method. The DSequence constructor specifies that QuickTime should use an offscreen *image buffer* during the decompression phase.

The frame's image is written to the buffer, where it's combined with earlier frame data. The resulting image is passed to the next stage of the conversion.

This works well when QTSnapper1 is decompressing samples in sequence, with no skipped frames (e.g. 1, 2, 3, 4,...), but skipping spells trouble. For example, what happens if QTSnapper1 skips frames 5 and 6, and then decompresses frame 7? The frame is written to a QuickTime image buffer, and combined with earlier frame data. Unfortunately, data from frames 5 and 6 is missing, and so the resulting image will be incorrect.

In brief, the scrambled pixels in an image are due to the use of temporal compression in the movie. One alternative is to use *spatial compression*, which compresses each frame individually, in a style similar to JPEG or GIF compression for a single image. This means that all the information for decompressing a frame is present in the frame itself; there's no need to examine earlier frames.

The QuickTime MOV format supports a spatial compression scheme called Motion-JPEG (M-JPEG). I used the export tool in QuickTime 6 Pro (`http://www.apple.com/quicktime/pro/`), to save the example in Figure 1 as a MOV file using the M-JPEG A codec. When this movie was played by the Movie3D application, no scrambling occurred.

Limiting Frame Skipping

Another issue with QTSnapper1 is that getFrame() doesn't place any limit on the number of frames that can be skipped. In my tests, the upper limit was a skip size of 3 frames, which wasn't noticeable. However, if getFrame() is given a larger sample (e.g. larger dimension, larger resolution) to convert, then it's increased slowness will be compensated for by more skipped frames. The movie quality could deteriorate very significantly.

5. Trying to Generate the Image Faster

The sample-to-BufferedImage conversion method, writeToBufferedImage(), used in QTSnapper and QTSnapper1, is taken from an example by Chris W. Johnson. Is there another, hopefully faster, way of extracting an image from a sample?

The standard book on QTJ is:

QuickTime for Java: A Developer's Notebook
Chris Adamson, O'Reilly, Jan 2005
`http://www.oreilly.com/catalog/quicktimejvaadn/`

Chapter 5 on QuickDraw contains a ConvertToJavaImageBetter.java example, which shows how to grab a sample as a PICT image, and convert it to a Java Image object. This example may also be found in the quicktime-java mailing list:

`http://lists.apple.com/archives/quicktime-java/2004/Jul/msg00032.html`

The conversion isn't straightforward, relying on the addition of a dummy 512-byte header to the PICT object so it can be treated as a PICT *file* by the QuickTime version of an ImageProducer.

I used Adamson's code as the basis of yet another 'Snapper' class, called QTSnapper2. It renders a frame sequence without skipping, in the same way as QTSnapper, but employs PICT-to-Image translation.

On small movies, QTSnapper2's performance is indistinguishable from QTSnapper, but for the slightly larger movie of Figure 1, its average frame rate descends to about 9 FPS, compared to QTSnapper's 16 FPS. In other words, PICT-based translation is slower than Johnson's technique.