An Introduction to Digital Video Data Compression in Java

Fore June

Chapter 10 Video Programming

10.1 Introduction

In order to experiment with the encoding and decoding of video data, we need to have a way to play the video on a PC. We need some well-developed tools to help us achieve this goal. Java has very powerful and complex tools to manipulate images and graphics. We also need some tools to process some video files that you can find in Internet and download them to carry out the tests and experiments. There exists a lot of video formats but we do not intend to address all of them; exploring video formats is **not** a goal of this book. Rather, we shall only discuss in detail the relatively simple AVI (Audio Video Interleaved) format and we use it as an intermediate format that we can read and save video data. There are also a lot of free utilities that we can use to change AVI files to other video formats and vice versa. To simplify things, we shall also make use of an open-source AVI (Audio Video Interleaved) library that can help us process AVI files.

The Abstract Windowing Toolkit (AWT) is a class library introduced in Java 1.0 for basic GUI interface. It also has image processing functions. A GUI interface created using AWT may look slightly different for different platforms. Later a user interface library code named "Swing" (or "Swing set") was created to provide platform-neutral GUI interface. Swing is the official name for the non-peer-based GUI toolkit that is part of the Java Foundation Classes (JFC), which are a set of classes that allow developers to write graphics and image applications. JFC is vast and is an integrated and core technology in the Java 2 platform (also code-named JDK 1.2). To enhance the image processing capabilities, java provides further libraries which may be huge to process images.

The **Java 2D API** is a set of classes for advanced 2D graphics and imaging, encompassing line art, text, and images in a single comprehensive model. The API provides extensive support for image compositing and alpha channel images, a set of classes to provide accurate color space definition and conversion, and a rich set of display-oriented imaging operators.

The **Java Advanced Imaging API** (JAI) provides a set of object-oriented interfaces to support a simple, high-level programming model. It extends the java 2 platform by allowing sophisticated, high-performance image processing to be incorporated into java applets and applications. It is a set of classes providing imaging functionality beyond that of Java 2D and the Java Foundation classes, though it is designed for compatibility with those APIs. JAI is complex and is a high-level API; it hides from users the details of image processing which may be complex. For example, using JAI, a user does not need to have any knowledge about the format of a JPEG image in order to render it.

The **Java Image I/O API** provides a pluggable architecture for working with images stored in files and accessed across the network. The JAI Image I/O Tools classes provide additional plugins for other stream types and for advanced formats such as JPEG-LS, JPEG2000, and TIFF.

The **Java Media Framework API** (JMF) enables audio, video and other time-based media to be added to applications and applets built on java platform technology.

All these java tools, libraries or APIs are huge and complex. We are not interested to study or use many of them. Our goal is to study the techniques and principles of compressing data, not the java APIs. Interestingly, java tries to hide data processing details from users and image data are operated at a very high level. It is inconvenient to use java to do something very simple like sending some data to the graphics framebuffer for rendering. Beginners may have to go through the maze of the APIs, tracing the functions of classes after classes to get to the right ones. Here we only discuss the classes and related ones for rendering. We try to make the process as simple as possible.

In this chapter, we shall first use the JAI to develop a simple video player (without audio part). Our goal of course is to decode the compressed data and render them on the screen. *Can we integrate the decoder and player seamlessly? Could we separate the decoder and the player functionalities so that changing the decoder would not affect the rendering and vice versa?* It turns out that this can be easily handled by the concept of the producer-consumer problem, which is a well-studied synchronization problem in Computer Science. In this case, the decoder is the producer which provides data and the player is the consumer that consumes the data. To accomplish these, they have to be run using different threads. Java provides simple thread functions that allow us to implement all these with relative ease. We shall discuss the principles and implementations of the player and related issues in the following sections.

10.2 Java Image Rendering

JAI is not the only API in java that provides image manipulating capabilities. AWT also provides functions to render and manipulate images. Actually, JAI is defined in a way that it is backward compatible with AWT imaging APIs defined by the **java.awt.image.Image** interface. The most fundamental and significant class in this API maybe the **java.awt.image. BufferedImage** class, which provides a 'framebuffer' for image data and relevant color models. Figure 10-1 shows the BufferedImage class and its encapsulated classes.

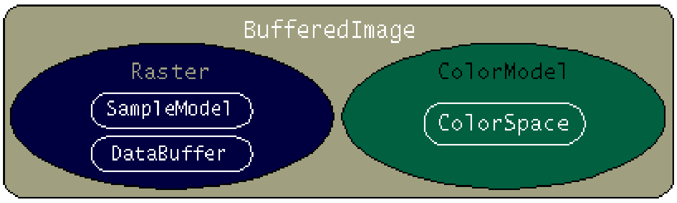

Figure 10-1 AWT BufferedImage Class

ColorModel

The **ColorModel** class shown in Figure 10-1 is an abstract class. An abstract class in java is a class declared *abstract* and cannot be instantiated (i.e. we cannot create objects of the class directly). However, we can extend it to a subclass (child class), which is not abstract and can be instantiated. An abstract class may or may not include abstract methods. An abstract method (function) is a method that is declared without an implementation; usually the implementations will be provided by subclasses. Though it cannot be instantiated, an abstract class can have static fields and methods, which can be referenced directly. (See the example in the section of **ColorSpace** below.) This class encapsulates methods for translating a pixel value to color components, red, green, and blue, and an alpha component, which denotes the degree of transparency at the pixel. In order to render an image to the screen, pixel values must be converted to color and alpha components. As arguments to or return values from methods of this class, pixels are represented as 32-bit ints or as arrays of primitive types. A **ColorSpace** class is used to specify the number, order, and interpretation of color components for the ColorModel. A ColorModel used with pixel data that does not include alpha information treats all pixels as opaque, which have an alpha value of 1.0. Java 2 provides three direct subclasses for ColorModel, namely, ComponentColorModel, IndexColorModel, and PackedColorModel. We shall only disucss and use the **ComponentColorModel** subclass here.

ComponentColorModel is a ColorModel class that works with pixel values representing color and alpha information as separate samples; each sample is stored in a separate data element. This class can be used with an arbitrary ColorSpace. The number of color samples in the pixel values must be the same as the number of color components in the ColorSpace. There may be a single alpha sample. For those methods that use a primitive array pixel representation of type *transferType*, the array length is the same as the number of color and alpha samples. Color samples are stored first in the array followed by the alpha sample, if present. The order of the color samples is specified by the ColorSpace. Typically, this order reflects the name of the color space type. For example, for TYPE_RGB, index 0 corresponds to red, index 1 to green, and index 2 to blue. ComponentColorModel has two constructors and we shall only use the following constructor in our applications:

```
public ComponentColorModel(ColorSpace colorSpace,
                           int[] bits,
                           boolean hasAlpha,
                           boolean isAlphaPremultiplied,
                           int transparency,
                           int transferType)
```

This constructor constructs a ComponentColorModel from the specified parameters. Color components will be in the specified ColorSpace. The supported transfer types are:

```
DataBuffer.TYPE_BYTE,
DataBuffer.TYPE_USHORT,
DataBuffer.TYPE_INT,
DataBuffer.TYPE_SHORT,
DataBuffer.TYPE_FLOAT, and
DataBuffer.TYPE_DOUBLE.
```

If not null, the *bits* array specifies the number of significant bits per color and alpha component and its length should be at least the number of components in the ColorSpace if there is no alpha information in the pixel values, or one more than this number if there is alpha information. When the *transferType* is DataBuffer.TYPE_SHORT, DataBuffer.TYPE_FLOAT, or DataBuffer.TYPE_DOUBLE the *bits* array argument is ignored. *hasAlpha* indicates whether alpha information is present. If *hasAlpha* is true, then the boolean *isAlphaPremultiplied* specifies how to interpret color and alpha samples in pixel values. If the boolean is true, color samples are assumed to have been multiplied by the alpha sample. The transparency specifies what alpha values can be represented by this color model. The acceptable transparency values are OPAQUE, BITMASK or TRANSLUCENT. The *transferType* is the type of primitive array used to represent pixel values. The following is an example of using this constructor:

```
int[] bits = { 8, 8, 8 };
ColorModel  colormodel = new ComponentColorModel(colorspace, bits,
          false, false, Transparency.OPAQUE, DataBuffer.TYPE_BYTE);
```

In this example, the red, green, and blue components are 8-bit samples. The ColorSpace class is discussed below.

ColorSpace

ColorSpace is also an abstract class which serves as a color space tag to identify the specific color space of a Color object or, via a ColorModel object, of an Image, a Buffered-Image, or a GraphicsDevice. For purposes of the methods in this class, colors are represented as arrays of color components represented as floats in a normalized range defined by each ColorSpace. For our programs here, we shall use the sRGB color space. sRGB (or standard RGB) color space is a proposed standard that uses the ITU-R BT.709 primaries, the same as are used in studio monitors and HDTV, and a transfer function (gamma curve) typical of CRTs. It is similar to the usual RGB color space but with more flexibilities, optimizing for the vast majority of computer monitors, operating systems and browsers.

As mentioned above, we can reference a static method of an abstract class without creating any class object. ColorSpace provides the static method **getInstance()** for us to create a ColorSpace object in a convenient way. The following is an example of creating an sRGB color space:

```
ColorSpace colorspace = ColorSpace.getInstance ( ColorSpace.CS_sRGB );
```

In the example, we can think of *colorspace* as a pointer pointing to a ColorSpace object. Note that though an abstract java class does not provide implementations to its abstract methods, it does provide implementations to its nonabstract methods. We can always access the nonabstract methods of an abstract class object. For example, the following piece of code makes use of the methods of ColorSpace to find out more information about the color space sRGB:

```
ColorSpace cs = ColorSpace.getInstance ( ColorSpace.CS_sRGB );
int n = cs.getNumComponents();
int dataType = cs.getType();
System.out.printf("\nNumber of Components in sRGB color space=%d",n);
System.out.printf("\nComponent MinValue  MaxValue  DataType  Name\n");
for ( int i = 0; i < n; i++ ) {
  float minValue = cs.getMinValue( i );
  float maxValue = cs.getMaxValue( i );
  String name =  cs.getName( i );
  System.out.printf("\n %d\t    %4.2f\t  %4.2f\t     %d \t    %s",
                          i,  minValue, maxValue, dataType, name);
}
System.out.println();
```

When executed, the above piece of code will produce some outputs similar to the following:

```
Number of Components in sRGB color space = 3

Component   MinValue    MaxValue  DataType  Name

 0            0.00        1.00       5      Red
 1            0.00        1.00       5      Green
 2            0.00        1.00       5      Blue
```

We can use this ColorSpace in the ColorModel discussed above to create color models. The following piece of code is an example of using the ColorModel to convert a color to a pixel sample:

```
int[] bits = { 8, 8, 8 };
ColorModel colormodel = new ComponentColorModel(cs, bits,
               false, false, Transparency.OPAQUE, DataBuffer.TYPE_BYTE);
Color color = new Color ( cs, new float [] {1.0f, 0.5f, 0.25f}, 0 );
float [] components = color.getComponents ( null );
```

```
int [] unnormalized = colormodel.getUnnormalizedComponents(
                                                components, 0, null, 0);
byte[] pixels =(byte[])colormodel.getDataElements(unnormalized, 0, null);
//convert to an unsigned byte
for ( int i = 0; i < 3; i++ ) {
   int pixelvalue = 0x000000ff & pixels[i];
   System.out.printf( "%d ", pixelvalue );
}
```

The code prints out the values “255 128 64” corresponding to the normalized red, green, and blue color values “1.0f, 0.5f, 0.25f”. In the code, we first create a ColorModel object using the subclass ComponentColorModel and sRGB color space as discussed above. A Color with normalized red, green, and blue values equal to 1.0f, 0.5f, and 0.25f respectively are created in the color space. Then the normalized color components are retrieved using **getComponents**(), which returns a float array, with the three elements, 1.0, 0.5, and 0.25. Next, these components are “unnormalized” using the **getUnnormalized**() method of the ColorModel class. Finally, the unnormalized components are converted to pixel samples using ColorModel’s getDataElements() method, which returns an array of the ColorModel’s transfer type (byte, in this case) containing the unnormalized component values. The last for-loop is to convert a signed 8-bit integer (byte) to an unsigned integer.

SampleModel

SampleModel is also an abstract class. It defines an interface for extracting samples of pixels in an image. All image data are expressed as a collection of pixels. Each pixel consists of a number of samples. A sample is a datum for one band of an image and a band consists of all samples of a particular type in an image. For example, a pixel might contain three samples representing its red, green and blue components; there are three bands in the image containing this pixel. One band consists of all the red samples from all pixels in the image. The second band consists of all the green samples and the remaining band consists of all of the blue samples. **ComponentSampleModel** is a subclass of SampleModel. It provides the implementations of the abstract methods of SampleModel. In our application, instead of using ComponentSampleModel to create SampleModel objects, we use a subclass of ComponentSampleModel, named **BandedSampleModel** to do the task. Figure 10-2 shows the relations between these classes:

java.lang.Object
    extended by **java.awt.image.SampleModel**
        extended by **java.awt.image.ComponentSampleModel**
            extended by **java.awt.image.BandedSampleModel**

Figure 10-2 Class BandedSampleModel

BandedSampleModel class represents image data which are stored in a band-interleaved fashion and for which each sample of a pixel occupies one data element of the DataBuffer. Accessor methods are provided so that image data can be manipulated directly. Pixel stride is the number of data array elements between two samples for the same band on the same scanline. The pixel stride for a BandedSampleModel is one. Scanline stride is the number of data array elements between a given sample and the corresponding sample in the same column of the next scanline. Band offsets denote the number of data array elements from the first data array element of the bank of the DataBuffer holding each band to the first sample of the band. The bands are numbered from 0 to N-1. Bank indices denote the correspondence between a bank of the data buffer and a band of image data. This class supports

TYPE_BYTE, TYPE_USHORT, TYPE_SHORT, TYPE_INT, TYPE_FLOAT, and TYPE_DOUBLE data types. The constructor

```
public BandedSampleModel ( int dataType, int w, int h, int numBands)
```

constructs a BandedSampleModel with the specified parameters. The pixel stride will be one data element. The scanline stride will be the same as the width. Each band will be stored in a separate bank and all band offsets will be zero. The following are the parameter specifications:

```
dataType - The data type for storing samples.
w - The width (in pixels) of the region of image data described.
h - The height (in pixels) of the region of image data described.
numBands - The number of bands for the image data.
```

The following is an example of using this constructor:

```
SampleModel  samplemodel =
        new BandedSampleModel ( DataBuffer.TYPE_SHORT, 128, 192, 3 );
```

In the example, the width of the image is 128 pixels, and the height is 192 pixels; the image data consists of 3 bands, red, green, and blue.

DataBuffer

DataBuffer is another abstract class, which exists to wrap one or more data arrays. Each data array in the DataBuffer is referred to as a bank. Generally, a DataBuffer object will be cast down to one of its data type specific subclasses to access data type specific methods for improved performance. Currently, the Java 2D API image classes use TYPE_BYTE, TYPE_USHORT, TYPE_INT, TYPE_SHORT, TYPE_FLOAT, and TYPE_DOUBLE DataBuffers to store image data; all of these data types are static fields and thus can be accessed in a form like “DataBuffer.TYPE_BYTE”.

Raster

Raster is a class representing a rectangular array of pixels. A Raster object encapsulates a DataBuffer object that stores the sample values and a SampleModel object that describes how to locate a given sample value in a DataBuffer object.

A Raster defines values for pixels occupying a particular rectangular area of the plane, not necessarily including (0, 0). The rectangle, known as the Raster’s bounding rectangle, which can be obtained through the getBounds() method, is defined by *minX, minY, width,* and *height* values. The *minX* and *minY* values define the coordinate of the upper left corner of the Raster.

One may use a SampleModel to describe how samples of a Raster are stored in the primitive array elements of a DataBuffer. Samples may be stored one per data element, as in a PixelInterleavedSampleModel or BandedSampleModel, or packed several to an element, as in a SinglePixelPackedSampleModel or MultiPixelPackedSampleModel classes.

The following is an example of using Raster and DataBuffer:

```
//allocate array to hold RGB data
int samples []  = new int[3*128*192];
.....
Byte[] bandValues = new Byte[1];
```

```
bandValues[0] = 3;
param1.add(new Float( 128.0 ));    // The width
param1.add(new Float( 192.0 ));    // Height
param1.add(bandValues);            // The band values
RenderedOp image = JAI.create("constant", param1 );

Raster ras = image.getData();
DataBuffer data = ras.getDataBuffer();
//output n sample data to raster
for ( int i = 0; i <  n; ++i )
  data.setElem ( i, samples[i] );
```

In the code, we have made use of the class RenderedOp, a node in a rendered imaging chain defined in Java Advanced Imaging (JAI), to create an "empty image". We then copy the data stored in the array *samples*[] to the raster of this image using the method "data.setElem(i, samples[i])".

TiledImage

TiledImage is the main class for writable images in JAI. It provides a straightforward implementation of the WritableRenderedImage interface, taking advantage of that interface's ability to describe images with multiple tiles. It may be the most efficient class that can create an image out of some given data. In Java2D, a tile is one of a set of rectangular regions that span an image on a regular grid.

The tiles of a WritableRenderedImage must share a SampleModel, which determines their width, height, and pixel format. The tiles form a regular grid, which may occupy any rectangular region of the plane. The contents of a TiledImage are defined by a single RenderedImage source provided by means of one of the set() methods or to a constructor which accepts a RenderedImage. The set() methods provide a way to selectively overwrite a portion of a TiledImage, possibly using a region of interest (ROI).

TiledImage also supports direct manipulation of pixels by means of the getWritableTile() method. This method returns a WritableRaster that can be modified directly.

Another way to modify the contents of a TiledImage is through calls to the object returned by createGraphics(), which returns a Graphics2D object that can be used to draw line art, text, and images in the usual Abstract Window Toolkit (AWT) manner.

Example

Putting all these together, we present a complete program that renders data to the screen. In this example, we hard-coded the image width and image height, and use BandedSampleModel to create a SampleModel object with the width and height and three bands. The sRGB color space is used in creating a ColorModel. The SampleModel and ColorModel objects are then used to create the TiledImage object outImage which will be used for rendering. Artificial data corresponding to an image which is half-red and half-white are saved in the integer array *samples*[]. The data are then assigned to *outImage* using the method setSample(). ScrollingImagePanel is used to render the *outImage*. The code is shown in Program Listing 10-1 and the corresponding output image is shown in Figure 10-3.

Program Listing 10-1: Example of Rendering Data

```
/*
  DataToImage.java
  Demonstrates how to render data as an image on the screen.
*/
import java.io.*;
import java.awt.Frame;
import java.awt.image.*;
import java.awt.image.ColorModel;
import java.awt.color.ColorSpace;
import java.awt.*;

import javax.media.jai.widget.ScrollingImagePanel;
import javax.media.jai.*;

public class DataToImage {

  public static void main(String[] args) throws InterruptedException {

    int width = 128;      //image width
    int height = 192;     //image height

    //create a SampleModel object
    SampleModel  samplemodel = new BandedSampleModel
                             ( DataBuffer.TYPE_BYTE, width, height, 3 );
    //use sRGB as our color space
    ColorSpace colorspace = ColorSpace.getInstance ( ColorSpace.CS_sRGB );

    //create a color model using our color space (sRGB), 8-bit components,
    // no alpha, no alpha premultiplied, opaque, data type is BYTE
    int[] bits = { 8, 8, 8 };
    ColorModel colormodel = new ComponentColorModel(colorspace,bits,false,
        false, Transparency.OPAQUE, DataBuffer.TYPE_BYTE );

    //allocate array to hold RGB data
    int samples []  = new int[3*width*height];

    //create a TiledImage using sample model and color model defined above
    TiledImage outImage;
    outImage = new TiledImage(0,0,width,height,0,0,samplemodel,colormodel);

    int isize = width * height;     //image size
    int bands = 3;                  //3 bands: Red, Green, Blue
    int k = 0;

    //create some artificial sample data for rendering
    for ( int i  = 0; i < isize; i++, k+=3 ){
        samples[k] = 255;           //red
      if ( i < isize / 2 ) {        //first half red
        samples[k+1] = 0;           //green
        samples[k+2] = 0;           //blue
      } else {                      //second half white
        samples[k+1] = 255;         //green
        samples[k+2] = 255;         //blue
      }
    }
    //assign data to the image
    k = 0;
    for (int y = 0; y < height; y++)
```

```
      for (int x = 0; x < width; x++)
        for (int band=0; band < bands; band++)
          outImage.setSample(x, y, band, samples[k++] );

    /* Attach image to a scrolling panel to be displayed. */
    ScrollingImagePanel panel =
               new ScrollingImagePanel( outImage, width, height );

    /* Create a frame to contain the panel. */
    Frame window = new Frame("Red and White");
    window.add(panel);
    window.pack();
    window.show();
    Thread.sleep( 10000 ); //sleep for ten seconds
    window.dispose();     //close the frame
  }
```

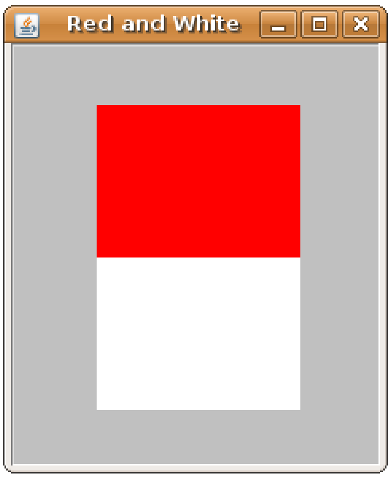

Figure 10-3 Image Created using TiledImage and Artificial Data

10.3 Threads

The effective use of threads is very important in modern programming. It allows a program to execute multiple parts of itself simultaneously in the same address space. In many cases, we basically cannot accomplish the tasks without using threads. For instance, consider a game program that needs to accept inputs from the mouse and keyboard, and at the same time has to play music at the background and generate some special sounds at various stages; it will be extremely difficult if not impossible to achieve these effects without using threads. Of course, we use threads only when we have to. We are not replacing simple nonthreaded programs with fancy, complex, threaded ones. Threads are just one more way we can use to make our programming tasks easier. The main benefits of using threads in programming include the following:

- gaining performance from multiprocessor hardware,
- easier programming for jobs with multi-tasks,

- ○ increasing job throughput by overlapping I/O tasks with computational tasks,
- ○ more effective use of system resources by sharing resources between threads,
- ○ using only one binary to run on both uniprocessors and multiprocessors,
- ○ creating well-structured programs, and
- ○ maintaining a single source for multiple platforms.

10.3.1 What Are Threads

A thread is also referred to as a **light weight process** (**LWP**). A process is a program (object code stored on some media) in execution. It is a unit of work in a modern time-sharing system. You can create several processes from the same program. A process not only includes the program code, which is sometimes referred to as the text section, but also the current activities and consumed resources including the program counter, processor registers, the process stack (which contains temporary data such as function parameters, return address, and local variable), and the data section, which contains global variables. A process may also have a heap, which is the memory dynamically allocated to it during run time.

Threads of execution, often shortened to threads, are the objects of activity within the process. Each **thread** is a basic unit of CPU utilization, comprising a **thread ID**, a **program counter**, a **register set**, and a **stack**. It shares with other threads of the same process its code section, data section, and other operating-system resources, such as open files, signals, and global variables. The various states of a thread can be represented by Figure 10-4, where quantum refers to the time the computer allocated to run a thread before switching to running another thread.

10.3.2 Pthreads

IEEE defines a POSIX standard API, referred to as **Pthreads** (IEEE 1003.1c), for thread creation and synchronization. It is defined in C language. Many contemporary systems, including Linux, Solaris, and Mac OS X implement Pthreads. To use Pthreads in your program, the user must include **pthread.h** and link with **-l pthread**. The following C/C++ exmaple, **pthreads.cpp** of Listing 10-2 shows how to use Pthreads:

Program Listing 10-2: Pthread Example in C/C++

```
/*
  pthreads.cpp
  A very simple example demonstrating the usage of pthreads.
  Compile: g++ -o pthreads_demo pthreads_demo.cpp -lpthread
  Execute: ./pthreads_demo
*/

#include <pthread.h>
#include <stdio.h>

using namespace std;

//The thread
```

```
void *runner ( void *data )
{
  char *tname = ( char * )data;

  printf("I am %s\n", tname );

  pthread_exit ( 0 );
}

int main ()
{
  pthread_t id1, id2;            //thread identifiers
  pthread_attr_t attr1, attr2;   //set of thread attributes
  char *tnames[2] = { "Thread 1", "Thread 2" }; //names of threads

  //get the default attributes
  pthread_attr_init ( &attr1 );
  pthread_attr_init ( &attr2 );

  //create the threads
  pthread_create ( &id1, &attr1, runner, tnames[0] );
  pthread_create ( &id2, &attr2, runner, tnames[1] );

  //wait for the threads to exit
  pthread_join ( id1, NULL );
  pthread_join ( id2, NULL );

  return 0;
}
```

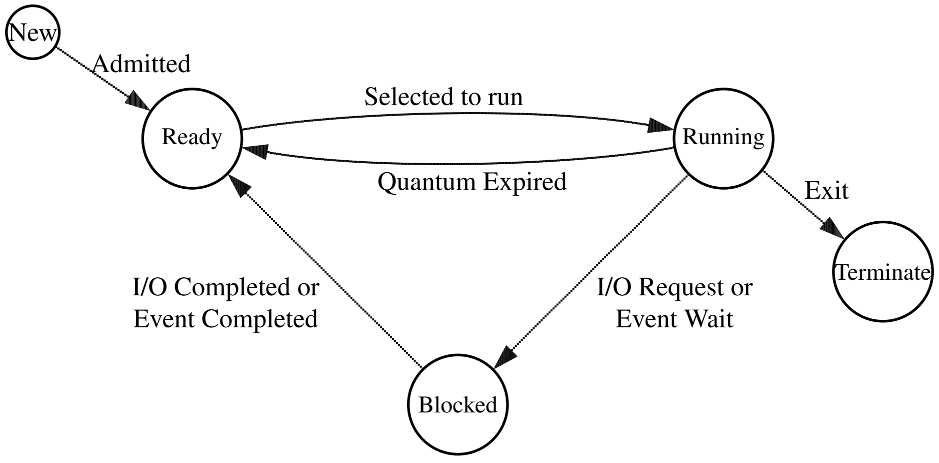

Figure 10-4. States of a Thread

In the example of **pthread.cpp** shown in Program Listing 10-2, we use **pthread_tid** to declare the identifiers for the threads we are going to create. Each thread has a set of attributes containing information about the thread like stack size and scheduling information. We use **pthread_attr_t** to declare the attributes of the threads and set the attributes in the function called by **pthread_attr_init**(). As we did not explicitly set any attributes, the default attributes will be used. The function **pthread_create**() is used to create a separate thread. In addition to passing the thread identifier and the attributes to the thread, we also pass the name of the function, *runner*, where the new thread will begin execution. The last argument passed to **pthread_create**() in the example is a string parameter containing the name of the thread. At this point, the program has three threads: the initial parent thread in **main**() and two child threads in **runner**(). After creating the child threads, the **main**() thread will wait for the **runner**() threads to complete by calling **pthread_join**() function.

Pthreads specification has a rich set of functions, allowing users to develop very sophisticated multithreaded programs. In the applications discussed here, we do not need to use many of the Pthread functions. Besides POSIX threads, different crucial thread programming schemes exist in the market. MS Windows has its own threading interface which is very different from POSIX threads. Though Sun's Solaris supports Pthreads, it also has its own thread API. Other UNIX systems may also have their own thread APIs. If you program in C/C++, an alternative way of creating threads is to use SDL threads which are platform independent. SDL solves the inconsistency of various thread-programming schemes with its own set of portable threading functions.

Java threads are relatively simpler and they are platform-independent as java byte codes run on java virtual machines.

10.3.3 Java Threads

Every thread in java is created and controlled by the **java.lang.Thread** class, which is a subclass of **java.lang.Object**. A java process can have many threads, and these threads can run concurrently, either asynchronously or synchronously. The following table shows some member methods of the Object class and Thread class:

Object	**Thread**

notify()
notifyAll()
wait()
	sleep()
	yield()

Every java thread has a priority; threads with higher priority are executed in preference to threads with lower priority. When code running in some thread creates a new thread object, the new thread has its priority initially set equal to the priority of the creating thread. There are two standard ways to create a new thread of execuation:

- ◦ implementing the Runnable interface (java.lang.Runnable) ,
- ◦ extending the Thread class (java.lang.Thread) and overriding its Run() method.

Implementing the Runnable Interface

The following example presents the code of creating a thread by declaring a class that implements the Runnable interface. The class then implements the run() method. An object of the class can then be allocated. In the example, the thread computes the modulo of an integer m with respect to an integer n and prints out the result:

```
    class ModRun implements Runnable {
      int m, n;
      ModRun ( int x, int y ) {
      m = x;
      n = y;
    }

    public void run() {
      System.out.printf ("%d mod %d is %d\n", m, n, m % n );
    }
  }
```

The following piece of code would then create a thread and start running it:

```
  public class RunnableExample {
    public static void main(String[] args) {
      ModRun modthread = new ModRun( 8, 3);
      //Start the thread
      new Thread ( modthread ).start();
      try {
        //delay for two second
        Thread.currentThread().sleep(2000);
      } catch (InterruptedException e) {
      }
      //Display info about the main thread
      System.out.println(Thread.currentThread());
    }
  }
```

This code generates the following output:

```
8 mod 3 is 2
Thread[main,5,main]
```

Extending Thread Class

The other way to create a new thread of execution is to declare a class to be a subclass of Thread. This subclass should override the run() method of class Thread. An object of the subclass can then be allocated and started. The above example of computing the modulo between two integers using this method could be written as follows:

```
class ModThread extends Thread {
  int m, n;
  ModThread ( int x, int y ) {
    m = x;
    n = y;
  }

  public void run() {
    System.out.printf ("%d mod %d is %d\n", m, n, m % n );
  }
}
```

The following code would then create a thread and start running it:

```
public class ThreadExample {
  public static void main(String[] args) {
    ModThread modthread = new ModThread( 8, 3);
```

```
      //Start the threads
      modthread.start();
      try {
        //delay for two second
        Thread.currentThread().sleep(2000);
      } catch (InterruptedException e) {}
      //Display info about the main thread
      System.out.println(Thread.currentThread());
    }
  }
```

This code generates the same output as the first method.

Usually the first method (implementing Runnable interface) is a preferrable way of creating threads. This is because we can then save our ‘subclassing’ of Thread for other purposes. Also, if for some reason our class is a final class so that we couldn’t make it a subclass, we must implement the Runnable interface to create thread execution. Moreover, a class might only be interested in being runnable, and therefore, inheriting the full overhead of the Thread class would be excessive.

10.4 The Producer-Consumer Problem

In Section 10.1, we have discussed how to render data as an image. We can modify it to a “video player” by adding a loop in **main**() to display a sequence of images: it reads in the image data from a file and saves it in a buffer *ibuf*, points the framebuffer to the data buffer, render the data to the screen, waits for a fixed period of time, and repeats the process by reading in another set of image data. Such a “video player” is single-threaded and is very inflexible. The whole program is dedicated to a single task, reading and displaying images. It cannot do other things like playing music or accepting inputs. If we add a decoder in the loop, it becomes difficult to synchronize the displaying speed and the decoding speed. We can overcome these shortcomings by changing the program to multi-threaded, and we will use the producer-consumer concept to handle the synchronization between decoding and rendering.

The producer-consumer problem is a common paradigm for thread synchronization. A **producer** thread produces information which is consumed by a **consumer** thread. This is in analog with whats happening in a fast-food restaurant. The chef produces food items and put them on a shelf; the customers consume the food items from the shelf. If the chef makes food too fast and the shelf is full, she must wait. On the other hand, if the customers consume food too fast and the shelf is empty, the customers must wait.

To allow producer and consumer threads to run concurrently (simultaneously), we must make available a buffer that can hold a number of items and be **shared** by the two threads; the producer fills the buffer with items while the consumer empties it. A producer can produce an item while the consumer is consuming another item. Trouble arises when the producer wants to put a new item in the buffer, which is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and finds it empty, it goes to sleep until the producer puts something in the buffer and wakes the consumer up. The **unbounded-buffer** producer-consumer problem places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items without waiting. The **bounded-buffer** producer-consumer problem puts a limit on the buffer size; the consumer must wait when the buffer is empty, and the producer must wait when the buffer is full.

The approach sounds simple enough, but if not properly handled, the two threads may **race** to access the buffer and the final outcome depends on who runs first. There is a simple technique to resolve the *race conditions*. E.W. Dijkstra introduced the concept of **semaphore** to handle synchronization problems in 1965. A semaphore is an integer variable associated with two operations, *down* and *up*, a generalization of *sleep* and *wakeup*, respectively. The *down* operation checks if the semaphore value is greater than 0. If yes, it decrements it and continues; if the semaphore value is 0, the thread is put on sleep. Checking the value, changing it, and possibly going to sleep are all done in a single, indivisible, **atomic action**. This it to guarantee that once a semaphore operation has started, no other thread can access the semaphore until the operation has completed or blocked.

Java uses a high-level synchronization technique called “monitors” to do synchronization; only one thread can be active inside a monitor. That is, only one thread can execute any of the methods at any time. In other words, a monitor is simply a lock that serializes access to an object. To gain access, a thread first acquires the necessary monitor, then proceeds. One can implement a semaphore using a monitor. A java monitor is signified by two basic synchronization idioms: *synchronized methods* and *synchronized statements*.

To make a method synchronized, we need to add the **synchronized** keyword to the declaration of a class like the following example:

```
class SynchronizedBuffer {
    private int buf = 0;

    public synchronized void write( int a ) {
      buf = a;
    }

    public synchronized void reset() {
      buf = 0;
    }

    public synchronized int value() {
      return buf;
    }
}
```

If *sharedBuffer* is an object of SynchronizedBuffer, then making these methods synchronized has two effects:

1. It is not possible for two invocations of synchronized methods on the object to interleave. That is, when one thread is executing a synchronized method for the object, all other threads that invoke synchronized methods for the same object will be blocked until the first thread has finished using the object.
2. When a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This ensures that changes to the state of the object are known to all threads.

Note that we cannot use the synchronized keyword in a constructor. If we do, a syntax error occurs. When a thread invokes a synchronized method, it automatically acquires the intrinsic lock of the associated object; the thread releases the lock when the method returns. The lock-release occurs even if the return was caused by an uncaught exception.

Another way to create synchronized code is to use *synchronized statements*. Unlike synchronized methods, *synchronized statements* must specify the object that provides the intrinsic lock like the following example:

```
class AddBuffer {
    private int buf = 0;

    public int addValue ( int a ) {
      synchronized ( this ) {
        buf += a;
      }
      return buf;
    }
}
```

Putting these together, we present the code of a simple program adopted from the official java web site in Lising 10-3, Listing 10-4, and Listing 10-5. The program simulates the producer-consumer problem.

Listing 10-3 presents the SingleBuffer class. The *buffer* variable of SingleBuffer holds one piece of item, which will be accessed by both the producer and consumer. Mutual exclusion of accessing *buffer* must be established between a procuder thread and a consumer thread. Therefore, the methods of SingleBuffer that access the shared variable *buffer* is "synchronized", which ensures that only one object can access each method at one time. The **wait**() function sends the accessing thread to sleep and allows another thread to access the SingleBuffer object; the sleeping thread will be wakened up by the second thread when it executes the function **notifyAll**(). In summary, the producer produces an item and puts it in the single buffer. (It is in analogy of a chef producing a hamburger and put it on a plate.) A consumer 'removes' the item from the single buffer. (This is in analogy of a customer removing the hamburger from the plate.) If the buffer is occupied (full), the producer goes to sleep and will be wakened up by the consumer. If the buffer is empty, the consumer goes to sleep and will be wakened up by the producer.

Program Listing 10-3: SingleBuffer object is shared

```
/*
  SingleBuffer.java
  The buffer variable is shared.  So it has to be synchronized.
  That is, only one object can access the synchronized method at a time.
*/
public class SingleBuffer
{
  private int buffer = -1;    //shared by producer and consumer
  private int count = 0;      //counts occupied buffers

  public synchronized void set( int value )
  {
    String name = Thread.currentThread().getName();
    while ( count == 1 ) {
      try {
        System.err.println( name + " tries to write." );
        System.out.println( "Buffer full. " + name + " waits. \t" +
                                  buffer + "\t" + count);
        wait();            //sleep if necessary
      } catch ( InterruptedException e ) {
        e.printStackTrace();
      }
    } //end while

    buffer = value;
    ++count;
    System.out.println( name + " writes \t\t" + buffer + "\t" + count );

    notifyAll();             //wake up other threads
  }  //end set()

  public synchronized int get()
  {
    String name = Thread.currentThread().getName();

    while ( count == 0 ) {
      try{
        System.err.println( name + " tries to read." );
        System.out.println( "Buffer empty. " + name + " waits. \t" +
                                 buffer + "\t" + count );
        wait();             //sleep if necessary
      } catch ( InterruptedException e ) {
        e.printStackTrace();
      }
    }  //while
    --count;
    System.out.println( name + " reads \t\t\t" + buffer +
                                 " \t" + count );
    notifyAll();       //wake up sleeping threads
    return buffer;
  }  //get()
} //SingleBuffer
```

The Producer and Consumer classes are presented in Listing 10-4. A shared SingleBuffer object is passed in as the argument to the Producer constructor. (More precisely, a pointer to the object is passed in.) The Producer thread puts a value in the *buffer* variable of the shared SingleBuffer object via the set() method. Similarly, the same SingleBuffer object is passed to a Consumer thread as the argument of the Consumer constructor. The Consumer 'removes' the value from *buffer* via the get() method. To simulate a real producer-consumer problem, both the Producer and Consumer thread sleeps a random amount of time between 0 to 3 seconds after performing a task; each thread performs its task for six times and then terminates:

Program Listing 10-4: Producer and Consumer Classes using a Single Buffer

```
/*
 *Producer-Consumer.java
 *Producer thread and Consumer thread shares a single buffer.
 *Only one thread can access the shared buffer at one time.
 */
class Producer extends Thread
{
  private SingleBuffer sharedBuffer;
  private int nTimes = 6;

  public Producer( SingleBuffer shared )
  {
    super ( "Producer" );
    sharedBuffer = shared;
  }
  public void run()
  {
    for ( int i = 1; i <= nTimes; i++ )
    {
        //sleep 0 to 3 seconds, then place value in Buffer
        try {
          Thread.sleep( (int) ( Math.random() * 3000 ) );
          sharedBuffer.set ( i ); //write to buffer
        } catch ( InterruptedException e ) {
          e.printStackTrace();
        }
    }//for
    System.err.println( getName() + " done producing." +
            "\nTerminating " + getName() + "." );
  }//end run
} //end class Producer

class Consumer extends Thread
{
  private SingleBuffer sharedBuffer;
  private int nTimes = 6; //number of times to run a loop

  public Consumer( SingleBuffer shared )
  {
    super( "Consumer" );
    sharedBuffer = shared;
  }
  public void run()
  {
    int sum = 0;

    for ( int i = 1; i <= nTimes; i++ )
    {
        //sleep 0 to 3 seconds, then reads value from Buffer
        try {
          Thread.sleep( (int) ( Math.random() * 3000 ) );
          sum += sharedBuffer.get(); //read from buffer
        } catch ( InterruptedException e ) {
          e.printStackTrace();
        }
    }//for
    System.err.println( getName() + " read values totaling: " + sum +
        "\nTerminating " + getName() + "." );
  } //end run
} //end Consumer
```

Listing 10-5 is a sample program that can be used to test the Procuder and Consumer classes which share a common SingleBuffer object. The SingleBuffer variable *sharedLocation* is used to construct a Procuder object and a Consumer object. The Producer thread puts a value in *sharedLocation* and the Consumer thread reads it.

Program Listing 10-5: Testing Producer and Consumer classes

```
//SharedBuffetTest creates producer and consumer threads
public class SharedBufferTest
{
  public static void main( String [] args )
  {
    SingleBuffer sharedLocation = new SingleBuffer();

    StringBuffer header = new StringBuffer( "Operation" );
    header.setLength( 40 );
    header.append( "\t\t\tBuffer\tCount");
    System.err.println( header );

    Producer p = new Producer( sharedLocation );
    Consumer c = new Consumer( sharedLocation );
    p.start(); //start producer thread
    c.start(); //start consumer thread
  } //main
}//SharedBufferTest
```

The following is a sample output of the program:

```
Operation                       Buffer  Count
Consumer tries to read.
Buffer empty. Consumer waits.   -1      0
Producer writes                 1       1
Consumer reads                  1       0
Producer writes                 2       1
Consumer reads                  2       0
Producer writes                 3       1
Producer tries to write.
Buffer full. Producer waits.    3       1
Consumer reads                  3       0
Producer writes                 4       1
Consumer reads                  4       0
Consumer tries to read.
Buffer empty. Consumer waits.   4       0
Producer writes                 5       1
Consumer reads                  5       0
Consumer tries to read.
Buffer empty. Consumer waits.   5       0
Producer writes                 6       1
Consumer reads                  6       0
Producer done producing.
Terminating Producer.
Consumer read values totaling: 21
Terminating Consumer.
```

In the above example, the buffer we use is a single buffer which can hold only one item. In many practical applications, we may use a circular queue to hold more than one item at a time. The producer inserts an item at the tail of the queue and the consumer removes an item at the head of it. We advance the tail and head pointers after an insert and a remove operation respectively. The pointers wrap around when they reach the “end” of the queue. If the tail reaches the head, the queue is full and the producer has to sleep. If the head catches up with the tail, the queue is empty and the consumer has to sleep. Actually, such a queue may handle the situation of multiple producers and multiple consumers. This concept is illustrated in Figure 10-5:

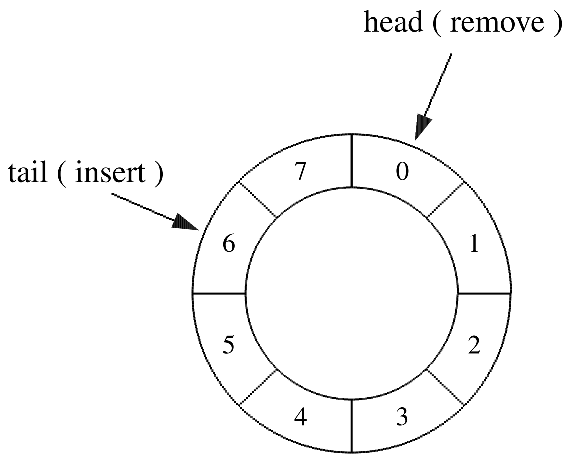

Figure 10-5. Circular Queue with Eight Slots

10.8 A Multi-threaded Raw Video Player

In this section, we put together what we have learned to develop a simple multi-threaded video player. Our concern here is to illustrate the concept of playing video data in an effective way. To simplify things, we hard-code the dimensions of a frame and assume that there's no compression in the data. We shall see that we can easily generalize the player to accommodate encoded data and data attributes.

A single-threaded raw video player is easy to implement: it just sits in a main loop to read in the video data and blit (bit-block transfer) them to the screen, wait for a while and repeat the data reading and blitting. In practice, a player has to handle various tasks besides blitting data on the screen. It may have to decode the data or to process the audio; a single-threaded player tangles all the tasks together and one needs to worry about the coordination between various tasks. On the other hand, a multi-threaded program can handle these tasks much better, as playing data (sending data to screen) can be cleanly separated from other tasks. The following section describes how to develop a multi-threaded program to play raw video data. The program and the sample raw data can be downloaded from this book's web site at *http://www.forejune.com/jvcompress/*.

In its simplest form, our multi-threaded player needs two threads, one for sending data to the screen and one for 'decoding' data (at this moment, 'decoding' is simply reading data from the file). Suppose we name these threads **Player** and **Decoder** respectively. This becomes a classical producer-consumer problem. Here, **Decoder** is the producer which produces resources (video data) and **Player**() is the consumer which consumes resources (video data). As discussed above, in the producer-consumer problem, to allow producer and consumer threads to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce zero or more items while the consumer is consuming an item; the number of items that can be produced or consumed depends on the production rate as well as the consumption rate and other factors that may influence the production and consumption operations. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced and the producer suspends production when the buffer is full as there will not be any space to hold the produced item. Therefore, the producer (**Decoder**) must wait when the buffer is full and the consumer (**Player**) must wait when the buffer is empty. In our implementation, the Decoder (producer) and Player (consumer) will operate independently by sharing a circular queue. The Decoder dumps data into the circular queue while the Player removes them from the queue. So our first task is to implement a circular queue that will be shared by the Player and Decoder.

Listing 10-6 presents the code of the class **CircularQueue** that implements a circular queue. The queue may have more than one slot and each slot can hold the data of one image frame. We assume that an RGB model is used for processing the image data. The length of the queue is passed in as a parameter of the constructor. Operations on the *head* and *tail* variables are synchronized. When the difference between the *tail* and the *head* is equal to the queue length, the queue is full. On the other hand, if the *tail* and the *head* point to the same slot (i.e. *tail* = *head*), the queue is empty. After producing an item, the producer has to increment *tail* and wakes up the consumer if it is sleeping by calling "notifyAll()"; these are implemented in the **tailInc**() method. On the other hand, after consuming an item, the consumer has to increment *head* and wakes up the producer if it is sleeping; these are implemented in the **headInc**() method. The operations "tail % length" and "head % length" implement the wrapping mechanism of the queue.

Note that in the class CircularQueue, the methods **setSamples**() and **putSamples**() are **not** synchronized. This is because when a consumer thread is reading the image data from the shared queue, we do not want to block the producer thread from writing data to it as long as they operate on different slots of the queue. Similarly, we do not block the consumer when the producer processes data from the queue.

Program Listing 10-6: CircularQueue Shared by Producer and Consumer

```
/*
 * CircularQueue.java
 * Implements a circular queue that may have more than
 * one slot.  Each slot may hold the data of an image frame.
 * Because setSamples() and putSamples() are not synchronized,
 * two threads can access the queue data simultaneously as
 * long as they are in different slots.
 * Assume an RGB model for the image.
 */
import java.io.*;
import javax.media.jai.*;

class CircularQueue
{
  private long head = 0;
  private long tail = 0;
  private int length;      //length of queue
  public int width;        //image width
  public int height;       //image height
  public byte buffer[][];  //the data queue
  public boolean quit = false;
  public CircularQueue ( int queueLength, int w, int h )
  {
    if ( queueLength > 0 )
      length = queueLength;
    else
      length = 1;
    width = w;
    height = h;
    int frameSize = 3 * width * height;
    buffer = new byte[length][frameSize];
  }

  public synchronized void headInc()
  {
    head++;
    notifyAll();   //wake up other threads
  }

  public synchronized void tailInc()
  {
    tail++;
    notifyAll();   //wake up other threads
  }

  public synchronized void waitIfBufferFull()
  {
     if ( tail >= head + length ){
      try {
        System.out.println( "Buffer full, producer waits." );
        wait();
      } catch ( InterruptedException e ) {
          e.printStackTrace();
      }
    }
  }

  public synchronized void waitIfBufferEmpty()
  {
    if ( tail <= head ){
      try {
        System.out.println( "Buffer empty, consumer waits." );
        wait();
      } catch ( InterruptedException e ) {
          e.printStackTrace();
      }
    }
  }

  //consumes data
  public void setSamples ( TiledImage outImage )
  {
    int bands = 3;
    int k = 0;
    int h = (int) ( head % length );     //wrap-around
    for (int y = 0; y < height; y++)
      for (int x = 0; x < width; x++)
         for (int band=0; band < bands; band++)
           outImage.setSample(x, y, band, buffer[h][k++] );
  }
```

```
  //produces data
  public int putSamples ( InputStream in )
  {
    int t = (int) ( tail % length );      //wrap-around
    int size = 3 * width * height;
    int num = 0;
    try {
        if (  ( num = in.read( buffer[t], 0, size ) )  < 0 )
         quit = true;
    } catch (IOException e) {
            e.printStackTrace();
            System.exit(0);
    }
    return num;  //return number of bytes read,-1 for end of data stream
  }
}
```

Typically, when the buffer is empty, the consumer goes to sleep and when the producer has finished producing an item and put it in the buffer, it is responsible to wake up the consumer. On the other hand, when the buffer is full, the producer goes to sleep and the consumer is responsible to wake up the producer after it has consumed an item. These mechanisms are implemented in the methods waitIfBufferFull(), waitIfBufferEmpty(), headInc(), and tailInc().

Once the CircularQueue class has been implemented, the implementations of the Player class (consumer) and the Decoder class (producer) are fairly straightforward. Listing 10-7 presents the code of the Player class. As discussed in Section 10.2, it uses the sRGB color space to create a color model, which in turn is used to create a TiledImage with the predefined image width and height. The TiledImage object (*outImage*) is attached to a ScrollingImagePanel, and the ScrollingImagePanel object (*panel*) is attached to a Frame object (*window*). The Player shares a CircularQueue with the Decoder. The CircularQueue method **setSamples**() is used to put the image data of a frame in *outImage*. The **set**() method of ScrollingImagePanel is utilized to send the image data of *outImage* to *panel* for display. The image frames are played at a rate of 20 frames per second (fps), which implies that the duration between two frames is 50 ms. To accomplish this frame rate, we use the function **System.currentTimeMillis**() to get the current time in milliseconds and insert a delay between playing frames estimated by the statement “delay = 50 - (int) (current_time - prev_time);”:

Program Listing 10-7: Player Class is a Consumer

```
//A consumer, CircularQueue buf is shared
class Player extends Thread
{
  private CircularQueue buf;
  private TiledImage outImage;
  private int width;
  private int height;
  private Frame window;
  private ScrollingImagePanel panel;

  //Constructor
  public Player( CircularQueue q )
  {
    super("Player");
    buf = q;              //shared CircularQueue
    width = buf.width;
    height = buf.height;

    //create a SampleModel object
    SampleModel  samplemodel = new BandedSampleModel (
                                DataBuffer.TYPE_BYTE, width, height, 3);
    //use sRGB as our color space
    ColorSpace colorspace = ColorSpace.getInstance ( ColorSpace.CS_sRGB );

    //create a color model using our color space (sRGB), each color 8-bit,
    // no alpha, no alpha premultiplied, opaque, data type is BYTE
    int[] bits = { 8, 8, 8 };
    ColorModel  colormodel = new ComponentColorModel( colorspace, bits,
        false, false, Transparency.OPAQUE, DataBuffer.TYPE_BYTE );

    //create a TiledImage using above the sample model and color model
    outImage = new TiledImage(0,0,width,height,0,0,samplemodel,colormodel);
    /* Attach image to a scrolling panel to be displayed. */
    panel = new ScrollingImagePanel( outImage, width, height );

    /* Create a frame to contain the panel. */
    window = new Frame("Raw Video Player");
    window.add(panel);
    window.pack();
    window.show();
  }

  public void run()
  {
     long prev_time = System.currentTimeMillis();    //time in ms
     long current_time;
     int delay;

     while ( !buf.quit ) {
       buf.waitIfBufferEmpty();
       //consumes the data
       buf.setSamples ( outImage );
       buf.headInc();   //advance head pointer
       current_time = System.currentTimeMillis();    //time in ms
       if (current_time - prev_time < 50)     //20 fps = 50 ms / frame
         delay = 50 - (int) ( current_time - prev_time );
       else
         delay = 0;
       prev_time = current_time;
       try { Thread.sleep( delay );} catch ( InterruptedException e ){}
       panel.set ( outImage );
     } //while
    window.dispose();
  }
}
```

The code of the Decoder class is particularly simple and is presented in Listing 10-8. The Decoder is a producer. At this point, the Decoder does not decode anything. It simply reads the image data from a file and puts them in the buffers of the shared CircularQueue.

It shares a CircularQueue with the Player. If the queue is full, it goes to sleep and will be wakened up by the Player when it calls the method **headInc**() of CircularQueue. Otherwise Decoder calls the **putSamples**() method of CircularQueue to deposit image data at the slot pointed by the variable *tail* of CircularQueue. After depositing the data, it calls **tailInc**() of CircularQueue to increment *tail* and wake up the Player if it is sleeping. The variable *quit* of CircularQueue is set to true when there is no more data.

In later chapters, we shall modify the Decoder class to include the task of decoding compressed data before putting the data in the queue.

Program Listing 10-8: Decoder is a Producer

```
//A Producer, CircularQueue buf is shared
class Decoder extends Thread
{
  private DataInputStream in;
  private CircularQueue buf;

  //Constructor
  public Decoder(DataInputStream ins,  CircularQueue q)
  {
    in = ins;
    buf = q;
  }

  public void run()
  {
     while ( !buf.quit ) {
       buf.waitIfBufferFull();
       //produce data
       buf.putSamples ( in );    //quit if out of data
       buf.tailInc();            //advance tail
     } //while
  }
```

Listing 10-9 presents a simple program for testing the Player and Decoder classes which share a common CircularQueue. In the program, the height and width of an image frame are hard-coded (320×240). The program creates a shared CircularQueue, a Player thread and a Decoder thread. Itreats the data in the file as a DataInputStream and passes the handle of the DataInputStream to the Decoder thread for “decoding”:

Program Listing 10-9: A Simple Multi-threaded Raw Video Player

```
/*
  RawPlayer.java
  Simple program for testing Player and Decoder which share
  a CircularQueue.  The height and width of an image frame
  are hard-coded ( 320 x 240 ).  It reads raw RGB video data
  from a file that has saved the data in raw 8-bit format.
*/
import java.io.*;

public class RawPlayer
```

```
{
  public static void main( String [] args )
  {
    if (args.length < 1) {
      System.out.println("Usage: java " + "RawPlayer" +
        " filename_of_raw_video_data\n" +
        "e.g. java RawPlayer ../../data/jvideo.raw"  );
      System.exit(-1);
    }

    //queue size, width, height
    CircularQueue sharedQueue = new CircularQueue( 4, 320, 240 );
    DataInputStream in;
    try {
      File f = new File ( args[0] );
      InputStream ins = new FileInputStream( f );
      in = new DataInputStream ( ins );
      //producer
      Decoder p = new Decoder( in, sharedQueue );
      p.start();                //start producer thread
    } catch (IOException e) {
       e.printStackTrace();
       System.exit(0);
    }
    //consumer
    Player c = new Player( sharedQueue );
    c.start();          //start consumer thread
  } //main
}
```

The user has to supply a file that has saved raw RGB video data in 8-bit format and the dimension of each frame is 320×240 in order to use the program. Again, a sample data file is provided in the web site of this book. For example, to excute the program, one can issue a command similar to the following:

```
java RawPlayer ../data/jvideo.raw
```

This plays the raw video saved in “jvideo.raw”. Figure 10-6 shows a sample output frame.

The program **RawPlayer.java** shown in Listing 10-9 uses hard-coded parameters and plays video data that are not compressed. If we need to play compressed data, we only have to change the **Decoder** thread, which decodes data and acts as the producer. Instead of using hard-coded parameters, we need to read in the parameters from the file containing the encoded data. Also, in order to test or perform experiments on our encoder and decoder, we need to download from the Internet some video files which are saved in a standard video format. We shall address these problems in the next chapter.

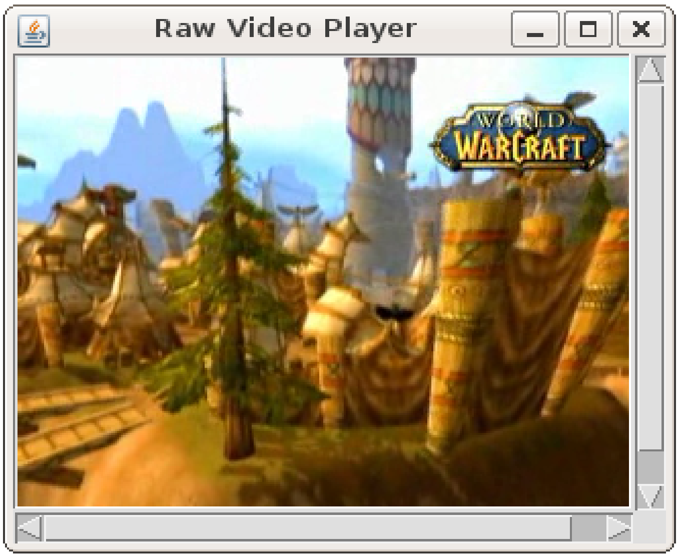

Figure 10-6 A Frame of Sample Raw Video