JDJEDGE CONFERENCE & EXPO MARCH 18–20, 2003 BOSTON, MA

JAVA™ DEVELOPER'S JOURNAL

The World's Leading Java Resource

March 2003 Volume:8 Issue:3

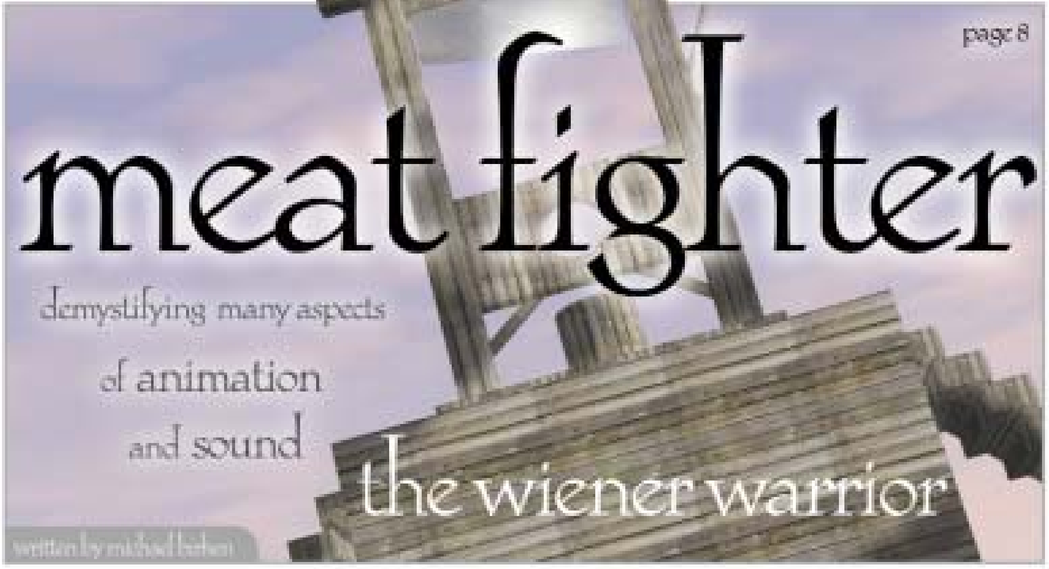
JAVADEVELOPERSJOURNAL.COM

FINAL CONFERENCE PROGRAM p.61

RETAILERS PLEASE DISPLAY UNTIL MAY 31, 2003

$5.99US $6.99CAN

SYS-CON MEDIA



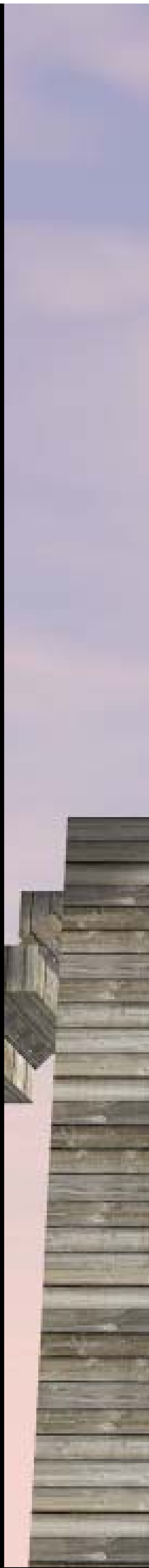
meat fighter

the wiener warrior

by michael birken

demystifying many aspects of animation and sound

If you have a bounty of creative energy and way too much time on your hands, why not explore the new Java 1.4 APIs by creating a video game. That's exactly what I did when I produced a parody of Street Fighter II called Meat Fighter (see Figure 1).



“Meat Fighter” is a side-scrolling one-to-one fighting game featuring anthropomorphic hot dogs and sausages. You can choose from six possible players and battle through seven stages of intensive meat fighting. So be prepared to meet the wiener warriors: Sal Lammee, Rat Dog, Oscar M. Wiener, Hot Doug, Cornelius Dog, and Oliver.

In this article, I present a simple framework to enable you to quickly take advantage of full-screen animation, music, and sound effects without learning all the intricate details of the new APIs. (The source code for this article as well as Listings 1–4 can be downloaded from www.sys-con.com/java/sourcec.cfm.) Check out Figures 1 and 2 for screenshots.

Getting Started with Animation

The animation in Meat Fighter, like any animation, is produced by displaying a series of still frames with minute variations in rapid succession. Each frame is generated in layers. For instance, during combat, the background is drawn first, followed by “background ornaments” such as the dancing crab on the beach stage, the fighters, and finally the vitality bars at the top. The element that appears closest to the front is the one that was drawn last in the sequence.

If this layering were done directly to the screen, the animation would appear to flicker wildly because the process of drawing the background erases all the foreground sprites. Instead, each animation frame is generated on an offscreen image buffer and transferred as a whole to the screen. To manage this process, I created the GraphicsSource class and the RenderListener interface.

A class that’s interested in generating animation frames implements RenderListener:

```
public interface RenderListener {
  public void init(
      boolean isPageFlipping,
      int bufferWidth,
      int bufferHeight);
  public void updateModel();
  public void render(Graphics g,
      boolean isBufferCleared,
      boolean isPageFlipping,
      int bufferIndex);
}
```

For each frame, GraphicsSource invokes updateModel() to update the state of the game, followed by render() to display the state to the offscreen image buffer. init() is invoked prior to frame generation on a different thread than the one for the animation loop. It should be used for loading images and sounds and for configuring the RenderListener.

All the methods of GraphicsSource are static. To initiate an animation, pass a RenderListener implementation to setRenderListener() and call startAnimation(). startAnimation() expects a frame rate value in frames-per-second (fps). Depending on the frame complexity and the speed of the underlying hardware, maintaining the specified fps consistently through an animation sequence may not be possible. Instead, GraphicsSource tries to sustain the apparent frame rate by assuming that the time required to update the game state is significantly less than the time to display it. It will occasionally and typically unnoticeably invoke updateModel() more often than render() to effectively skip frames. In this way, the game should run at approximately the same speed even on slower computers.

GraphicsSource also provides the means to enter and exit the full-screen exclusive mode via the methods enterDisplayMode() and exitDisplayMode(), respectively. The

FIGURE 1 Intro sequence

FIGURE 2 Game demo

enterDisplayMode() method requires the desired screen resolution and will attempt to find a mode of that size with the highest bit depth and refresh rate attainable. If the graphics environment cannot provide such a display mode, enterDisplayMode() will create a window with a drawing surface of the specified dimensions, and the game will run outside the full-screen mode. Since enterDisplayMode() also internally configures GraphicsSource for frame rendering, it must be the first method of GraphicsSource invoked.

Listing 1 illustrates all these methods. We'll get back to the arguments of init() and render() in the next section. The listing also introduces ImageSource, a class I created as a façade to the javax.imageio.ImageIO class. ImageSource provides two static methods, getImage() and getImages(), that load a single image and an array of images, correspondingly.

Meat Fighter uses exactly 212 GIFs placed in a directory called images on the same level as the gamingtools directory (see Table 1). Since this directory is within the classpath, the program can locate the files using the system class loader:

```
URL fileURL = ClassLoader
    .getSystemResource(IMAGES_DIR
        + fileName);
```

This technique works even after all the files are bundled into a JAR, which makes deployment one step easier.

Behind the Scenes

The process of rendering to an offscreen image buffer to reduce animation flickering is known as "double-buffering." The offscreen buffer is commonly referred to as the "back buffer" and the other buffer involved, the "front buffer," is the area of video RAM (VRAM) that's read from during the refresh cycle of the monitor to set the intensities of all the pixel phosphors.

Double-buffering is maintained by a java.awt.image. BufferStrategy object created by GraphicsSource.enterDisplayMode(). It will automatically select one of two techniques. If the BufferStrategy employs page flipping, then both the back and the front buffer are allocations of VRAM. The video pointer (a register on the video card) determines which buffer is read from during the monitor refresh cycle. By merely adjusting this pointer, the roles of the buffers are instantly exchanged (see Figure 3). The alternative strategy is bit blitting. In this case, the back buffer may be stored in VRAM or in ordinary system memory. After drawing to it is completed, it's copied to the front buffer (see Figure 4). For obvious reasons, bit blitting is less efficient than page flipping.

DIRECTORY	DESCRIPTION
/gamingtools	Reusable gaming classes not specific to Meat Fighter
/native	Location of the Windows-specific timer DLL
/images	Sprites and backgrounds used in Meat Fighter
/sounds	Sound effects and music used in Meat Fighter
/meatfighter	Meat Fighter–specific classes

TABLE 1 Directory structure of the game

Both strategies accomplish the same goal, but they affect how you must render each successive animation frame. To understand why, consider how to animate Pac-Man wandering around a maze. The simplest approach is to build up each frame in layers as described before. Specifically, draw the entire background image of the maze and then draw Pac-Man at the new location. But, a more efficient approach is to set up a clipping region around the previous location of Pac-Man and then draw the background image. We only restore the part of the maze that was painted over the last time Pac-Man was drawn to the buffer.

With this in mind, if page flipping is used, then even frames are not rendered to the same buffer as odd frames. This means that you must restore the maze considering Pac-Man's location two frames back. If bit blitting is used, you're always rendering to the same buffer; you only need to consider Pac-Man's previous position.

What makes things a little more complicated is that the buffers used by BufferStrategy are usually of type java.awt.image.VolatileImage. A VolatileImage offers significant performance benefits over other kinds of images because it stores the image contents in VRAM. However, VRAM is a limited resource and the operating system and other applications can borrow that memory for their own purposes at any time. For example, if a screen saver starts running in the middle of the game, since it's also a full screen application it will take away at least the memory used by the front buffer. After the screen saver stops, the VolatileImage will reallocate the VRAM, but by then the contents of the buffer are lost.

If the game is drawing to a buffer at the moment its contents are lost, no exceptions will be thrown. Instead, BufferStrategy provides methods that indicate if the contents of a buffer are still the same since the last time a graphics context was obtained for it; this information is passed to render() as the boolean parameter isBufferCleared. render() also receives an integer called bufferIndex that alternates between 0 and 1 when page flipping is used to indicate which buffer g refers to. It's always 0 for bit blitting.

When you call setRenderListener(), GraphicsSource calls back init() and passes it the type of buffer strategy in use, as the boolean isPageFlipping, and the dimensions of the buffers. GraphicsSource also assumes that the program is entering a new animation sequence and the contents of the buffers are no longer valid. This means that if page flipping is used as the buffer strategy, then isBufferCleared is set to true for the first two callbacks of render() directly after invoking setRenderListener(). Similarly, isBufferCleared is set to true for the first callback of render() if bit blitting is used. This saves you the trouble of writing special logic to initialize the buffers when the background changes for a different part of the game.

The Animation Loop

GraphicsSource.startAnimation() creates a thread that loops and calls back the methods of RenderListener until stopAnimation() kills it. The loop depends on high-resolution timing to decide when to invoke updateModel() more often than render(). I abstracted the concept of the timer into an interface called StopWatch:

```
public interface StopWatch {
  public void start();
  public long stop();
  public long getResolution();
}
```

start() begins timing and stop() returns elapsed time in nanoseconds (10^{-9} seconds).

getResolution() returns the error expected in the measured elapsed time, meaning the actual elapsed time is somewhere in the range of the measured elapsed time plus or minus this value.

My simplest implementation of StopWatch uses System.currentTimeMillis(); however, as the Javadoc explains, "While the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger." On my Windows XP box, I measured the resolution to be approximately 15ms. This is not good considering that at 60fps, the frame period is less than 17ms.

The Java Media Framework (JMF) and the Java 3D API (J3D) provide the high-resolution timers javax.media.SystemTimeBase and com.sun.j3d.utils.timer.J3DTimer, respectively. Although SystemTimeBase provides a getNanoseconds() method, I found it just as inaccurate as System.currentTimeMillis(). It may perform better on your box. J3DTimer, on the other hand, provides excellent high-resolution timing with a granularity less-than 1000 ns on my machine.

Resolution aside, the disadvantage of using these implementations is that JMF and J3D are not included as part of the standard installation of the Java 2 Runtime Environment (J2RE), which limits your gaming audience. So, I turned to the Java Native Interface (JNI) and created a DLL in Visual C++ that wraps the Windows functions QueryPerformanceFrequency() and QueryPerformanceCounter(). On application startup, a 40KB DLL is copied to the temporary directory defined by the system property java.io.tmpdir unless it's already located there. This is necessary because System.load(), used to load the DLL, cannot access files within a JAR.

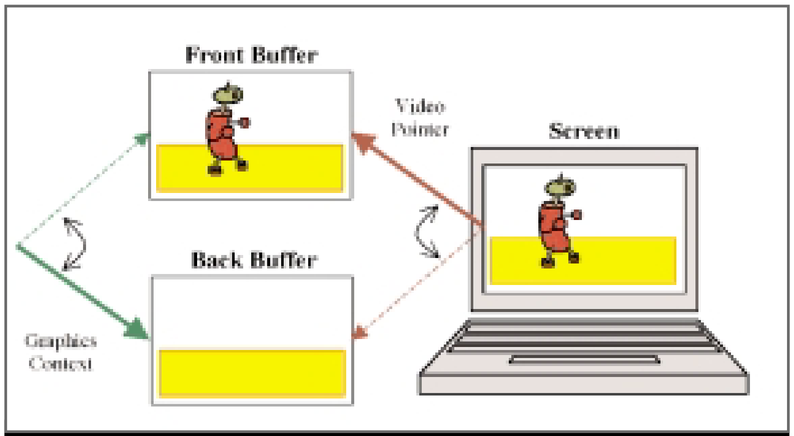


FIGURE 3 Page flipping

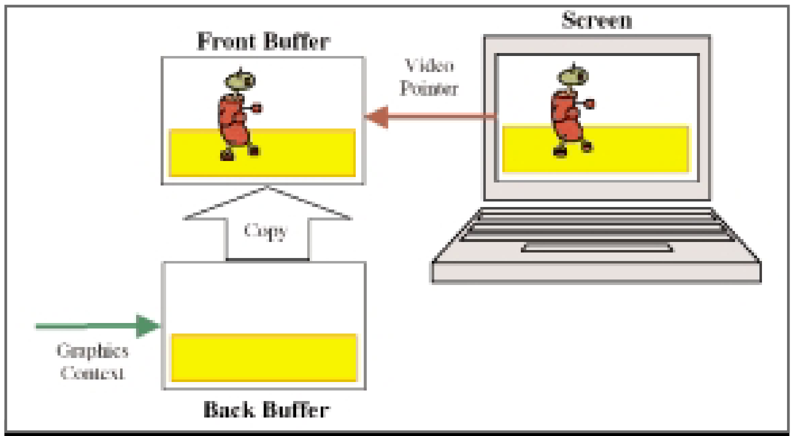


FIGURE 4 Bit blitting

StopWatchSource provides the static method getStopWatch(), which obtains the best available StopWatch implementation. To do so, it creates a List and adds the System.currentTimeMillis() implementation available on all platforms. Next, if JMF or J3D is installed, it will add their associated implementations. If the operating system is a version of Windows, it adds the JNI implementation. Finally, it sorts the List and returns the StopWatch with the highest resolution.

Listing 2 shows the animation loop in GraphicsSource that relies on StopWatch. The loop updates the game state and renders it, and then it sleeps for any time that's left over in the frame period. It computes the sleep time as:

```
long sleepTime = PERIOD
    - stopWatch.stop() - overSleepTime;
```

where PERIOD is the frame period and stopWatch.stop() returns the time expended in the last iteration while not asleep. Since Thread.sleep() is not accurate to the millisecond either (usually off about 1ms), we measure the actual sleep time and subtract that value from the sleep time of the successive iteration. This is the overSleepTime.

If sleepTime is negative, the duration of the last iteration exceeded the frame period. The overshoot, equal to -sleepTime, is typically a fraction of a frame period; it's added to a total called fractions. When fractions grows larger than the frame period, the animation is at least an entire frame behind. To compensate, the while-loop on line 57 invokes updateModel() for an additional fractions/PERIOD time(s) and leaves fractions with any remainder.

It Gets Easier

To create an animation frame without redrawing the entire background, render() requires access to the "dirty regions" of the buffer it's acting on. These are the rectangular areas made "dirty" as the result of drawing sprites on top of the background. If page flipping is used as the buffer strategy, you need to maintain two sets of dirty regions, one for each buffer. Bit blitting requires only one set.

To automatically maintain these sets, I created an abstract class called FrameBuilder. FrameBuilder implements the methods of RenderListener and actually declares them final to prevent you from overloading them. In this case, a class that desires to generate animation frames extends FrameBuilder and implements the five abstracts methods listed in Table 2a.

init()is called back soon after you pass the FrameBuilder reference to setRenderListener(). As before, it's invoked on a different thread than the one used for the animation loop; it can make blocking calls, such as loading images and sounds, without interrupting a running animation. After it returns, GraphicsSource switches to the new class to generate the frames.

The two overloads of renderBackground() are for drawing the entire background and for restoring a specified dirty region. The former version is invoked when the buffer contents are lost or isBackgroundSame() returns false. The latter version is called once per dirty region and is passed a graphics context with a clipping region set accordingly.

renderForeground() is invoked to display sprites and other graphics on top of the background. It should use the concrete methods listed in Table 2b. markDirtyRegion() records a dirty region for restoration in a successive frame. drawSprite() is a convenience method that invokes g.drawImage() followed by markDirtyRegion().

updateState() serves exactly the same purpose as updateModel(). An alternative method is used because isBackgroundSame() is actually checked in FrameBuilder's implementation of updateModel() instead of render(). This way FrameBuilder is able to keep track of background changes even when frames are skipped. Note: updateModel() delegates the call to updateState() before the check since isBackgroundSame() depends on the current game state.

It's important that these methods avoid creating temporary objects because the incremental garbage collection that occurs to clean up those objects will introduce noticeable pauses.

Listing 3 demonstrates a class that extends FrameBuilder. Observe that the code is completely oblivious to the buffer strategy. In this case, renderBackground() cleans up the dirty regions by invoking its overloaded counterpart. It can do this because its clipping region is automatically set to the dirty region. If you're developing a game like Pac-Man that uses tile graphics, you should provide logic that identifies and redraws only the dirty tiles (see Tile Graphic sidebar).

Returning to the subject of VolatileImages, sprites can also take advantage of them for the rendering performance benefits. To accomplish this, you would load an image into a nonvolatile form like a BufferedImage and then copy it to a VolatileImage for drawing. If the contents are ever lost, you can restore it from the nonvolatile version.

Luckily, BufferedImage already has this mechanism built in. When a BufferedImage is drawn repeatedly to a buffer in VRAM, Java 2D will automatically create a VolatileImage version of it so that future rendering may perform better. This being the case, ImageSource.getImage() returns a BufferedImage. The data layout and color model is set in accordance with the graphics configuration for optimal bit blitting.

Scrolling

If you load a background image wider than the screen, you could scroll it horizontally by redrawing it at different positions; however, this means you must paint the entire back buffer for each frame. Alternatively, you could allocate buffers wider than the screen in VRAM and adjust the video pointer offset so that the visible area of the front buffer changes. Since bit blitting is not used, this type of scrolling has no impact on performance whatsoever.

Unfortunately, BufferStrategy doesn't yet provide fine control of the video pointer; it's only capable of flipping the pointer between buffers. As such, Meat Fighter scrolls the background using the former technique, which is significantly slower. However, the background is not in continuous motion. It responds to the player's position; it's occasionally stationary and isBackgroundSame() returns true when the background position has not changed.

Sound Effects and Music

One of the coolest features of Java that actually evolved from the applet era is a simple interface for playing sound effects and music. The static method Applet.newAudioClip() obtains an AudioClip from a specified URL. AudioClip is an interface with three methods: play(), loop(), and stop(). It couldn't be easier.

However, after calling play(), there's no way to know when the audio ends. Such a feature is necessary for coordinating animation sequences with sound. For example, the animation at the start of Meat Fighter does not enter into the demo stage until the introductory music completely finishes.

To overcome this limitation, I created the AudioListener interface and an extension of AudioClip called Audio:

```
public interface AudioListener {
  public void audioStopped();
}

public interface Audio
    extends AudioClip {
  public void setAudioListener(
      AudioListener audioListener);
  public void clearAudioListener();
  public void dispose();
}
```

A class implements AudioListener to receive notification that a sound effect or a piece of music has completed playing. This listener is registered with an Audio object via setAudioListener().

Audio objects originate from the static method AudioSource.getAudio(). AudioSource uses the same technique as ImageSource to locate files within the classpath and it expects to find audio files in the sounds directory on the

JAVA GAME PROFILE (JSR-134)

Imagine if you could take that Java game you're working on, burn it onto a CD, pop it into your PS2 or Xbox, and actually play it. As Java programmers, we're used to porting our applications from one platform to another without any coding changes; why not apply the write once, run anywhere principle to game consoles?

The Java Game Profile (JGP) is an accepted Java Specification Request, designed with the help of companies like Sega and Sony, with the goal of developing a platform-independent standard for game development. Its ultimate goal is to rid the world of game ports by providing Java-based libraries capable of taking advantage of the high-performance graphics hardware found in modern game consoles as well as your PC. The proposal covers 2D and 3D graphics, streaming media, sound, controllers, physics modeling, and network communications, among other topics. It highly promotes leveraging existing APIs whenever possible and creating new APIs as needed to fill in the gaps. A game-specific API in the javax.games package would act as a layer above them all. For example, the proposal suggests classes for modeling features commonly found in game characters that rest on top of the Java 3D API.

Sony already encourages hobbyists to develop for the PS2 via its $200 Linux Kit. The kit includes a 40GB hard drive, an Ethernet adapter, keyboard, mouse, monitor cables, and the Linux software, which effectively turns the PS2 into a desktop computer. However, until Sun releases a JVM for it, it looks like you'll be stuck with gcc.

As for the Xbox, since it took a federal court to order Microsoft to distribute a modern JVM with Windows, it does not seem likely that Microsoft will participate in the JGP anytime soon. Almost unsurprisingly, hackers have managed to get a version of Linux running on the Xbox. But, if the JGP is going to be an attractive option for game companies as well as hobbyists, more console manufacturers will need to get involved.

You can view JSR-134 at www.jcp.org/en/jsr/detail?id=134. The JGP white paper, a document presented at the Game Developers Conference 2002 detailing JGP beyond JSR-134, has not been released to the Web at the time of this writing.

(2A) ABSTRACT METHODS
void init()
void updateState()
boolean isBackgroundSame()
void renderBackground(Graphics g)
void renderBackground(Graphics g, int x, int y, int width, int height)
void renderForeground(Graphics g)
(2B) CONCRETE METHODS
void drawSprite(Graphics g, Image image, int x, int y)
void markDirtyRegion(int x, int y, int width, int height)

TABLE 2 Methods of FrameBuilder

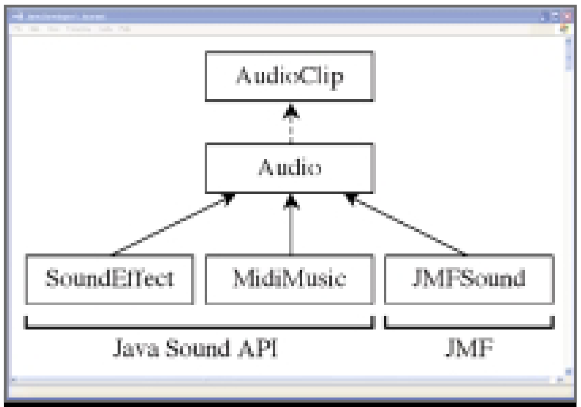


FIGURE 5 Implementations of Audio

TILE GRAPHICS

Tile graphics evolved from the earliest text-based games, where ASCII characters were arranged to form images of maps and objects in the game world. In those days, tiles served two primary purposes. First, they saved precious ROM space. For example, the 224x288 maze in Pac-Man actually consisted of 28x36 square tiles, each 8x8 pixels in size. Second, tiles provided a convenient method of bestowing a repeated behavior upon the game world. For instance, in Super Mario Brothers, the tiles that Mario walked on, which often hung in midair, all shared the behavior of a floor-like surface. Modern releases for systems like the handheld Nintendo Game Boy Advance still use tile graphics for exactly the same reasons.

To use FrameBuilder for a game like Pac-Man, you need a way to transform a dirty region into a set of dirty tiles. Assume the 28x36 maze is represented as a two-dimensional integer array of constants, each associated with a particular tile graphic. The ghost monsters and Pac-Man consist of 16x16 pixel sprites; rendering any of them means making nine of the 8x8 background tiles dirty. Given a sprite position (*x*,*y*), you can find its corresponding array element by dividing each coordinate by 8. Alternatively, you can right-shift by 3, an equivalent operation that takes less time to execute. Apply this to opposite corners of the sprite to compute the dirty tile region. *Note:* Pac-pellets are background tiles, not sprites. As Pac-Man gobbles them up, adjust the array elements accordingly and Pac-Man's position will automatically force the pellet tiles to be repainted.

A Super Mario Brothers type game presents a different challenge because the tiled backgrounds are one screen high and multiple screens wide. The Nintendo Entertainment System used 8x8 pixel tiles in a 32x28 grid to form 256x224 sized screens. Its hardware provided a wraparound buffer and the ability to adjust the video pointer within the buffer for scrolling. Since BufferStrategy does not support an equivalent, the entire back buffer needs to be repainted for each frame. One approach is to maintain two screen-sized segments of the background as BufferedImages and scroll from one to the other by drawing them end-to-end. As the first image is scrolled off, it would be incrementally updated one new column of tiles at a time. When the second image is completely in view, the first is ready to serve as the next segment.

same level as the gamingtools directory (see Table 1). Call dispose() after an Audio object is no longer required to release system resources that it may be holding. Listing 4 demonstrates how to create an Audio object and how to associate an AudioListener with it.

AudioSource.getAudio() returns one of three implementations of Audio depending on the file type (see Figure 5). SoundEffect and MidiMusic use the Java Sound API, which is part of the current J2RE installation, to provide sampled and synthesized sound, respectively. SoundEffect is capable of playing AIFF-C, AIFF, AU, SND, and WAVE files, and MidiMusic is capable of playing MIDI files. The quality of synthesized sound will vary depending upon which soundbank was shipped with the J2RE. A soundbank contains sound samples for an array of instruments. Typically, the J2RE installation includes the smallest and lowest quality one. See the links in the References section for information on how to determine which soundbank you're using and where to get a better one if required.

The Java Sound API does not provide support for MP3 files as of this time; however, the Java Media Framework (JMF) does and it's used by JMFSound. JMFSound is actually capable of playing all the sound file formats mentioned earlier in addition to MP3. The disadvantage of using it, as mentioned previously, is that JMF is not included in the standard J2RE installation. Note: The latest release of JMF actually removes some MP3 functionality due to "licensing issues," but you should still be able to play them under Windows.

AUTHOR BIO

Michael Birken is actively involved in the design and research of emerging trading technologies at a Manhattan-based financial software company. He's a Sun Certified Java programmer and developer. Michael holds a BS in computer engineering from Columbia University. Michael is a vegetarian.

Deployment

I JARred up all the resources into meatfighter.jar using this command:

```
jar cvfm meatfighter.jar theManifest
    gamingtools meatfighter native
    images sounds
```

The m option directs the JAR utility to use the manifest file, theManifest, instead of creating a default one. TheManifest is a one-line text file that specifies the class containing the main() method:

```
Main-Class: meatfighter.MeatFighter
```

You can launch Meat Fighter via:

```
java -jar meatfighter.jar
```

Alternatively, under Windows, you can double-click on the JAR to start it.

To promote reusability, I also created gamingtools.jar, which contains the gamingtools package and the native directory. You can find these JARs along with their complete sources on my Web site, www.meatfighter.com. All the code is covered by the GNU General Public License, so feel free to redistribute, modify, and use it in your own programs at no cost.

Conclusion

Since I posted Meat Fighter, I've received only positive feedback and a wealth of questions from enthusiastic Java game programming hobbyists. I hope this article has demystified many of the aspects of animation and sound and serves as a springboard for your creative energy. Unfortunately, at least for now, Game Over Man!

References

- Meat Fighter: www.meatfighter.com
- Full-Screen Exclusive Mode API: http://java.sun.com/docs/books/tutorial/extra/fullscreen/
- Java Image I/O API: http://java.sun.com/j2se/1.4.1/docs/guide/imageio/
- The VolatileImage API User Guide: ftp://ftp.java.sun.com/docs/j2se1.4/VolatileImage.pdf
- Java Sound API: http://java.sun.com/products/java-media/sound/
- Java 3D API: http://java.sun.com/products/java-media/3D/
- Java Media Framework: http://java.sun.com/products/java-media/jmf/
- Java Native Interface: http://java.sun.com/docs/books/tutorial/native1.1/
- JSR-134: www.jcp.org/en/jsr/detail?id=134
- PlayStation 2 Linux Community: http://playstation2-linux.com/
- Xbox Linux Project: http://xbox-linux.sourceforge.net/
- Nintendo Entertainment System Architecture: www.zophar.net/tech/files/nes.txt
- Swing Sightings Volume 3: http://java.sun.com/products/jfc/tsc/sightings/S03.html
- Java Gaming: www.javagaming.org
- Video Game Music Archive: www.vgmusic.com/
- Drawing Tablets: www.wacom.com
- History of Street Fighter: www.videogames.com/features/universal/sfhistory/

o__1@hotmail.com