Chapter 2 – 2D Graphics and Animation

Full-Screen Graphics

In Java there are three types of graphical games:

- Applets - Run in a web browser
- Windowed - Runs in a window
- Full-Screen - takes up the entire screen and can change graphics modes

The text concentrates on full screen games because they give the highest level of control over the system running them and they are more likely to fully immerse the player in the game world.

Key Ideas: The display on a computer consists of two main parts, the video card and the monitor. The video card stores the screen contents and can also hold images for hardware-accelerated graphics. The monitor translates the stored screen image into an actual light image.

Screen Layout

The screen of a computer is divided into pixels, single points of a certain color, that are arranged in a grid format with its origin at the top left of the screen.

The available resolutions depend on both the video card and the monitor. Typical resolutions include 640x480, 800x600, 1024x768, and 1280x1024. You should offer more than one resolution in your game because players will want to adjust for performance and because newer LCD displays can have problems with non-native resolution graphics.

Color Models and Primary Colors

- Color Models
  - CIE
  - CMY(K)
  - HSV
  - RGB

- Primary Colors
  - Dominant frequency
  - Combine two or more sources with different dominant frequency we can generate additional colors
  - The hues of the sources are called primary colors.
  - Two primaries that produce white are called complementary colors
  - No finite set of real primary colors can produce all visible colors
  - Given a set of 3 colors a fourth can be produced
**Pixel Color and Bit Depth**

**Pixel Color**
Computers use the RGB color model to control pixel color. In this system different levels of red green and blue are combined to make a color for display.

The number of colors a monitor can display depends on bit depth which is the number of bits used to store each pixel's color information.

**Bit Depth**
- 8 bit - $2^8 = 256$ colors
- 15 bit - 5 bits per color - $2^{15} = 32,768$ colors
- 16 bit - 5 bits for red and blue, 6 bits for green - $2^{16} = 65,536$ colors
  [note: the human eye is twice as sensitive to tones of green]
- 24 bit - 8 bits per color - $2^{24} = 16,777,216$ colors
- 32 bit - same as 24 bit, but it fits into a 32 bit space which is more convenient for computers to work with

---

**Refresh Rate**

The refresh rate is the number of times per second that the monitor redraws itself based on the content of the video card. Common rates are between 75Hz and 85Hz.
Switching the Display to Full-Screen Mode

To invoke full screen graphics and change graphics modes there are several objects we need to use.

- A Window object such as JFrame
- A DisplayMode object to specify what graphics mode to change to
- A GraphicsDevice object to change graphics modes and inspect display properties. The GraphicsDevice object is acquired from the GraphicsEnvironment object.

```java
JFrame window = new JFrame();
DisplayMode displayMode = new DisplayMode(800, 600, 16, 75);
// get the GraphicsDevice
GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice device = environment.getDefaultScreenDevice();
// use the JFrame as the full screen window
device.setFullScreenWindow(window);
// change the display mode
device.setDisplayMode(displayMode);
```

The previous code sample does not address possible exceptions that may be thrown, or allow much reusability. The text provides us with a SimpleScreenManager class that handles setting graphics modes.

```java
public void paint(Graphics g) {
    if (g instanceof Graphics2D) {
        Graphics2D g2 = (Graphics2D)g;
        g2.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    }
    g.drawString("Hello World!", 20, 50);
}
```

Anti-Aliasing

In the FullScreenTest program the text on the screen had jagged edges because it was not anti-aliased. Anti-aliased text is blurred or in some cases rendered at the sub-pixel level in order to make the edges look smoother.

To make text anti-aliased, set appropriate rendering hint before drawing any text. The functionality is present only in the Graphics2D class, a subclass from Graphics.

Anti-Aliasing (cont.)

- For backward compatibility, the `paint()` method takes a Graphics object as parameter
- Here is the code for a paint method with anti-aliasing:

Which Display Mode to Use

- Your game should be able to run in more than one display mode.
- When possible allow the player to select the same resolution as the current resolution.
- Bit depth: 16, 24, and 32 bit color are all good selections. 16 bit is faster while 24 and 32 bit graphics look better.
- A refresh rate between 75Hz and 85Hz is suitable for the human eye.

Images

- Transparency
  - There are three types of image transparency, opaque, transparent, and translucent.
  - Opaque - every pixel is drawn
  - Transparent - a pixel is either visible or not visible
  - Translucent - a pixel can be partially visible (the final color is a weighted average of the color in the picture and the color behind it)
File Formats

Java supports three image formats, GIF, PNG, and JPEG.

- GIF - can be opaque or transparent. Limited to 8 bit color.
- PNG - can be opaque, transparent, or translucent. PNG images also support any bit depth.
- JPEG - opaque 24 bit images only. Works well for photographs, however the compression method is based on an 8x8 grid which can give diagonal lines a stair-step appearance.

Apache's Scalable Vector Graphics (SVG) implementation, called Batik, at http://xml.apache.org/batik/

Reading Images

To read an image file you can use Toolkit's `getImage()` method which parses the file and returns an Image object.

```java
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image = toolkit.getImage(fileName);
```

The `Toolkit.getImage()` method starts another thread to load the image. If the image is displayed before it is finished loading it will not display correctly.

The ImageIcon class in the `javax.swing` package loads an image using the Toolkit and waits for it to finish loading before it returns.

```java
ImageIcon icon = new ImageIcon(fileName);
Image image = icon.getImage();
```

The ImageTest program uses the SimpleScreenManager class to establish fullscreen mode then draws a JPEG background image and four PNG foreground images.

Hardware Accelerated Images

Java tries to accelerate all images that are loaded with Toolkit’s `getImage()` method by default. There are times that an image will not be automatically accelerated, such as:

- If the image is constantly being changed (for instance being drawn on)
- If the image is rendered translucently. Only opaque and transparent rendering is accelerated as of java 1.4.1.
- If the system does not support hardware acceleration.

A `VolatileImage` is stored in video memory and thus it is forced to be accelerated.

```java
VolatileImage image = createVolatileImage(int w, int h)  
```

A VolatileImage can lose its contents at any time such as when another program uses video memory.

The `validate()` and `contentsLost()` methods can be used to detect if the image data has been lost. `validate()` makes sure the image is compatible with the current graphics mode and `contentsLost()` returns information about if the image has been erased since the last call to `validate()`.

Hardware Accelerated Images (2)

- VolatileImages are created using Component’s `createVolatileImage(int w, int h)` method or GraphicsConfiguration’s `createCompatibleVolatileImage(int w, int h)` method. Unfortunately, VolatileImages can be only opaque.
- A VolatileImage can lose its contents at any time such as when another program uses video memory.
- The `validate()` and `contentsLost()` methods can be used to detect if the image data has been lost. `validate()` makes sure the image is compatible with the current graphics mode and `contentsLost()` returns information about if the image has been erased since the last call to `validate()`.

Image Drawing Benchmarks

The ImageSpeedTest program is a modified version of the ImageTest program that spends one second each drawing the four types of images tested in the ImageTest program and prints a display of how many of each type of image was drawn.

```java
ImageSpeedTest test = new ImageSpeedTest();
```

Note: You should not spend large amounts of time in the paint() method as in ImageSpeedTest. It will prevent the AWT event dispatch thread from performing its other duties such as handling user input.

To give you an idea, here are the results of this test on a 600MHz Athlon with a GeForce-256 video card, a display resolution of 800x600, and a bit depth of 16—and in a good mood at the time:

- Opaque: 5550.599 images/sec
- Transparent: 5478.6953 images/sec
- Translucent: 85.2197 images/sec
- Translucent (Anti-Aliased): 113.18243 images/sec
Animation

First will look at cartoon-style animation. This is where several images are drawn in a sequence to create the illusion of movement.

Each image is a frame and each frame is displayed for a certain amount of time. An example of how this can be done in code is shown in Animation.java.

Active Rendering

Active Rendering is a term used to describe drawing directly to the screen in the main thread. Using active rendering means that you do not need to wait for the paint() method to be invoked by the AWT event dispatch thread which may be busy. The following code shows an example of using active rendering:

```java
Graphics g = screen.getFullScreenWindow().getGraphics();
draw(g); // draw is a method you define
g.dispose();
```

In this code the graphics context for the screen is obtained using Component's getGraphics() method. Then the draw method, defined in the main class, draws directly onto the screen. The graphics device is disposed because the garbage collector may take some time to get to it and the object is created on every screen update.

The Animation Loop

The following program, AnimationTest1, uses active rendering to draw the animation continuously in a loop. The steps of the animation loop are:

1. Update any animations
2. Draw to the screen
3. Optionally sleep for a short time
4. Return to step 1

Getting Rid of Flicker and Tearing

You probably noticed that AnimationTest1 looks terrible. The image may be constantly flickering because it is being drawn directly on the screen and then drawn over by the background before it is drawn again. To address this problem we introduce a buffer.

Double Buffering

An animation that is double buffered is first drawn to an image somewhere in memory (the back buffer), then that image is copied to the screen once it is completed.

Page Flipping

- When using double buffering you always have to copy an image the size of the entire screen to draw each frame. Page Flipping allows you to skip this extra copying step.
- In page flipping you have two buffers that are usually both in video memory. One buffer acts as the screen device and the other holds the next frame until it is ready to be displayed. Then the graphics device's display pointer is changed from one buffer to the other making it the source for the monitor's screen refresh instantly. The process repeats with the buffers in exchanged roles.
- In page flipping the screen is updated instantly without any copying of data.

Monitor Refresh and Tearing

The monitor will update itself based on video memory a certain number of times per second. If the display pointer is changed from one image to another during a redraw then the top of the monitor displays a different image than the bottom. This is called tearing because it looks as though things on screen are being torn in half.

To avoid this problem the display pointer must be changed between refresh cycles. This can be achieved using java's BufferStrategy class.
The BufferStrategy Class

The BufferStrategy class will handle buffering for you completely. It will try page flipping first. If that is not possible it will try double buffering. It will also wait on monitor refresh for you to prevent tearing.

Canvas and Window objects can have a BufferStrategy. For double buffering or page flipping you must create a BufferStrategy with two or more buffers as in the following code:

```java
frame.createBufferStrategy(2);
```

Then use the GetBufferStrategy() method to access the buffer. The getDrawGraphics() method returns the draw buffer and the show() method updates the display. The following code puts these ideas to use:

```java
BufferStrategy strategy = frame.getBufferStrategy();
Graphics g = strategy.getDrawGraphics();
draw(g);
g.dispose();
strategy.show();
```

Creating a Screen Manager

The ScreenManager class is an improved version of the SimpleScreenManager class that adds the following features:

- Double buffering and page flipping using the BufferStrategy class
- `getGraphics()` which gets the graphics context for the display
- `update()` which updates the display
- `getCompatibleDisplayModes()` Gets a list of the compatible display modes
- `getCurrentDisplayMode()` gets current display mode
- `findFirstCompatibleMode()` which gets the first compatible mode from a list of modes

In the new ScreenManager class make sure to note the new methods:

- `displayModesMatch()` and
- `createCompatibleImage()`

Sprites

A sprite is a graphic that moves independently around the screen. Sprites can also be animated.

The following Sprite class defines a movement based on its position and velocity. By basing the velocity on time the sprite is made to move the same speed no matter what speed the machine renders frames.

```java
Sprite.java
```

Simple Effects

Transforms

Transforming an image allows you to do such things as rotate, scale, flip, and shear images. These effects can be performed in real time but they are not hardware accelerated.

The AffineTransform object describes a transform. The class provides methods for controlling transforms such as `rotate()`, `scale()`, and `translate()`.

There is a special `drawImage()` method int the Graphics2d object that takes an AffineTransform object as a parameter.

SpriteTest2 uses transforms to make the sprite face in the direction it is going.
Chapter 2 – 2D Graphics and Animation

- Full Screen Graphics
- Screen Layout
- Pixel Color and Bit Depth
- Refresh Rate
- Switching the Display to Full-Screen Mode
- Anti-Aliasing
- Which Display Mode to Use

- Images
  - Transparency
  - File Formats
  - Reading Images
  - Hardware-Accelerated Images
  - Image Drawing Benchmarks
  - Animation
  - Active Rendering
  - The Animation Loop

- Getting Rid of Flicker and Tearing
  - Double Buffering
  - Page Flipping
  - Monitor Refresh and Tearing
  - The BufferStrategy Class
  - Creating a Screen Manager
  - Sprites

- Simple Effects
  - Image Transforms