JavaOne℠

Sun's 2000 Worldwide Java Developer Conference℠

JavaOne℠

Sun's 2000 Worldwide Java Developer Conference℠

Getting Started With the Java™ Media APIs

Bill Day
day@acm.org
www.billday.com
Sun Microsystems, Inc.

Agenda

- Overview of the Java™ Platform
- Introduction to the Java Media APIs
- Java 2D™ API
- Java Media Framework
- Java 3D™ API
- Resources

This tutorial focuses on released APIs. The tutorial assumes that you will be deploying on the Java 1.1 or Java 2 platform

Overview of the Java™ Platform

- Java programming language spec (grammar, keywords)
- Virtual machine spec (including bytecode)
- Core APIs
  - Specified for each Java 2 Platform edition: J2EE™, J2SE™, J2ME™
- Standard extension APIs
  - Examples include JMF and Java 3D API
- Related tools
  - Compiler, RMI registry, javadoc, etc.

Differences Between Core and Extension APIs

- All Java technology licensees must implement the Core API for a given edition
  - For the J2SE platform, this is java packages plus a few related such as javax.swing. Includes Java 2D API
- Licensees may optionally implement an Extension API
  - If they do, must implement entire specification
  - Provided in javax packages. Includes much of the Java Media APIs
- Historically, Media APIs start out as Extensions but may become Core

Java™ Platform Media Support

- Java 1.0 and 1.1 technology
  - Primitive core support for AWT-based 2D graphics, limited audio (applets only)
  - Java 1.1 supports JMF and Java Speech extensions
- Java 2 version 1.2 technology
  - Core now includes Java 2D, Java Sound Engine
  - Java 3D, JMF, Java Speech extensions
- Java 2 version 1.3 technology
  - Released early 2000
  - New Java 2D features, Java Sound API

Introduction to the Java™ Media APIs

- Java 2D
- Java Media Framework
- Java 3D
- Java Sound
- Java Speech
- Java Advanced Imaging
- Java Telephony
- Java Shared Data Toolkit

Note: "Java Media" properly refers to the entire set of Java Media APIs. "JMF" refers to a specific API, the Java Media Framework.

Introduction to the Java™ Media APIs

- Java 2D™ API
  - 2D graphics and image manipulation
  - Graphics capabilities extended in Graphics2D
- Java™ Media Framework API
  - Playback of synchronized media in 1.0 API
  - 2.0 API supports capture, streaming of audio and video
- Java 3D™ API
  - Object-based 3D graphics runtime
  - VRML 97 reference browser written with Java technology

Introduction to the Java™ Media APIs

- Java Sound API
  - Software sound processor and MIDI synthesizer
  - Sound engine (Java 2 SDK 1.2) and sound API (beginning in Java 2 SDK 1.3)
- Java Speech API
  - Speech recognition and synthesis
- Java Advanced Imaging API
  - Advanced 2D image processing
  - Implements many Java 2D API interfaces

Introduction to the Java™ Media APIs

- Java Telephony API
  - Computer and telephony integration
  - Portion specified as part of JavaPhone™ API and upcoming J2ME platform Mobile Information Device Profile
- Java Shared Data Toolkit
  - Development toolkit and API for adding collaborative features to Java technology-based applications
  - Objects share data via a Session object, JSDT URLs, and a JSDT registry
  - Version 2.0 is freely available from Sun

Availability of the Java™ Media APIs

API	*Type*	*Spec*	*FAQ*	*Reference Impl.*	*Mailing list*
Java 2D	Core Java 2	Yes (1.0 in v1.2; updates in v1.3)	Yes	Yes (included in Java 2)	Yes
JMF	Extension	Yes (2.0)	Yes	Yes (2.0)	Yes
Java 3D	Extension	Yes (1.1.3 released; 1.2 in Beta 2)	Yes	Yes (1.1.3 released)	Yes
Java Sound	Core Java 2 (Engine v1.2, API v1.3)	Yes (Java 2 v1.3 RC 3)	Yes	Yes (included in Java 2; also in JMF 2.0)	Yes

Availability of the Java™ Media APIs

API	*Type*	*Spec*	*FAQ*	*Reference Impl.*	*Mailing list*
Java Speech	Extension	Yes (API, 1.0; JSGF, 1.0; JSML, 0.5Beta)	Yes	No, but IBM, Lernout & Hauspie, other impls	Yes
Java Advanced Imaging	Extension	Yes (1.0 released; 1.1 in JCP)	Yes	Yes (1.0.2)	Yes
Java Telephony	Extension	Yes (1.3 released; 2.0 in JCP)	Yes	No, but various impls exist	Yes
Java Shared Data Toolkit	Extension	Yes (2.0 released)	Yes	Yes	Yes

Potentially Competing Technologies

- OpenGL
  - Solaris™, Win32, Linux, IRIX Java 3D API implementations are built on top of OpenGL
  - Java technology-to-OpenGL bindings widely available
  - Primarily competes with Java 3D API
- Fahrenheit
  - Microsoft initiative to extend and unify OpenGL and DirectX technologies; lost momentum with the withdrawal of SGI's support
  - Unclear how/when any of this might be available, if at all

Potentially Competing Technologies

- VRML
  - Complementary to Java 3D API
  - Sun joined the Web 3D (formerly VRML) Consortium in mid-1998
  - Sun co-founded the VRML-Java 3D Working Group and contributed much of VRML97 Java 3D API browser codebase
- QuickTime for Java™
  - Apple has released Java platform bindings to QuickTime multimedia architecture
  - Targets established QuickTime market
  - Primarily competes with JMF

Potentially Competing Technologies

- RealSystem G2
  - Includes an optional JMF binding for G2 players, plus some proprietary interfaces for the Java platform
  - Simultaneously competes with and leverages JMF
- Netshow Services
  - Provides Microsoft Media Player development APIs and tools
  - Server supports only Windows
  - Primary competition is JMF, though Media Player can be exposed in Java via JMF wrapper

Java 2D™ API

- Part of the Java Foundation Classes (JFC)
- Included in Java 2 Platform
  - All J2SE and J2EE platform implementations, including Java Plug-in technology, are required to support Java 2D API
- Treats all forms of 2D visual information (text, primitive shapes, polygons, Bezier curves, images, etc.) alike
- Enables consistent compositing, color manipulation, other 2D operations

Java 2D API Package Summary

- Java 2D API is specified in the following packages:
  - *java.awt*
  - *java.awt.color*
  - *java.awt.font*
  - *java.awt.geom*
  - *java.awt.image, java.awt.image.renderable*
  - *java.awt.print*
- Sun implementations of the Java 2 platform also provide support for JPEG in:
  - *com.sun.image.codec.jpeg*

Graphics2D: A Better Graphics Class

- Graphics2D is the rendering engine for Java 2D API
- Graphics2D extends abstract class Graphics, maintaining backwards compatibility
- Use Graphics2D by casting a Graphics reference to a Graphics2D reference

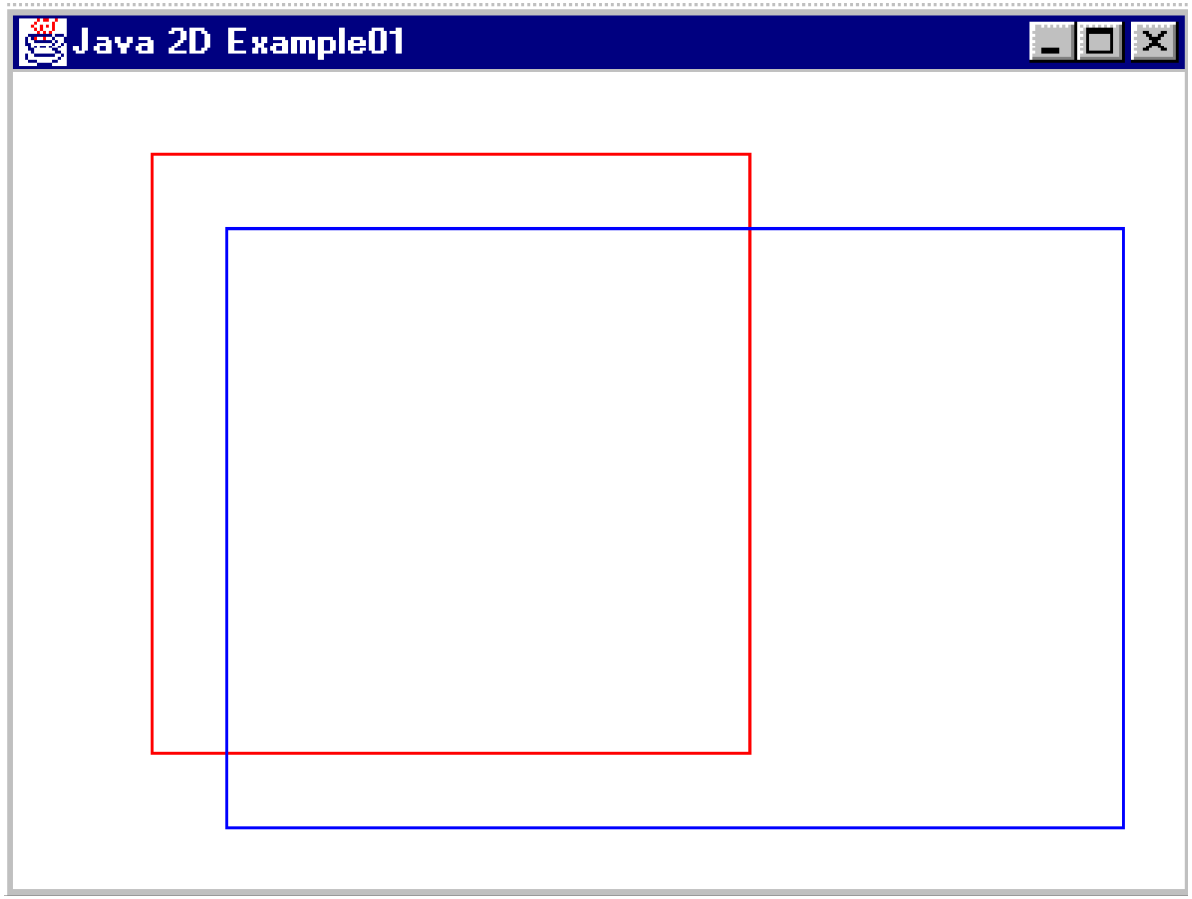
Java 2D™ API: Graphics2D

- Example01 illustrates getting a reference to Graphics2D
- Note that all of the examples in this presentation are available for download under the GNU General Public License
- Download the code in billday.jar from: **www.billday.com/Work**

Java 2D API: Example01

```
001   /**
002    * The paint method provides the real magic.  Here we
003    * cast the Graphics object to Graphics2D to illustrate
004    * that we may use the same old graphics capabilities with
005    * Graphics2D that we are used to using with Graphics.
006    **/
007   public void paint(Graphics g) {
008     //Here is how we used to draw a square with width
009     //of 200, height of 200, and starting at x=50, y=50.
010     g.setColor(Color.red);
011     g.drawRect(50,50,200,200);
012
013     //Let's set the Color to blue and then use the Graphics2D
014     //object to draw a rectangle, offset from the square.
015     //So far, we've not done anything using Graphics2D that
016     //we could not also do using Graphics.  (We are actually
017     //using Graphics2D methods inherited from Graphics.)
018     Graphics2D g2d = (Graphics2D)g;
019     g2d.setColor(Color.blue);
020     g2d.drawRect(75,75,300,200);
021   }
```

Java 2D API: Example01 Output



Java 2D API: Shapes and GeneralPaths

- Shapes are used to create arbitrarily shaped 2D graphics
- GeneralPaths are the most general implementation of Shape
- GeneralPath interiors are specified using winding rules
- All Shapes, including GeneralPaths, are manipulated using matrices in AffineTransforms

Java 2D API: Example02

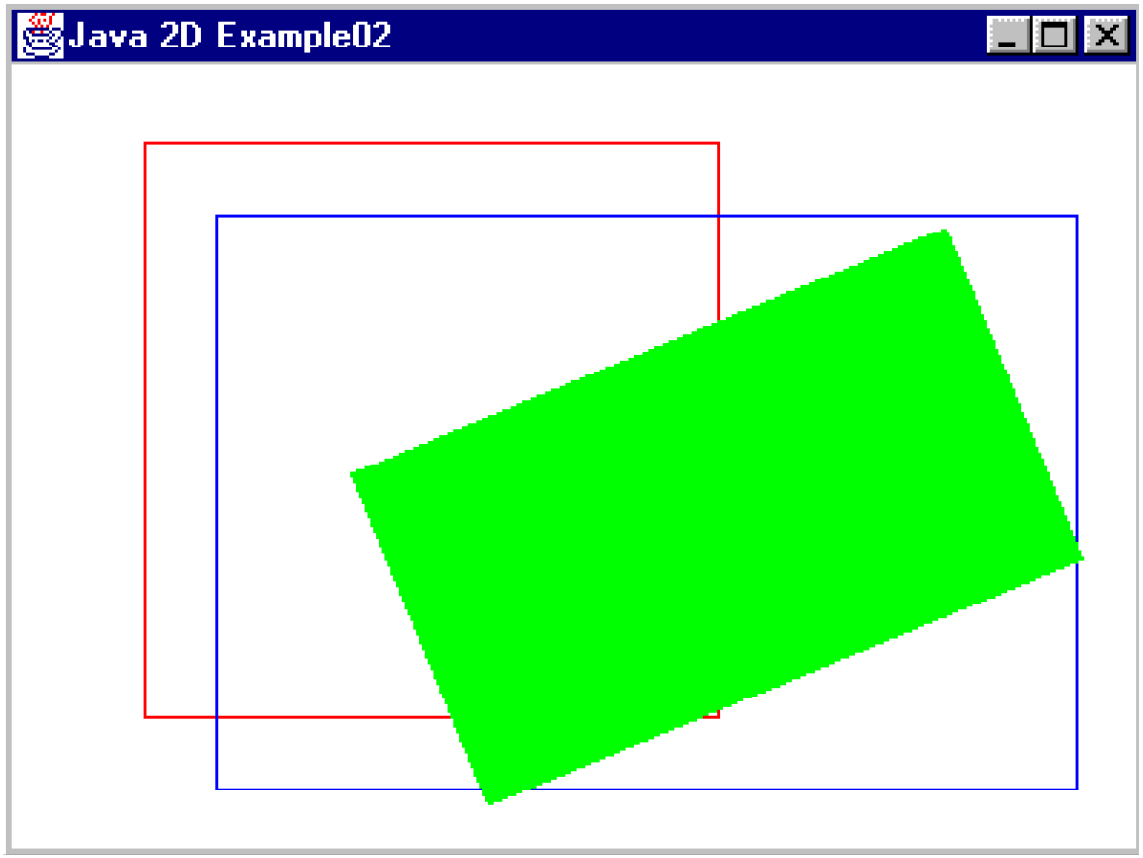
- Example02 illustrates GeneralPath, winding rules, and AffineTransform

```
021     //Now, let's draw another rectangle, but this time, let's
022     //use a GeneralPath to specify it segment by segment.
023     //Furthermore, we're going to translate and rotate this
024     //rectangle relative to the Device Space (and thus, to
025     //the first two quadrilaterals) using an AffineTransform.
026     //We also will change its color.
027     GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
028     path.moveTo(0.0f,0.0f);
029     path.lineTo(0.0f,125.0f);
030     path.lineTo(225.0f,125.0f);
031     path.lineTo(225.0f,0.0f);
032     path.closePath();
```

Java Java 2D API: Example02 (Cont.)

```
034     AffineTransform at = new AffineTransform();
035     at.setToRotation(-Math.PI/8.0);
036     g2d.transform(at);
037     at.setToTranslation(50.0f,200.0f);
038     g2d.transform(at);
039
040     g2d.setColor(Color.green);
041     g2d.fill(path);
042   }
```

Java 2D API: Example02 Output



Java 2D API: Curves, Text, and Antialiasing

- Example03 and Example04 introduce GeneralPath's lineto(), quadto(), and curveto() methods plus text support
- For more information on these topics, please refer to the billday.jar Java 2D examples and the Media Programming Java 2D series (URLs in Resources)
- Example05 builds on these by requesting antialiased rendering

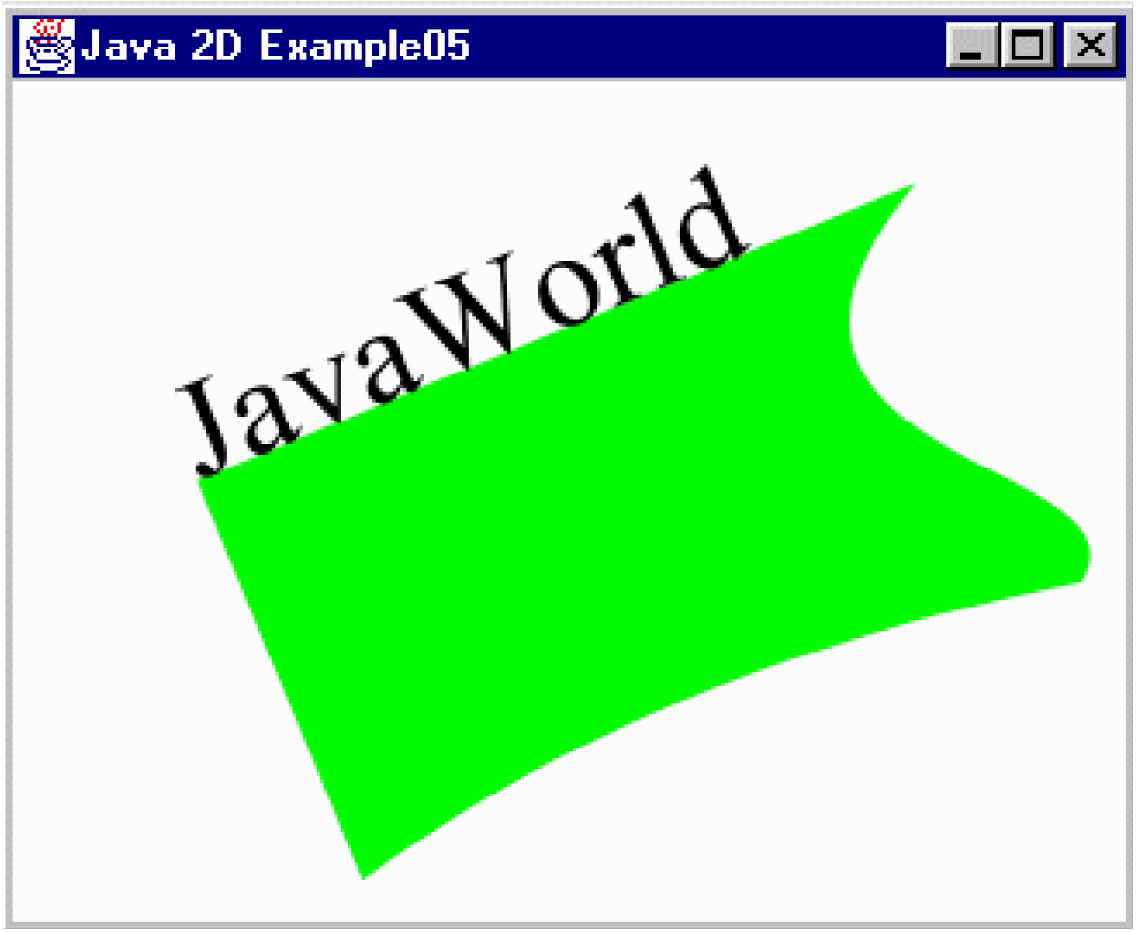
Java 2D API: Example05

```
007   public void paint(Graphics g) {
008     Graphics2D g2d = (Graphics2D) g;
009
010     //This time, we want to use anti-aliasing if possible
011     //to avoid the jagged edges that were so prominent in
012     //our last example.  We ask the Java 2D rendering
013     //engine (Graphics2D) to do this using a "rendering hint".
014     g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
015        RenderingHints.VALUE_ANTIALIAS_ON);
016
017     //We reuse our GeneralPath filled shape.  We translate
018     //and rotate this shape as we did before.
019     GeneralPath path = new GeneralPath(GeneralPath.WIND_EVEN_ODD);
020     path.moveTo(0.0f,0.0f);
021     path.lineTo(0.0f,125.0f);
022     path.quadTo(100.0f,100.0f,225.0f,125.0f);
023     path.curveTo(260.0f,100.0f,130.0f,50.0f,225.0f,0.0f);
024     path.closePath();
```

Java 2D API: Example05 (Cont.)

```
026     AffineTransform at = new AffineTransform();
027     at.setToRotation(-Math.PI/8.0);
028     g2d.transform(at);
029     at.setToTranslation(0.0f,150.0f);
030     g2d.transform(at);
031
032     g2d.setColor(Color.green);
033     g2d.fill(path);
034
035     //Now, let's use some of the Java font and text support.
036     //Note that you need to be sure you have the same fonts I
037     //use in the example (Times New Roman True Type) if you
038     //execute this example code.
039     Font exFont = new Font("TimesRoman",Font.PLAIN,40);
040
041     g2d.setFont(exFont);
042     g2d.setColor(Color.black);
043     g2d.drawString("JavaWorld",0.0f,0.0f);
044   }
```

Java 2D API: Example05 Output



Java 2D API: Clipping, images, and Compositing

- Please refer to my JavaWorld™ columns (see Resources) to learn more about:
  - Clipping paths in Example06
  - Basics of loading and manipulating images, Example07
  - Example08 demonstrates how to alpha composite, or blend, 2D graphics objects together

Java 2D API: Image Processing

- Java 2D API presents a new model for image processing, the buffered image model
- ImageDicer example makes use of the buffered image model to blur, sharpen, and otherwise manipulate user specified images

ImageDicer Source Image

Lady Agnew of Locknaw, by John Singer Sargent

Java 2D API: Color Inversion With ImageDicer

- Java 2D API provides lookup table support for use in color-related image manipulations
- Perform color inversion by inverting each of the red, blue, and green (RGB) color values for each pixel in an image

```
001 short[] invert = new short[256];
002 for (int i = 0; i < 256; i++)
003   invert[i] = (short)(255 - i);
004 BufferedImageOp invertOp = new LookupOp(
005     new ShortLookupTable(0, invert), null);
```

ImageDicer Inverted Image

Inverting all three RGB channels gives a *negative* image

Java 2D API: Snapshot Your Components

- Use Sun's JPEG support classes to save snapshots of Components
- The basic steps are:
  - Create a BufferedImage with the same dimensions as your Component
  - Draw the Component into the BufferedImage
  - Save the BufferedImage into a file using the JPEG package and FileOutputStream

Note: Requires the use of com.sun.image.codec.jpeg, which may not be available in all Java 2 runtimes

Java 2D API: SaveComponentAsJPEG

```
001   public void saveComponentAsJPEG(Component myComponent,
002                                   String filename) {
003     Dimension size = myComponent.getSize();
004     BufferedImage myImage =
005         new BufferedImage(size.width, size.height,
006         BufferedImage.TYPE_INT_RGB);
007     Graphics2D g2 = myImage.createGraphics();
008     myComponent.paint(g2);
009     try {
010       OutputStream out = new FileOutputStream(filename);
011       JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(out);
012       encoder.encode(myImage);
013       out.close();
014     }
015     catch (Exception e) { System.out.println(e); }
016   }
```

Java 2D API: New Features in Java 2 SDK 1.3

- Support for PNG image format
- Support for use of multiple monitors
- Various minor features and fixes

Java™ Media Framework API (JMF)

- JMF delivers and renders synchronized multimedia
- Specified and implemented in phases:
  - JMF 1.0 supports Java Media Players to play audio and video from both push and pull sources
  - JMF 2.0 adds Java Media Capture, support for capture from input devices, on-the-fly manipulations of media data, plugable codecs
  - Java Media Conference has been discussed for future work

JMF: Implementers

- Sun Microsystems
  - Java platform (all Java) version including Web server tools
  - Win32 and Solaris™ performance packs
- IBM
  - Co-developed JMF 2.0 with Sun
- RealNetworks
  - Built Win32 G2 player JMF 1.0 bindings
  - Previously promised support for all G2 player platforms

JMF: Previously Involved

- Intel
  - Implemented a JMF 1.0 ActiveMovie/ DirectShow-based Win32 player for standalone JDK™, Netscape and Microsoft browsers
  - Discontinued support and withdrew from JMF working group, late 1998
- SGI
  - Discontinued JMF 1.0Beta version of IRIX player in 1997

JMF: Supported Content Types

- Supported JMF 1.0 or 1.1 content types:
  - QuickTime, AVI video (Sun, Intel, SGI); MPEG-1, all but Sun Java version
  - WAV, AU audio (Sun, Intel, SGI); MIDI, all but Sun Java version
  - Sun supports MPEG-1 Layer 3 (MP3) audio in its JMF 2.0 implementation
  - RealAudio and RealVideo formats supported in G2 (Real)
  - H.261, H.263 video and G.723 audio low bitrate ITU protocols (Sun perf packs; Sun Java version supports audio-only)

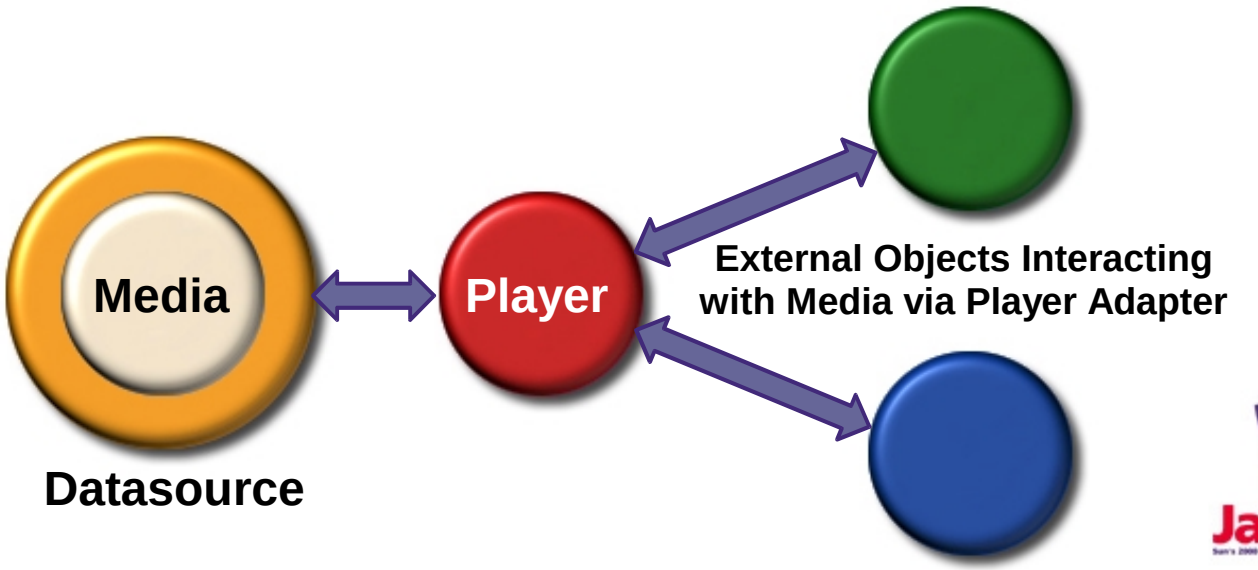
JMF: Supported Protocols

- Supported JMF 1.0 or 1.1 protocols:
  - HTTP, FILE (Sun, Intel, SGI)
  - FTP (Sun, Intel)
  - RTP receive-only (Sun)
  - RealNetworks protocols supported in G2 (Real)
- JMF 2.0 adds support for RTP send
  - RTP broadcasting plus the new JMF 2.0 capture support enable JMF-based audio and video servers

JMF: Player Basics

- Players extend MediaHandler and serve as an adapter for time-based media
- Media itself is encapsulated by a DataSource object

Player as an Adapter for Media



JMF: Player Basics

- In many implementations, the player is implemented in native (not Java technology) code
- Sun's entirely Java technology version implements the player using Java programming language code (no native methods)
- Sun's entirely Java technology player supports a subset of the content and protocol types supported by Sun performance packs

JMF 1.0 API: javax.media

- Players, content handling, and core synchronization are in javax.media package
  - Clock, Controller, Player interfaces
  - MediaHandler, MediaProxy interfaces
  - various Control and Listener interfaces
  - Manager, PackageManager classes
  - All events, errors, and exceptions to support state machine model
- Note: JMF events extend java.util.EventObject, consistent with standard Java 1.1 event mechanisms

JMF 1.0 API: javax.media.protocol

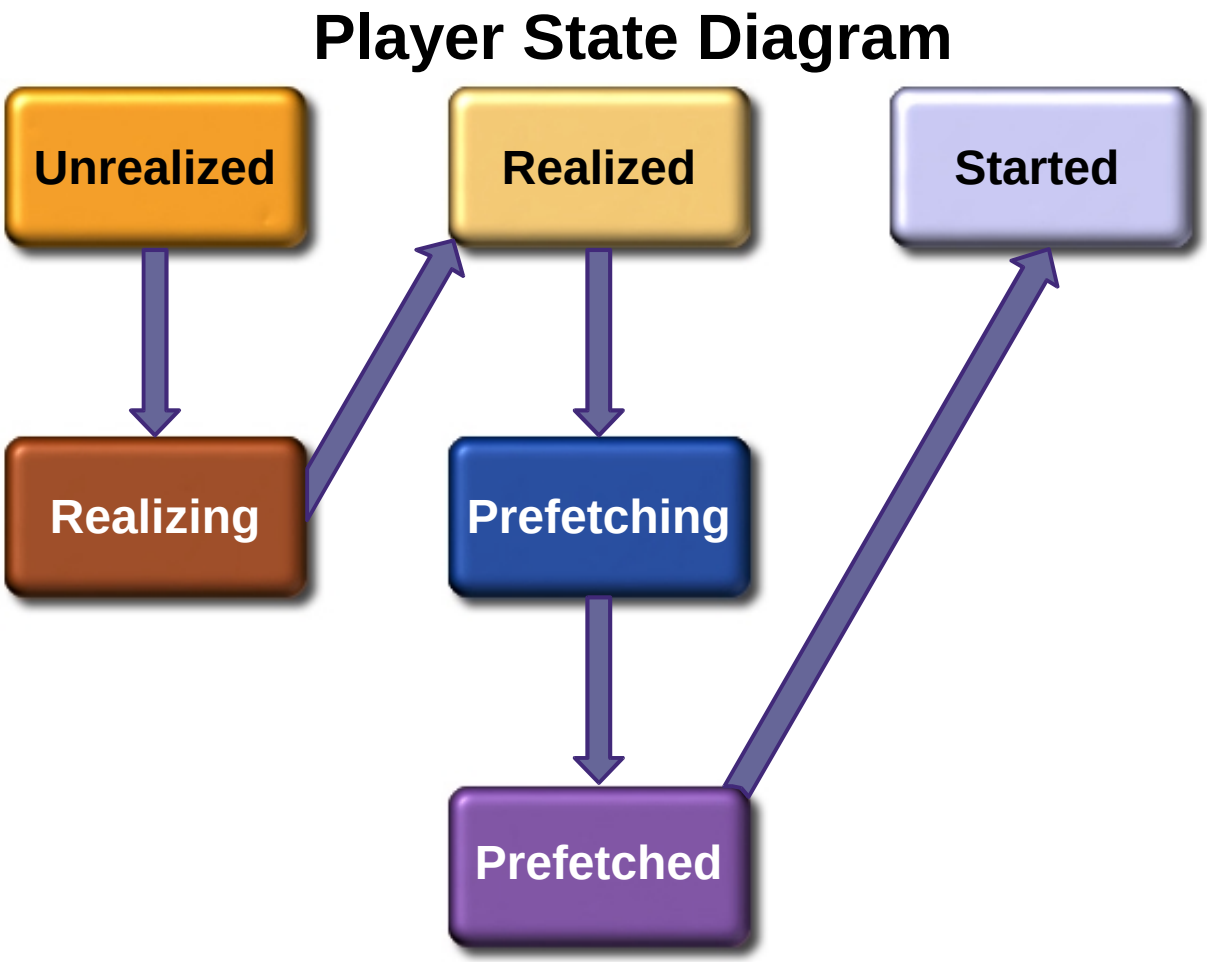
- DataSource and protocol resolution are specified in the javax.media.protocol package
  - Interfaces for source stream configuration and source controls
  - Classes to support push and pull DataSources
- Most JMF developers will not need to use this package (implement DataSources)
  - Manager automatically creates Player and DataSource and hooks the two together
  - Player methods indirectly use DataSource

JMF 1.0 API: Player States

- Players behave as state machines
- JMF specification details both legal and illegal state transitions and the corresponding Events and Exceptions

JMF 1.0 API: State Model

JMF 2.0 supports access to internals of JMF state model



JMF: JMF11Applet Example

- JMF11Applet illustrates several key JMF concepts
  - Use Manager to request a Player instance
  - Register ControllerListener for JMF callbacks
  - Catch RealizeCompleteEvent in controllerUpdate() method to finish setting up Player
  - Stop and properly deallocate Player to free up any exclusive resources

JMF11Applet

```
039 public class JMF11Applet extends Applet implements
040                                     ControllerListener {
041     private URL myURL = null;
042     private Player myPlayer = null;
043     private Component myVisual = null;
044     private Component myControls = null;
045     private Panel visualPanel = null;
046
047     /**
048      * Initialize JMF11Applet.  We lay out the interface and
049      * create our player in the init().
050     **/
051     public void init() {
052         super.init();
053
054         // Specify AWT Layout Manager.
055         setLayout (new BorderLayout());
056
057         // Load URL from the web page JMF11Applet is embedded in.
058         String asset = getParameter("ASSET");
```

JMF11Applet (Cont.)

```
059         // Check the URL and create a URL object to hold it.
060         if (asset.equals("")) {
061            //we haven't entered an asset in the applet.
062         } else {
063            try {
064               myURL = new URL(getDocumentBase(),asset);
065            } catch (MalformedURLException e) {
066               //We entered an incomplete asset or built incorrect URL.
067               //More robust applet should handle this gracefully.
068            }
069         }
```

JMF11Applet (Cont.)

```
070         try {
071            //Here's an interesting bit.  Manager is used to
072            //create the actual player for this URL.  We then
073            //add JMF11Applet as a ControllerListener for myPlayer.
074            //This lets us respond to RealizeCompleteEvents.
075            myPlayer = Manager.createPlayer(myURL);
076            myPlayer.addControllerListener(this);
077         } catch (IOException e) {
078            // Encountered some problem with I/O; exit.
079            System.out.println("I/O problem with player...exiting");
080            System.exit(1);
081         } catch (NoPlayerException e) {
082            // Unable to return a usable Player; exit.
083            System.out.println("No usable Player returned...exiting");
084            System.exit(1);
085         }
086     }
```

JMF11Applet (Cont.)

```
088     /**
089      * Override the default applet start method to call Player's
090      * realize().  This will first do the realization, which in turn
091      * triggers the bits of GUI building in the controllerUpdate()
092      * method.  We do not automatically start playback:  User needs
093      * to click on "play" button in our applet to start playing the
094      * media sample.
095     **/
096     public void start() {
097        myPlayer.realize();
098     }
099
100
101     /**
102      * Override default applet stop method to call myPlayer.stop()
103      * and myPlayer.deallocate() so we properly free up resources
104      * if someone exits this page in their browser.
105     **/
106     public void stop() {
107        myPlayer.stop();
108        myPlayer.deallocate();
109     }
```

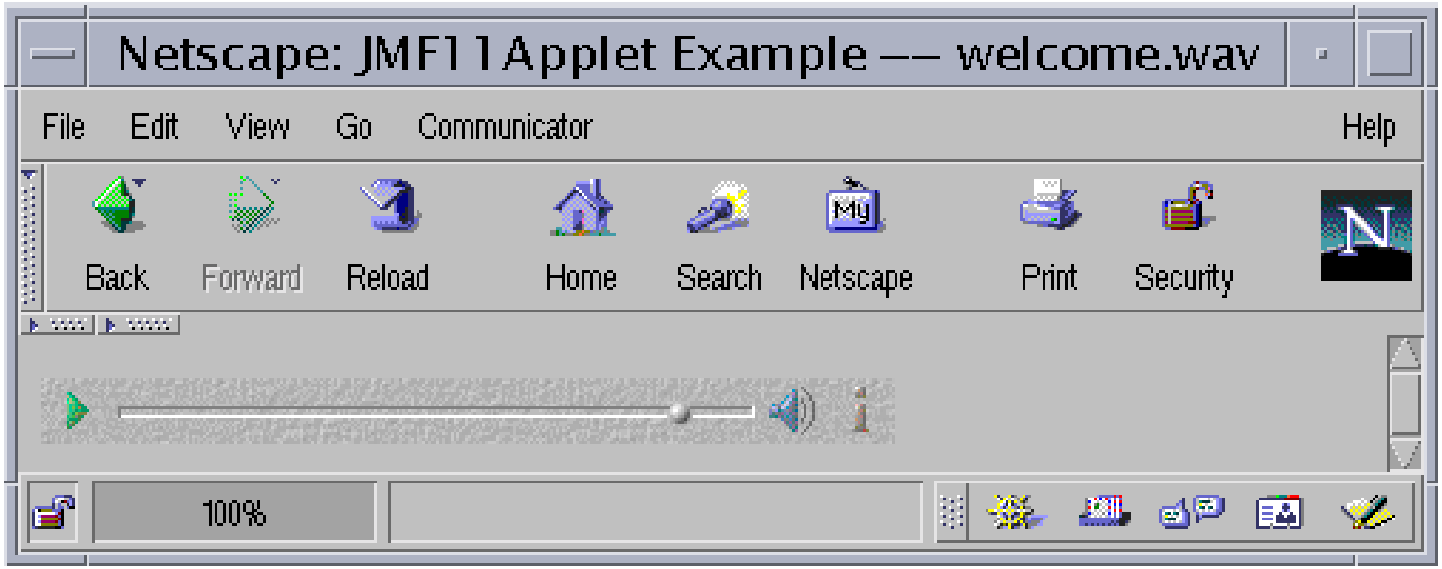
JMF11Applet (Cont.)

```
111     /**
112      * Since we must know when realize completes, we use
113      * controllerUpdate() to handle RealizeCompleteEvents.
114      * When we receive the RealizeCompleteEvent, we layout
115      * and display the video component and controls in our
116      * applet GUI.
117     **/
118     public void controllerUpdate(ControllerEvent event) {
119        if (event instanceof RealizeCompleteEvent) {
120           //System.out.println("Received RCE...");
121           // Now that we have a Realized player, we can get the
122           // VisualComponent and ControlPanelComponent and pack
123           // them into our applet.
124           myVisual = myPlayer.getVisualComponent();
125           if (myVisual != null) {
126              // In order to ensure that the VisualComponent
127              // is not resized by BorderLayout, I nest it
128              // within visualPanel using FlowLayout.
129              visualPanel = new Panel();
130              visualPanel.setLayout(new FlowLayout());
131              visualPanel.add(myVisual);
```

JMF11Applet (Cont.)

```
132               add(visualPanel,BorderLayout.CENTER);
133               //System.out.println("Added VisualComponent...");
134             }
135             myControls = myPlayer.getControlPanelComponent();
136             if (myControls != null) {
137                add(myControls,BorderLayout.SOUTH);
138                //System.out.println("Added controls...");
139             }
140             //invalidate();
141             validate();
142         }
143         // Else we simply consume the event.
144     }
145 }
```

JMF11Applet Output



- Sun JMF, Java platform-based player running on Solaris™ 7 software within Netscape Communicator™ 4.51

JMF11Applet Output

- The same Sun JMF, Java platform-based player, this time running on Win32 within Microsoft Internet Explorer 4.01

JMF11Applet Output

- Media information, loaded by clicking on “i” button in Sun player interface

JMF 1.0 API: Other Player Capabilities

- JMF allows the programmer to
  - Control multiple Players with a single Controller object, perhaps a Player itself
  - Integrate Players with other Java technology-based software. Same language, same tools, etc.
  - Integrate Players with JavaBeans™ architecture-based components for reusable streaming media components that can be quickly connected together using visual builder tools, allowing higher level programmers to take advantage of JMF and its flexibility

Java 3D™ API

- Java 3D API is an Extension API specifying a scene graph based 3D graphics runtime
- Java 3D API is optimized for display speed (interactive graphics and games) rather than image quality (render farms)
- Sun provides Win32 and Solaris™ operating environment implementations
- SGI has IRIX version; HP promises HP-UX implementation
- jFree-D on Linux and IRIX, others

Java 3D API: Requirements for Sun Implementation

- Use of Sun's implementation of Java 3D API requires Win32 or Solaris software and
  - Java 3D API itself, available from Sun's web site
  - OpenGL 1.1, bundled with WinNT 4.0, Win98, and Win95 OSR 2 (download from Microsoft for Win95 OSR 1)
  - Java 2 platform, available from Sun
- Sun's implementation of Java 3D API requires the Java 2 platform

Java 3D API: Requirements for jFree-D

- jFree-D will run on a variety of platforms, provided that they have
  - Java 1.1 platform (JDK™ 1.1.x)
  - OpenGL implementation (may use Mesa)
  - C compiler
  - GNU make (available for all Unix flavors as well as Win32)
- jFree-D uses the GL4Java Java platform-to-OpenGL binding in its implementation
- jFree-D is still in early availability and does not support all of Java 3D spec

Java 3D API: Strengths

- High level, object oriented view of 3D graphics
- Java 3D API is optimized for speed where possible using compiled branch groups, capability bits, etc.
- Large number of 3D loaders are available to import content into Java 3D runtime
  - Currently 20 loaders available, for formats varying from VRML to DXF to Protein DataBank

Java 3D API: Strengths

- Java 3D API supports exotic input and control devices
  - Data gloves
  - Wands
  - Heads-up displays (HUDs)
  - Virtual environments such as the NCSA C.A.V.E.
- Java 3D API specifies spatialized vector math support not available elsewhere in the Java platform

Java 3D API: Weaknesses

- Java 3D API hides rendering pipeline details from the developer, a “feature” with sometimes negative consequences
  - Java platform-to-OpenGL bindings may be a better choice for developers needing direct access to the rendering pipeline
- Java 3D API documentation is somewhat limited compared to OpenGL
- Java 3D API components are heavyweight, which complicates JFC/Swing-based GUI development

Java 3D API Package Summary

- Java 3D API is specified in:
  - javax.media.j3d
  - javax.vecmath
- Sun provides some very useful supporting classes and utilities under:
  - com.sun.j3d

Java 3D API: Scene Graph Basics

- Java 3D API programs create a tree-like structure of Nodes to represent the world to be rendered and rendering instructions
- Java 3D API scene graphs contain two major branches
  - Content branch describes the objects to render (how to draw them, color them, arrange them in 3D space, how they should behave)
  - View branch contains everything else (placement of user's view in 3D space, ability to move this view interactively, manipulations for stereo viewing, HUDs, etc.)

Java 3D API: Scene Graph Basics

- Java 3D API view branches are typically quite small compared to content branches
- View branches will often contain only a few nodes, while content branches may contain thousands for complicated 3D worlds
- Consequently, many Java 3D API optimizations focus on the content branch

Java 3D API: View Branch Example

- Example01 creates a very simple Java technology-based 3D world
- This world illustrates
  - Using the heavyweight Canvas3D component within a Frame container
  - Creating the view branch of the scene graph
  - Attaching a View to the view branch

Java 3D API: Example01, Creating Canvas3D

```
026     //Title our frame and set its size.
027     super("Java 3D Example01");
028     setSize(400,300);
029
030     //Here is our first Java 3D-specific code.  We add a
031     //Canvas3D to our Frame so that we can render our 3D
032     //graphics.  Java 3D requires a heavyweight component
033     //Canvas3D into which to render.
034     Canvas3D myCanvas3D = new Canvas3D(null);
035     add("Center",myCanvas3D);
036
037     //Turn on the visibility of our frame.
038     setVisible(true);
```

Java 3D API: Example01, Constructing the View

```
059   /**
060    * constructView() takes a Canvas3D reference and constructs
061    * a View to display in that Canvas3D.  It uses the default
062    * PhysicalBody and PhysicalEnvironment (both required to be
063    * set or else the 3D runtime will throw exceptions).  The
064    * returned View is used by constructViewBranch() to attach
065    * the scene graph's ViewPlatform to a Canvas3D for rendering.
066    *
067    * @see constructViewBranch(View)
068   **/
069   private View constructView(Canvas3D myCanvas3D) {
070     View myView = new View();
071     myView.addCanvas3D(myCanvas3D);
072     myView.setPhysicalBody(new PhysicalBody());
073     myView.setPhysicalEnvironment(new PhysicalEnvironment());
074     return(myView);
075   }
```

Java 3D API: Example01, Finishing the View

```
079     * constructViewBranch() takes as input a View which we
080     * attached to our Canvas3D in constructView().  It constructs
081     * a default view branch for the scene graph, attaches
082     * the View to the ViewPlatform, and returns a reference to
083     * our Locale for use by constructContentBranch()
084     * in creating content for our scene graph.
085     *
086     * @see constructView(Canvas3D)
087     * @see constructContentBranch(Locale)
088    **/
089    private Locale constructViewBranch(View myView) {
090
091      //First, we create the necessary coordinate systems
092      //(VirtualUniverse, Locale), container nodes
093      //(BranchGroup, TransformGroup), and platform which
094      //determines our viewing position and direction (ViewPlatform).
095      VirtualUniverse myUniverse = new VirtualUniverse();
096      Locale myLocale = new Locale(myUniverse);
097      BranchGroup myBranchGroup = new BranchGroup();
098      TransformGroup myTransformGroup = new TransformGroup();
099      ViewPlatform myViewPlatform = new ViewPlatform();
```

Java 3D API: Example01, Finishing the View (Cont.)

```
101     //Next, we insert the platform into the transform group,
102     //the transform group into the branch group, and the branch
103     //group into the locale's branch graph portion of the
104     //scene graph.
105     myTransformGroup.addChild(myViewPlatform);
106     myBranchGroup.addChild(myTransformGroup);
107     myLocale.addBranchGraph(myBranchGroup);
108
109     //Finally, we attach our view to the view platform and we
110     //return a reference to our new universe.  We are ready to
111     //render 3D content!
112     myView.attachViewPlatform(myViewPlatform);
113     return(myLocale);
114   }
```

Java 3D API: Example01 Output

- Calling no-op constructContentBranch() turns the Java 3D renderer on (sets the scene graph to be live), which renders empty universe

Java 3D API: Content Branch Example

- Example02 adds a more interesting body to the constructContentBranch() method of our previous example
  - Uses the heavyweight Canvas3D component within a Frame container
  - Creates the view branch of the scene graph
  - Attaches a View to the view branch

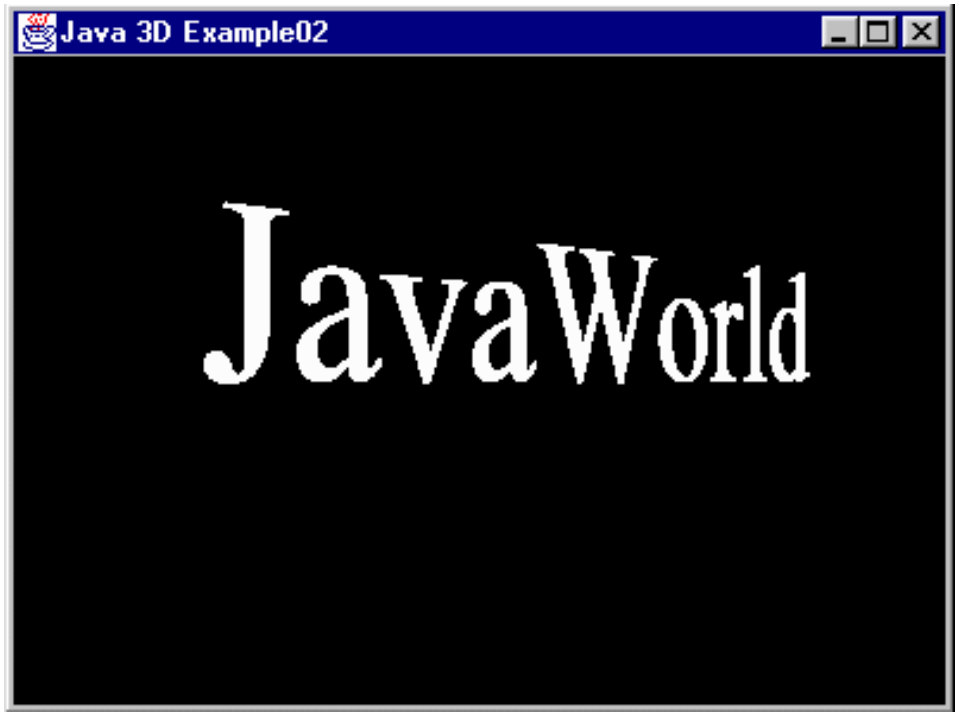
Java 3D API: Example02

```
014   private void constructContentBranch(Locale myLocale) {
015     //We first create a regular 2D font, then from that a
016     //3D font, 3D text, and 3D shape, in succession.  We use
017     //the default constructors for FontExtrusion and Appearance.
018     Font myFont = new Font("TimesRoman",Font.PLAIN,10);
019     Font3D myFont3D = new Font3D(myFont,new FontExtrusion());
020     Text3D myText3D = new Text3D(myFont3D, "JavaWorld");
021     Shape3D myShape3D = new Shape3D(myText3D, new Appearance());
022
023     //We created a new branch group and transform group to hold
024     //our content.  This time when we create the transform group,
025     //however, we pass in a Transform3D to the transform group's
026     //constructor.  This 3D transform has been manipulated
027     //to perform the transformations we desire, which results
028     //in those manipulations being carried out on all children
029     //of the given transform group, in this case, our content.
030     BranchGroup contentBranchGroup = new BranchGroup();
```

Java 3D API: Example02 (Cont.)

```
031     Transform3D myTransform3D = new Transform3D();
032     myTransform3D.setTranslation(new Vector3f(-1.0f,0.0f,-4.0f));
033     myTransform3D.setScale(0.1);
034     Transform3D tempTransform3D = new Transform3D();
035     tempTransform3D.rotY(Math.PI/4.0d);
036     myTransform3D.mul(tempTransform3D);
037     TransformGroup contentTransformGroup = new
038                             TransformGroup(myTransform3D);
039
040     //We add our child nodes and insert the branch group into
041     //the live scene graph.  This results in Java 3D rendering
042     //the content.
043     contentTransformGroup.addChild(myShape3D);
044     contentBranchGroup.addChild(contentTransformGroup);
045     myLocale.addBranchGraph(contentBranchGroup);
046   }
```

Java 3D API: Example02 Output



- Our universe is empty no more!

Java 3D API: Utilities From Sun Can Make Code Simpler

- You may have looked at the set-up code in the first two examples and wondered “Why do we have to make so many redundant calls each time we use Java 3D API?”
- We do not, if we are willing to use Sun utility classes (or write our own)
- Example03 makes use of Sun’s SimpleUniverse and ColorCube

Java 3D API: Example03, Using SimpleUniverse

```
001     //Now that we have our Frame and Canvas3D, we are ready
002     //to start building our scene graph.  We need to construct
003     //both a view branch and a content branch.  Using Sun's
004     //SimpleUniverse utility, we can combine the View and
005     //view branch construction into one simple step.
006     SimpleUniverse myUniverse = new SimpleUniverse(myCanvas3D);
007     BranchGroup contentBranchGroup = constructContentBranch();
008     myUniverse.addBranchGraph(contentBranchGroup);
009   }
```

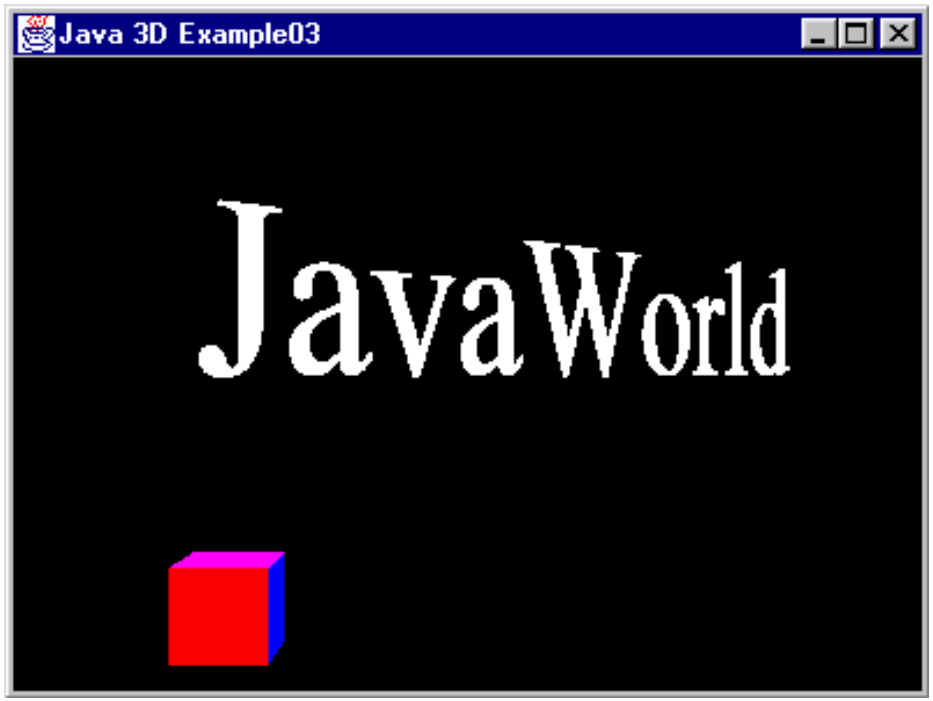
Java 3D API: Example03, ColorCube

```
012     * constructContentBranch() is where we specify the 3D graphics
013     * content to be rendered.  We return the content branch group
014     * for use with our SimpleUniverse.  We also demonstrate the
015     * use of com.sun.j3d.utils.geometry.ColorCube to build more
016     * complicated 3D shapes.
017     *
018    **/
019    private BranchGroup constructContentBranch() {
020      Font myFont = new Font("TimesRoman",Font.PLAIN,10);
021      Font3D myFont3D = new Font3D(myFont,new FontExtrusion());
022      Text3D myText3D = new Text3D(myFont3D, "Gamasutra");
023      Shape3D myShape3D = new Shape3D(myText3D, new Appearance());
024      Shape3D myCube = new ColorCube();
025
026      BranchGroup contentBranchGroup = new BranchGroup();
027      Transform3D myTransform3D = new Transform3D();
028      myTransform3D.setTranslation(new Vector3f(-1.0f,0.0f,-4.0f));
029      myTransform3D.setScale(0.1);
030      Transform3D tempTransform3D = new Transform3D();
031      tempTransform3D.rotY(Math.PI/4.0d);
032      myTransform3D.mul(tempTransform3D);
```

Java 3D API: Example03, ColorCube (Cont.)

```
033     TransformGroup contentTransformGroup = new
034                              TransformGroup(myTransform3D);
035
036     contentTransformGroup.addChild(myShape3D);
037     contentBranchGroup.addChild(contentTransformGroup);
038
039     myTransform3D.setIdentity();
040     myTransform3D.setTranslation(new Vector3f(-0.5f,-0.5f,-2.3f));
041     myTransform3D.setScale(0.1);
042     TransformGroup cubeTransformGroup = new
043                            TransformGroup(myTransform3D);
044
045     cubeTransformGroup.addChild(myCube);
046     contentBranchGroup.addChild(cubeTransformGroup);
047
048     return(contentBranchGroup);
049   }
```

Java 3D API: Example03 Output



- Note the multi-colored ColorCube, and that it is offset in space from the text

Java 3D API: Behaviors

- In Java 3D API, behaviors are scheduled when the view platform crosses the stimulus bounds, a region of space defined by the programmer
- Bounds are used by the Java 3D runtime to avoid computations for non-visible or inaudible Nodes
- Both sounds and behaviors have bounds
- Example04 illustrates basic behavior, adding rotation to previous examples

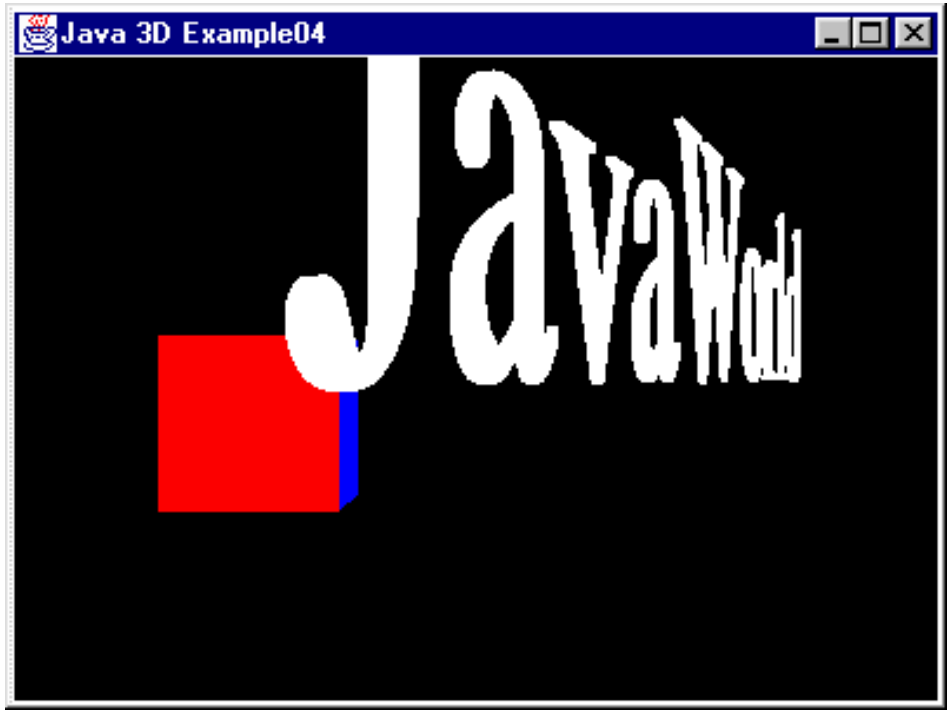
Java 3D API: Example04

```
002     * constructContentBranch() is where we specify the 3D graphics
003     * content to be rendered.  We return the content branch group
004     * for use with our SimpleUniverse.  We have added a RotationInterpolator
005     * to Example03 so that in this case, our "JavaWorld" text rotates
006     * about the origin.  We have also removed the scaling and static
007     * rotation from the text, and the scaling from our ColorCube.
008    **/
009    private BranchGroup constructContentBranch() {
010      Font myFont = new Font("TimesRoman",Font.PLAIN,10);
011      Font3D myFont3D = new Font3D(myFont,new FontExtrusion());
012      Text3D myText3D = new Text3D(myFont3D, "JavaWorld");
013      Shape3D myShape3D = new Shape3D(myText3D, new Appearance());
014      Shape3D myCube = new ColorCube();
015
016      BranchGroup contentBranchGroup = new BranchGroup();
017      Transform3D myTransform3D = new Transform3D();
018      TransformGroup contentTransformGroup = new TransformGroup(myTransform3D);
019      contentTransformGroup.addChild(myShape3D);
020
021      Alpha myAlpha = new Alpha();
022      myAlpha.setIncreasingAlphaDuration(10000);
```

Java 3D API: Example04 (Cont.)

```
023     myAlpha.setLoopCount(-1);
024     RotationInterpolator myRotater =
025                         new RotationInterpolator(myAlpha,contentTransformGroup);
026     myRotater.setAxisOfRotation(myTransform3D);
027     myRotater.setMinimumAngle(0.0f);
028     myRotater.setMaximumAngle((float)(Math.PI*2.0));
029     BoundingSphere myBounds = new BoundingSphere();
030     myRotater.setSchedulingBounds(myBounds);
031     contentTransformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
032     contentTransformGroup.addChild(myRotater);
033
034     contentBranchGroup.addChild(contentTransformGroup);
035
036     myTransform3D.setTranslation(new Vector3f(-0.5f,-0.5f,-2.3f));
037     TransformGroup cubeTransformGroup = new TransformGroup(myTransform3D);
038     cubeTransformGroup.addChild(myCube);
039     contentBranchGroup.addChild(cubeTransformGroup);
040
041     return(contentBranchGroup);
042   }
```

Java 3D API: Example04 Output

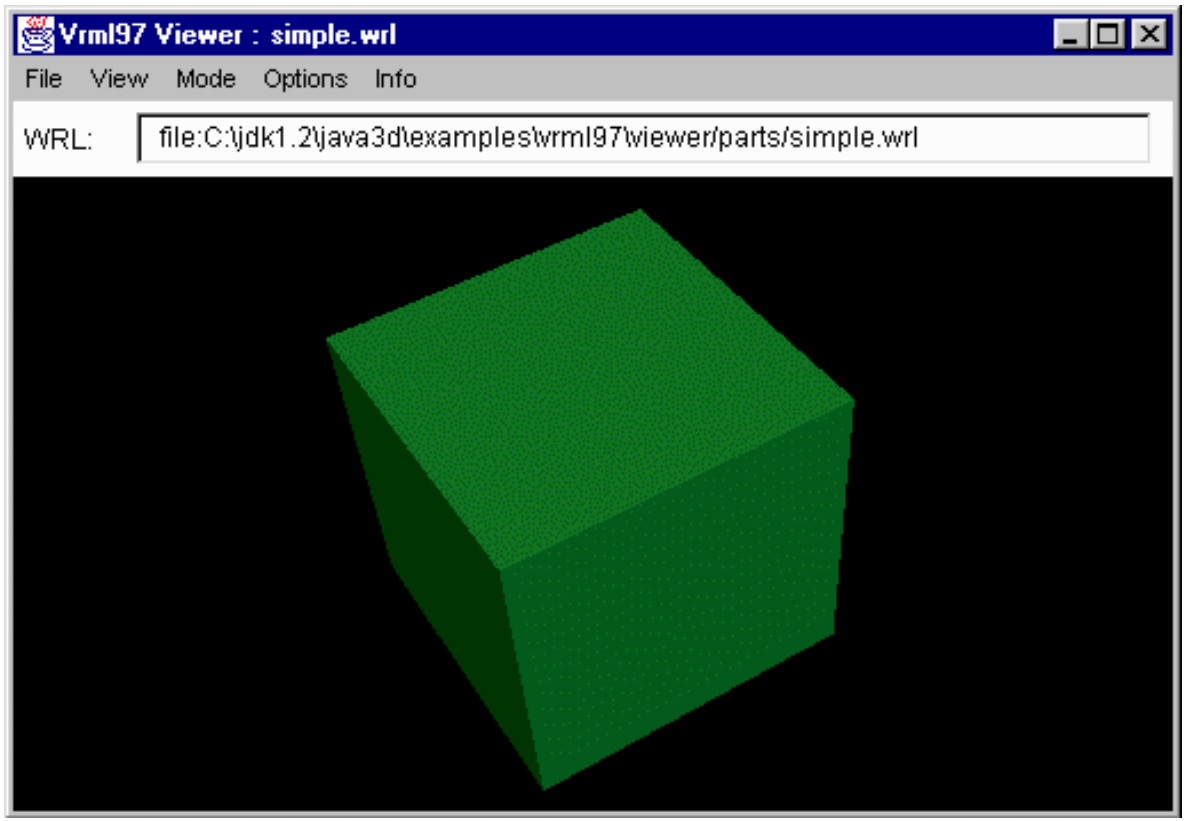


- The text rotates around in 3D space

Java 3D API: VRML97 Browser

- In addition to the various Java 3D API file loaders, Sun and the VRML-Java 3D Working Group have made a VRML97 browser available
- The browser is written in all Java technology, and so should run anywhere that the Java 3D API runs

Java 3D API: VRML Browser



- VRML97 Viewer, with the included sample.wrl VRML file loaded

Resources

- Sun's Java Media APIs page links to individual pages for each API: **java.sun.com/products/java-media**
- Media Programming resources, including all example code in this presentation, are available from: **www.billday.com/Work**
- Java 3D Loader Archive: **www.billday.com/Java3DArchive**

Copyright Statement

Software License

All software provided for download herein is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

The software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

See the GNU General Public License for more details:
www.fsf.org/copyleft/gpl.html

Speaker Bio

Bill Day is a Technology Evangelist at Sun Microsystems. Bills writes a technical column for JavaWorld™ magazine as well as *Computing Careers* columns which appear in ITWorld.com newsletters, Dr. Dobb's Journal Software Careers, CareerMag.com, and PCINews.com. Bill moderates JavaWorld's *Device Programming* discussion hosted by ITWorld.com and speaks frequently on multimedia and mobile device programming.

More information is available from Bill's web site:
www.billday.com

JavaOne℠

Sun's 2000 Worldwide Java Developer Conference℠