Chapter 16.5. Augmented Reality with NyARToolkit

[**Note**: this draft differs quite significantly from the previous version. I've removed the reliance on JMF, replacing it with JavaCV's FrameGrabber. Since NyARToolkit on the PC utilizes JMF for image capturing, this has meant some large changes to my code. I've not changed any of the NyARToolkit API, but bypass most of its optional utility classes.]

Augmented Reality (AR) enhances a user's view of the real world with computer-generated imagery, rendered quickly enough so that the added content can be changed/updated as the physical view changes.

AR started its rise with the development of Head Mounted Displays (HMDs) which superimpose images over the user's field of vision. Tracking sensors allow these graphics to be modified in response to the user's head movement. But AR received its biggest boost with the appearance of mobile devices containing cameras, GPS, accelerometers, wireless internet connection, and more. Applications are starting to appear which allow you to simply point a camera phone at something (e.g. a shop window, a theatre) and the on-screen display will be augmented with information (e.g. sales offers, discounted tickets), customized to your interests, at that time and place. Two popular examples are Layar (http://layar.com/) and Wikitude (http://www.wikitude.org/).

ARToolkit is probably the most widely used AR library (http://www.hitl.washington.edu/artoolkit/): it identifies predefined physical markers in a supplied video stream, and overlays those markers with 3D models. One of its many 'children' is NyARToolkit – a OOP port aimed at Java 3D, Processing, Android, C#, and C++ (http://nyatla.jp/nyartoolkit/wp/?page_id=198); I'll be using the Java 3D version in this chapter.

Figure 1 illustrates how the toolkit can be utilized.

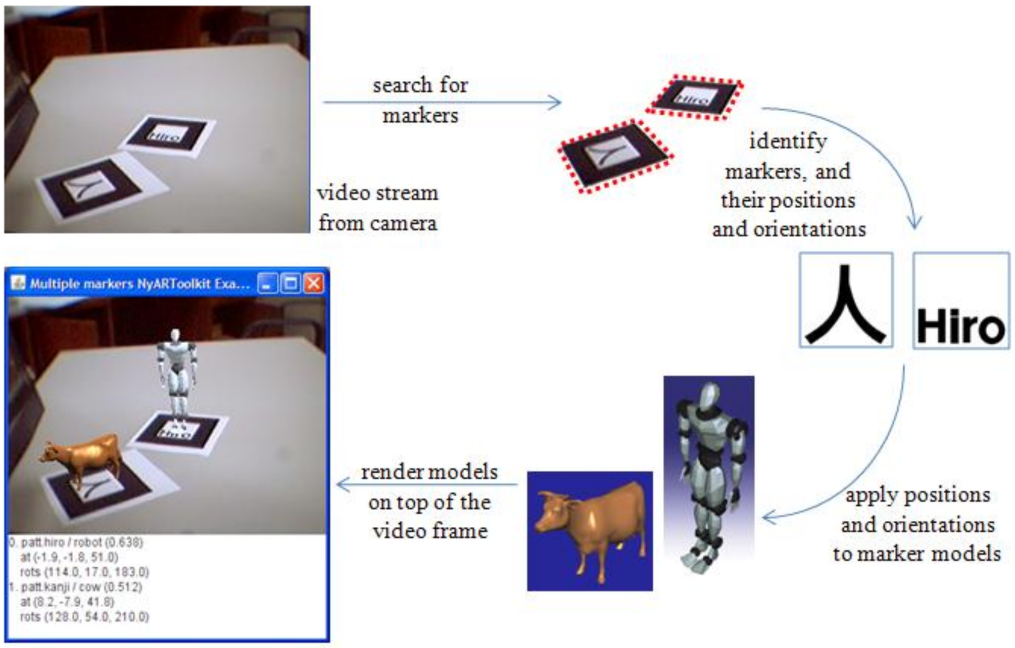
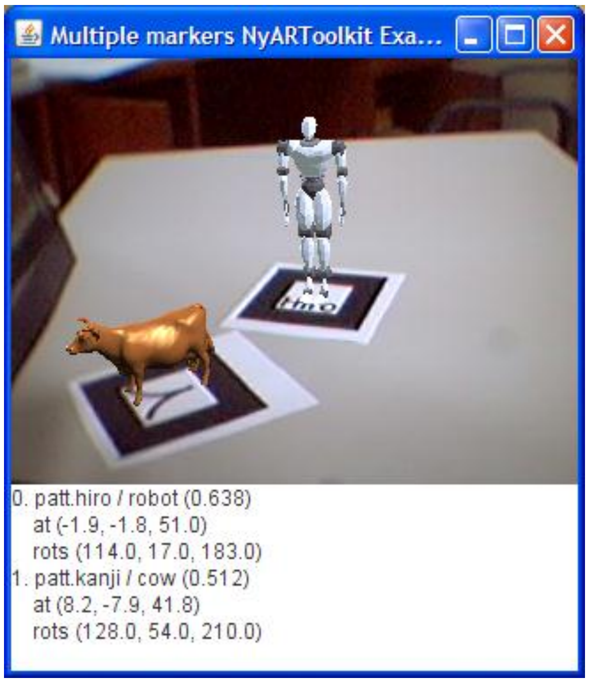

search for markers
video stream from camera
identify markers, and their positions and orientations
人
Hiro
apply positions and orientations to marker models
render models on top of the video frame
Multiple markers NyARToolkit Exa...

Figure 1. Using NyARToolkit.

A camera streams video into the NyARToolkit application (MultiNyAR.java in this chapter), which searches for markers in each frame. The markers (squares with thick black borders) are identified, and their orientation relative to the image frame is calculated. 3D models associated with the markers are added to the frame, after being transformed so they appear to be standing on top of their markers. Figure 2 shows the MultiNyAR GUI in more detail.

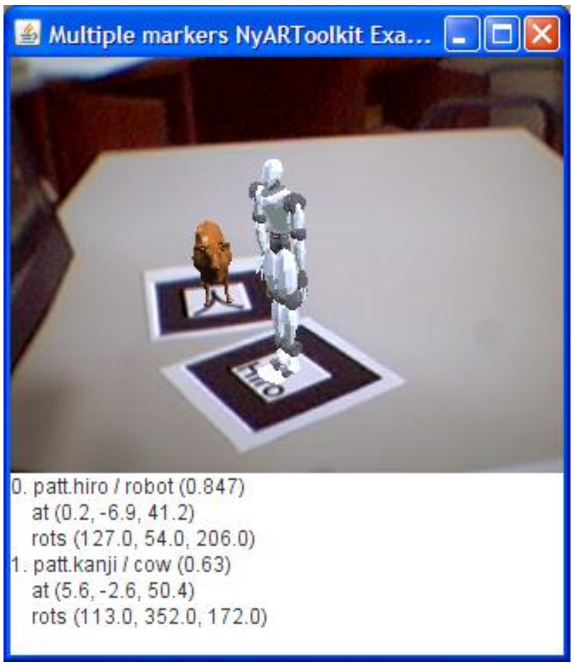


```
0. patt.hiro / robot (0.638)
   at (-1.9, -1.8, 51.0)
   rots (114.0, 17.0, 183.0)
1. patt.kanji / cow (0.512)
   at (8.2, -7.9, 41.8)
   rots (128.0, 54.0, 210.0)
```

Figure 2. The MultiNyAR GUI.

The application consists of a panel showing the augmented video stream, and a text area giving extra details about the markers and models. For example, the robot model in Figure 2 is positioned at (-1.9, -1.8, 51.0) and the cow at (8.2, -7.9, 41.8). The positive z-axis points into the scene (as explained later), which means that the robot is 'behind' the cow.

Figure 3 shows the same scene after the markers have been moved around.



```
0. patt.hiro / robot (0.847)
   at (0.2, -6.9, 41.2)
   rots (127.0, 54.0, 206.0)
1. patt.kanji / cow (0.63)
   at (5.6, -2.6, 50.4)
   rots (113.0, 352.0, 172.0)
```

Figure 3. A Changed MultiNyAR Scene.

The model's orientation and position have changed to correspond to the new locations of the markers. For instance, the robot's z-axis position in Figure 3 is now 41.2, while the cow's is 50.5, indicating that the robot is in the foreground.

To summarize: the MultiNyAR.java program described in this chapter explains how to:

- utilize multiple markers in Java 3D;
- place arbitrary 3D models on top of markers;
- extract position, orientation, and confidence information from NyARToolkit;
- reduce model 'shaking' and handle undetected markers.

My code also replaces JMF with JavaCV's FrameGrabber. This only affects the capturing parts of the code; the core elements of NyARToolkit are untouched.

1. Installing NyARToolkit

The first hurdle for writing AR applications in Java is the installation of a large number of libraries. You'll need JavaCV and OpenCV to handle the video coming from the camera (but I'll assume you've already installed those for the earlier chapters). I also utilize Java 3D and the NCSA Portfolio library, a useful collection of Java 3D model loaders. Java 3D installers for a variety of platforms can be obtained from https://java3d.java.net/binary-builds.html. NCSA Portfolio was developed by the now-defunct NCSA Java 3D Group, in a project led by Steve Pietrowicz and Chris Heistad. It can be downloaded from
http://fivedots.coe.psu.ac.th/~ad/jg/code/portfolio.zip

The top-level English language page for NyARToolkit is at http://nyatla.jp/nyartoolkit/wp/?page_id=198, which leads to different versions of the library for Java, Android, and others. Source code is stored at http://sourceforge.jp/projects/nyartoolkit/ I downloaded "NyARToolkit for Java - NyARToolkit Core" v4.1.1 as a zipped file called NyARToolkit-4.1.1.zip.

1.1. Compiling NyARToolkit

Unfortunately, the unzipped NyARToolkit does not include a ready-to-use JAR executable, but it's fairly simple to create one.

1. The unzipped NyARToolkit-4.1.1\lib\src\ directory should be copied to a new location (e.g. into C:\NyARToolkit\).
2. The unzipped NyARToolkit-4.1.1\utils\java3d\src\jp\nyatla\nyartoolkit\java3d\utils\ directory contains Java 3D utilities. Copy only J3dNyARParam.java to a new java3d\utils\ directory in C:\NyARToolkit\src\jp\nyatla\nyartoolkit\.
3. The unzipped NyARToolkit-4.1.1\utils\j2se\src\jp\nyatla\nyartoolkit\utils\j2se\ directory contains J2SE utilities. Copy only the three files into a new utils\j2se\ directory in C:\NyARToolkit\src\jp\nyatla\nyartoolkit\.

This C:\NyARToolkit subdirectory structure will now look something like Figure 4.

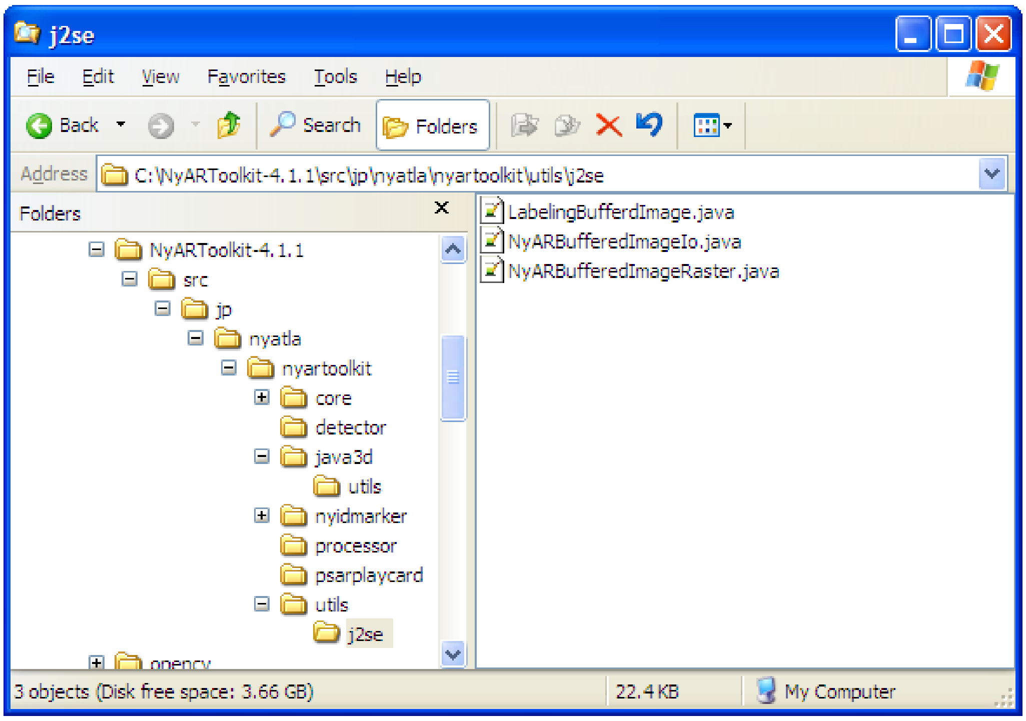

j2se
File Edit View Favorites Tools Help
Back Search Folders
Address C:\NyARToolkit-4.1.1\src\jp\nyatla\nyartoolkit\utils\j2se
Folders
NyARToolkit-4.1.1
src
jp
nyatla
nyartoolkit
core
detector
java3d
utils
nyidmarker
processor
psarplaycard
utils
j2se
opencv
LabelingBufferdImage.java
NyARBufferedImageIo.java
NyARBufferedImageRaster.java
3 objects (Disk free space: 3.66 GB)
22.4 KB
My Computer

Figure 4. The Modified nyartoolkit\ Directory.

All the Java files in C:\NyARToolkit\src and below must now be compiled. The easiest way of doing this is to create a text file listing all the Java files, using:

```
> dir /b /s *.java > sources.txt
```

If this command is called inside C:\NyARToolkit\src\, then sources.txt will end up containing:

```
C:\NyARToolkit-4.1.1\src\jp\nyatla\nyartoolkit\core\INyARDisposable.java
C:\NyARToolkit-4.1.1\src\jp\nyatla\nyartoolkit\core\NyARCode.java
C:\NyARToolkit-4.1.1\src\jp\nyatla\nyartoolkit\core\NyARException.java
      :
```

On UNIX, you could try `find src -name \*.java -print > sources.txt`

Then use a text editor to remove the "C:\NyARToolkit\src\" prefixes, leaving:

```
jp\nyatla\nyartoolkit\core\INyARDisposable.java
jp\nyatla\nyartoolkit\core\NyARCode.java
jp\nyatla\nyartoolkit\core\NyARException.java
      :
```

Still in C:\NyARToolkit\src\, javac.exe can be called with the sources.txt file as an argument:

```
>  javac -encoding ISO-8859-1 @sources.txt
```

Note the use of "@". The javac call requires the "-encoding" option so the character set is extended to deal with the Japanese text comments in the files. Ignore the warning about NyARLinkList.java due to the lack of a generic declaration.

When javac finishes, a quick look in the code directories below src\ will show you that the Java files have been compiled.

Still in src\, the jp\ directory containing all the compiled Java code can be packaged into a JAR file:

```
>  jar cf NyARToolkit.jar jp
```

The resulting NyARToolkit.jar can now be moved to your test directory, and the C:\NyARToolkit\src\ directory (and subdirectories) deleted (if you wish).

1.2. Testing NyARToolkit

The NyARToolkit download includes a Java 3D example called NyARJava3D.java, but unfortunately it relies on JMF to capture images from the webcam. I've recoded it as the application BoxNyAR, using the same NyARToolkit processing, but with images captured via JavaCV's FrameGrabber.

You must have your webcam pointing at a print-out of the "Hiro" marker (Figure 5). Its PDF file can be found in the NyARToolkit download in Data/pattHiro.pdf.

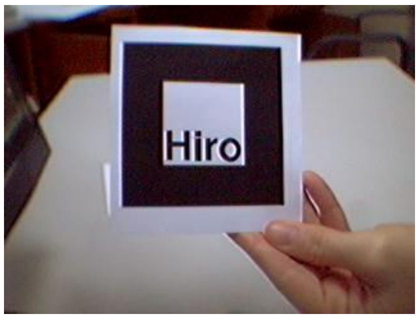


Figure 5. The Hiro Marker.

If everything is working fine, BoxNyAR creates a small window displaying a Java 3D colored cube resting on top of the marker, as in Figure 6.

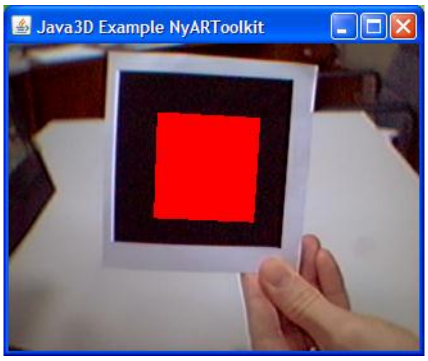


Figure 6. A Colored Cube on the Hiro Marker.

Turning the marker, causes the cube to move as well, so it stays on top of the "Hiro" text, as in Figure 7.

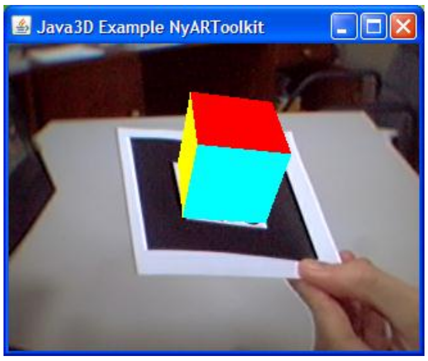


Figure 7. Moving the Marker and Cube.

BoxNyAR has the benefit of being relatively short, and relatively easy to understand. However, it lacks several features needed for more useful AR programming. For example, it utilizes the Java 3D built-in ColorCube class which is a rather boring 3D model. More importantly, it employs the NyARToolkit NyARSingleDetectMarker class which can only detect a single marker in an image.

1.3. More Help with NyARToolkit

The Japanese documentation for NyARToolkit is at http://sixwish.jp/Nyartoolkit/, and a English version of the installation steps can be found at http://sixwish.jp/Nyartoolkit/Java/section01.en/. This includes details for installing Java, Java 3D, JOGL, and JMF, with numerous screenshots (although they are in Japanese).

There’s a English language forum for NyARToolkit at http://sourceforge.jp/projects/nyartoolkit/forums/

The best place for an overview of NyARToolkit is at the parent ARToolkit site (http://www.hitl.washington.edu/artoolkit/) which includes documentation and publication sections. The documentation is particularly helpful for explaining how to calibrate the camera, create new markers, and deal with multiple markers. The ARToolkit download comes with several utilities which can help with calibration and marker creation.

2. Introducing Java 3D

A few years ago, I wrote a book that spent hundreds of pages explaining the uses of Java 3D for games programming (*Killer Game Programming in Java*), so it's not realistic to try to explain all the intricacies of the API in a few pages. Instead this section gives a non-technical overview of Java 3D's main idea (the scene graph), which is enough to understand the general structure of the MultiNyAR application (which is based around the manipulation of a scene graph).

For more details, you'll need to turn elsewhere. If you don't want to buy the oh-so-excellent *Killer Game Programming in Java* text, then you can read early drafts of the chapters at its website, http://fivedots.coe.psu.ac.th/~ad/jg/. Lots of code examples can be found there, including the HelloUniverse example described below.

2.1. The Scene Graph

The scene graph makes 3D programming much easier for novices (and even for experienced programmers) because it emphasizes scene design rather than rendering, by abstracting away from the graphics pipeline. The pipeline is represented by a tree-like structure built from nodes representing 3D models, lights, sounds, the background, the camera, and many other scene elements.

The nodes are typed, the main division being between Group and Leaf nodes. A Group node is one with child nodes, grouping the children so that operations such as translations, rotations, and scaling can be applied en masse. Leaf nodes are the leaves of the graph (did you guess that?), which often represent the visible things in the scene, such as 3D shapes, but may also be non-tangible entities, such as lighting and sounds. Additionally, a Leaf node may have node components, specifying color, reflectivity, and other attributes of the Leaf.

The scene graph can contain behaviors – nodes holding code that affects other nodes in the graph at run time. Typical behavior nodes move shapes, detect and respond to shape collisions, and cycle lighting from day to night.

The term scene graph is used, rather than scene tree, because it's possible for nodes to be shared (i.e. have more than one parent).

Before looking at a real Java 3D scene graph, Figure 8 shows how the scene graph idea can be applied to defining the contents of a living room.

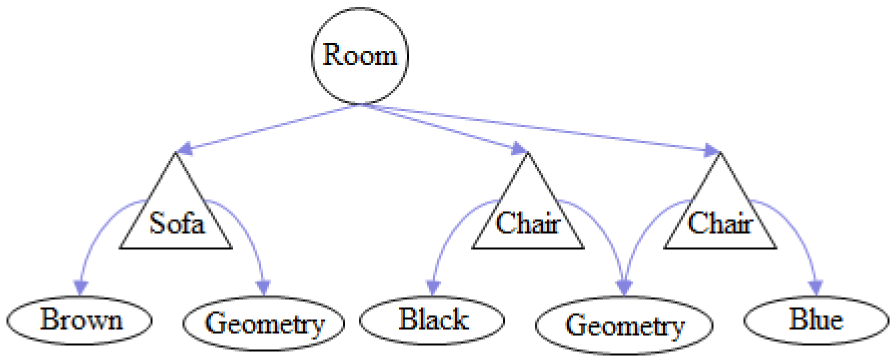


Figure 8. Scene Graph for a Living Room.

The room Group node is the parent of Leaf nodes representing a sofa and two chairs. Each Leaf utilizes geometry (shape) and color node components, and the chair geometry information is shared. This sharing means that both chairs will have the same shape, but be different colors.

The choice of symbols in Figure 8 comes from a standard symbol set (shown in Figure 9) used in scene graph diagrams. I'll explain the VirtualUniverse and Locale nodes, and the Reference relationship in a moment.

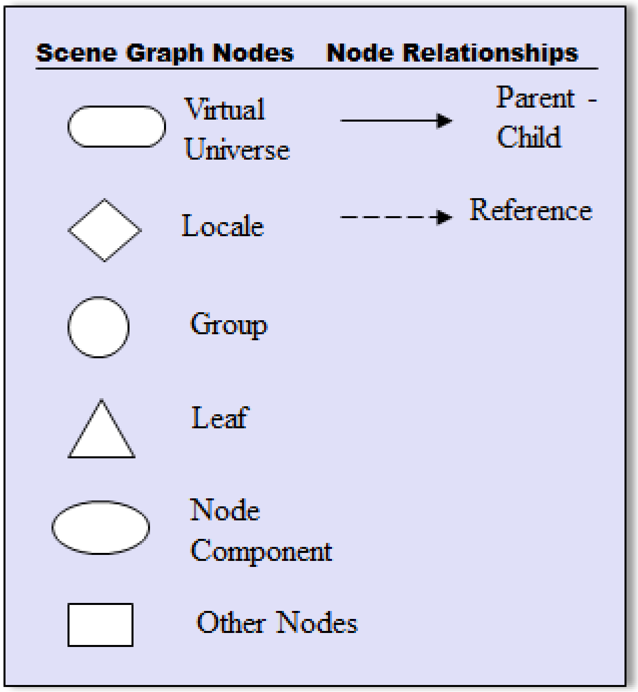


Figure 9. Scene Graph Symbols.

2.2. Some Java 3D Scene Graph Nodes

The Java 3D API can be viewed as a set of classes which subclass the Group and Leaf nodes in various ways. The Leaf class is subclassed to define different kinds of 3D shapes and environmental nodes (i.e. nodes representing lighting, sounds, and behaviors).

The main shape class is called Shape3D, which uses two node components, to define its geometry and appearance; these classes are called Geometry and Appearance.

The Group class supports basic node positioning and orientation for its children, and is subclassed to extend those operations. For instance, BranchGroup allows children to be added or removed from the graph at run time, while TransformGroup permits the position and orientation of its children to be changed.

2.3. The HelloUniverse Scene Graph

My HelloUniverse application displays a rotating colored cube, as in Figure 10.

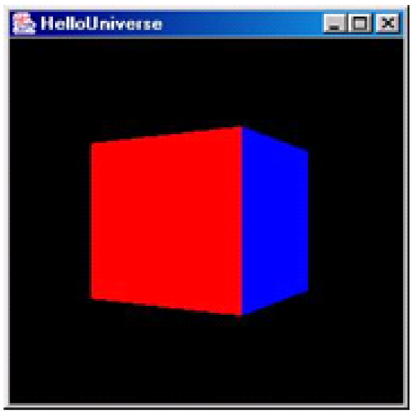


Figure 10. A Rotating Colored Cube.

It can be downloaded from
http://fivedots.coe.psu.ac.th/~ad/jg/code/HelloUniverse.zip.

Its scene graph is given in Figure 11.

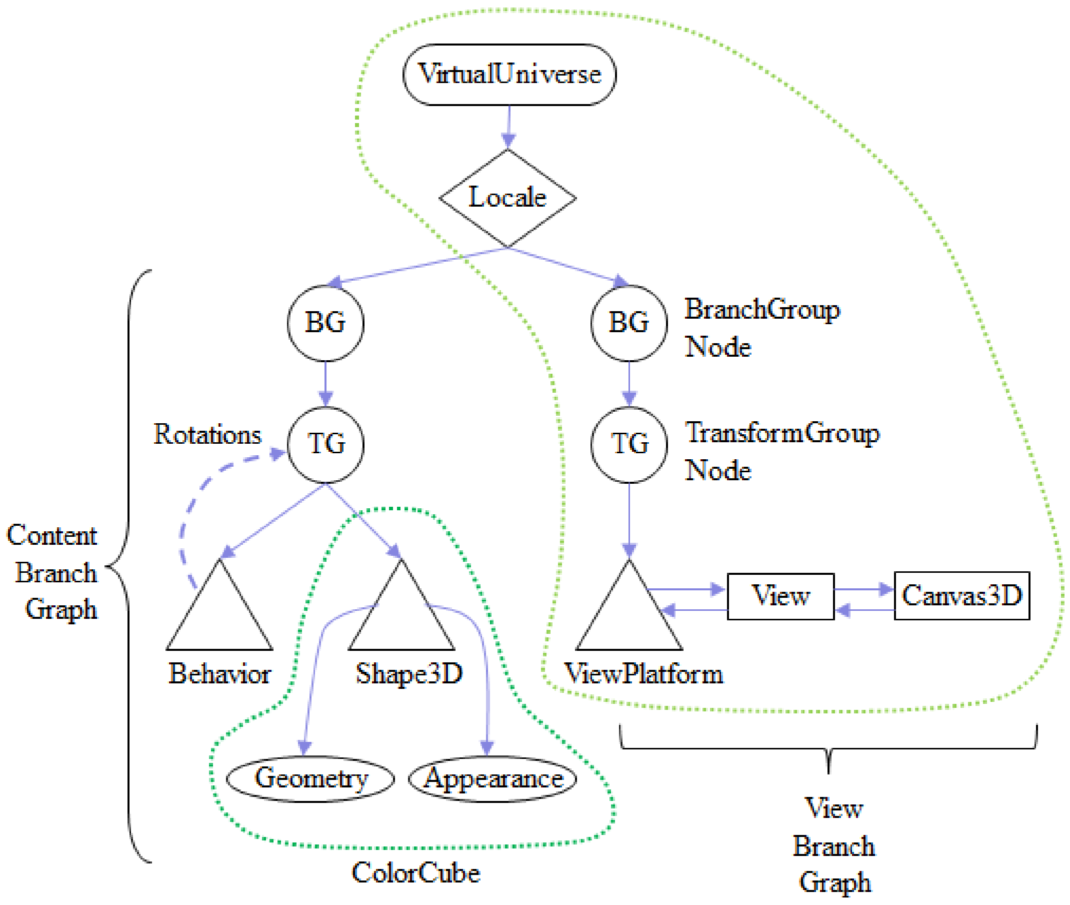


Figure 11. Scene Graph for HelloUniverse.

VirtualUniverse is the top node in every scene graph, and represents the virtual world space and its coordinate system. Locale acts as the scene graph's location in the virtual world. Below the Locale node there are always two subgraphs – the left branch is the *content branch graph*, holding program-specific content such as geometry, lighting, textures, and the world's background. The content branch graph differs significantly from one application to another.

The ColorCube is composed from a Shape3D node and associated Geometry and Appearance components. Its rotation is carried out by a Behavior node which affects the TransformGroup parent of the ColorCube's shape.

There are many predefined Behavior subclasses in the API, whose objects are triggered automatically at run time when their specified conditions become true. For example, behaviors can be linked to geometry and appearance changes, to external events such as keypresses and mouse movements, or to the passage of time.

The right hand branch below Locale is the *view branch graph*, and specifies the user’s position, orientation, and perspective as they look into the virtual world from the physical world (e.g. from in front of a monitor). The ViewPlatform node stores the viewer’s position in the virtual world; the View node states how to turn what the viewer sees into a physical world image (e.g. a 2D picture on the monitor). The Canvas3D node is a Java GUI component that allows the 2D image to be placed inside a Java application or applet.

The VirtualUniverse, Locale, and view branch graph often have the same structure across different applications, since most programs use a single Locale and view the virtual world as a 2D image on a monitor. For most applications, the relevant nodes can be created with Java 3D's SimpleUniverse utility class, relieving the programmer of a lot of graph construction work.

In summary: Java 3D applications are mostly about constructing a scene graph using a variety of classes to create different kinds of nodes.

3. Multiple Markers and Models

MultiNyAR demonstrates how to use multiple markers, each with its own 3D model. The application also displays position, orientation, and confidence information for its markers, which is useful if marker models need to be examined (e.g. is the robot facing the camera?, is the robot in front of the cow?) MultiNyAR also includes simple techniques for reducing model ‘shaking’ and handling undetected markers.

Figure 12 shows the class diagrams for MultiNyAR, with only public methods listed.

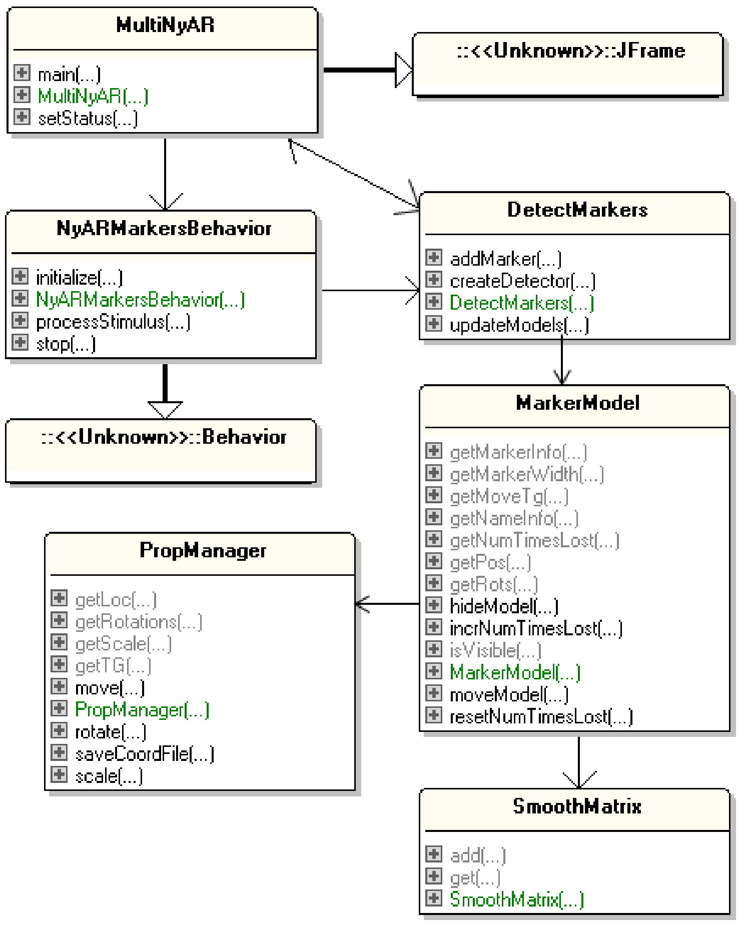


Figure 12. The MultiNyAR Class Diagrams.

MultiNyAR.java creates the GUI interface and the Java 3D scene graph. The interface is quite simple (see Figure 2): the JFrame contains a JPanel holding a Java 3D Canvas3D component for rendering the scene graph, and a JTextArea below it for displaying position, orientation, and confidence information about the markers and their models.

The content branch of the scene graph is shown in Figure 13 (I've left out the view branch for now).

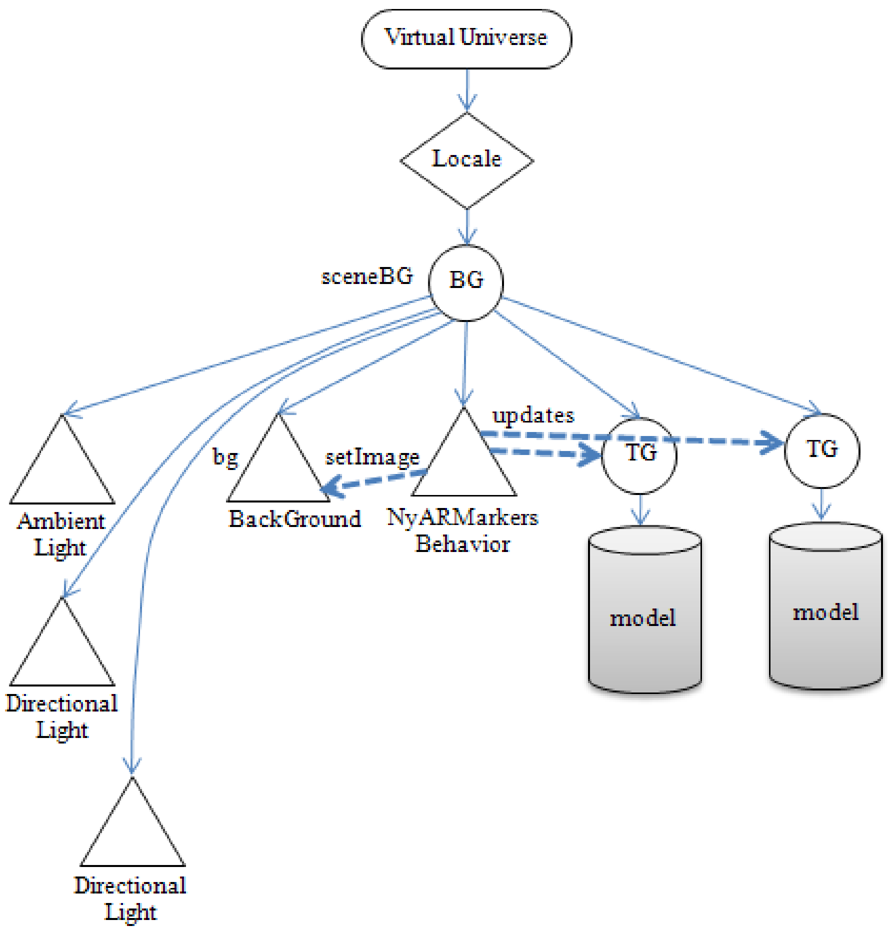


Figure 13. The Content Branch of the 3D Scene.

There are three lights (an ambient and two directional lights), a background node that displays the current webcam image, two branches for the robot and cow models, and a behavior.

NyARMarkersBehavior, is a time-triggered behavior which executes 30 times every second. It changes the image in the background node to be the current camera picture, and updates the position and orientation of the models so they stay on top of their respective markers.

Model updating is quite complex, and so NyARMarkersBehavior delegates the work to a DetectMarkers object which maintains a list of MarkerModel objects and a detector that finds the markers in the camera's captured image. A marker position is used to move its corresponding model via a TransformGroup (the TG nodes in Figure 13). Each MarkerModel object holds marker information and builds the Java 3D scene graph for its associated model.

The models are loaded using the PropManager class, which is described in chapter 16 of *Killer Game Programming in Java* (http://fivedots.coe.psu.ac.th/~ad/jg/ch9/). It utilizes the NCSA Portfolio library so the models can be defined in a range of different 3D formats.

The transforms applied to the models tend to make them ‘jitter’, due to slight variations in their calculated rotations and positions from frame to frame.

SmoothMatrix reduces the shaking by employing an averaged transform derived from the current and several previous values.

4. Creating the 3D Scene

The MultiNyAR class' constructor creates the GUI interface (a Canvas3D object and a text area for messages), and builds the Java 3D scene graph that's drawn onto the canvas.

```
// globals
private final String PARAMS_FNM = "Data/camera_para.dat";

private static final int WIDTH = 640;   // size of panel
private static final int HEIGHT = 480;

private J3dNyARParam cameraParams;
private NyARMarkersBehavior nyaBeh;
private JTextArea statusTA;


public MultiNyAR()
{
  super("Multiple markers NyARToolkit Example");
  Loader.load(opencv_objdetect.class);

  cameraParams = readCameraParams(PARAMS_FNM);

  Container cp = getContentPane();

  // create a JPanel in the center of JFrame
  JPanel p = new JPanel();
  p.setLayout( new BorderLayout() );
  p.setPreferredSize( new Dimension(WIDTH, HEIGHT) );
  cp.add(p, BorderLayout.CENTER);

  // put the 3D canvas inside the JPanel
  p.add(createCanvas3D(), BorderLayout.CENTER);

  // add status field to bottom of JFrame
  statusTA = new JTextArea(7, 10);
     // updated by DetectMarkers object (see createSceneGraph())
  statusTA.setEditable(false);
  cp.add(statusTA, BorderLayout.SOUTH);

  addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e)
    { nyaBeh.stop();
      System.exit(0);
    }
  });

  setResizable(false);
  pack();
  setLocationRelativeTo(null);
  setVisible(true);
}  // end of MultiNyAR()
```

One AR-related aspect of the constructor is the call to readCameraParams(), which loads details about the webcam. These will be used later to set up the Java 3D view graph branch, which controls the user's view of the 3D scene.

```
private J3dNyARParam readCameraParams(String fnm)
{
  J3dNyARParam cameraParams = null;
  try {
    cameraParams = J3dNyARParam.loadARParamFile(
                              new FileInputStream(fnm));
    cameraParams.changeScreenSize(WIDTH, HEIGHT);
  }
  catch(Exception e)
  {  System.out.println("Could not read params from " + fnm);
     System.exit(1);
  }
  return cameraParams;
}  // end of readCameraParams()
```

createCanvas3D() builds the scene graph, using createSceneGraph() to construct the content branch (see Figure 13), and createView() for the view branch (Figure 14 below).

```
private Canvas3D createCanvas3D()
{
  Locale locale = new Locale( new VirtualUniverse() );
  locale.addBranchGraph( createSceneGraph() );   // add the scene

  // get preferred graphics configuration for screen
  GraphicsConfiguration config =
                SimpleUniverse.getPreferredConfiguration();

  Canvas3D c3d = new Canvas3D(config);
  locale.addBranchGraph( createView(c3d) );  // add view branch

  return c3d;
}  // end of createCanvas3D()
```

createSceneGraph() calls lightScene() to add the ambient and directional lights to the graph, makeBackground() for the Background node, and creates two MakerModel objects which handle the loading of the models and their association with markers.

```
// globals
private J3dNyARParam cameraParams;
private NyARMarkersBehavior nyaBeh;


private BranchGroup createSceneGraph()
// creates the scene graph shown in Figure 13 above
{
  BranchGroup sceneBG = new BranchGroup();
  lightScene(sceneBG);              // add lights

  Background bg = makeBackground();
  sceneBG.addChild(bg);             // add background
  DetectMarkers detectMarkers = new DetectMarkers(this);

  // the "hiro" marker uses a robot model
  MarkerModel mm1 =
           new MarkerModel("patt.hiro", "robot.3ds", 0.15, false);
  if (mm1.getMarkerInfo() != null) {     // creation was successful
    sceneBG.addChild( mm1.getMoveTg() );
    detectMarkers.addMarker(mm1);
  }

  // the "kanji" marker uses a cow model
  MarkerModel mm2 =
           new MarkerModel("patt.kanji", "cow.obj", 0.12, true);
  if (mm2.getMarkerInfo() != null) {
    sceneBG.addChild( mm2.getMoveTg() );
    detectMarkers.addMarker(mm2);
  }

  // start the detector once all markers have been added
  detectMarkers.createDetector(cameraParams);

  // create a NyAR multiple marker behavior
  nyaBeh = new NyARMarkersBehavior(bg, detectMarkers);
  sceneBG.addChild(nyaBeh);

  sceneBG.compile();       // optimize the sceneBG graph
  return sceneBG;
  }  // end of createSceneGraph()
```

The MarkerModel objects are managed by a DetectMarkers object, which finds the markers in the current camera frame, and moves their models so they stay positioned over those markers.

The top-level TransformGroups for the model branches are retrieved with calls to MarkerModel.getMoveTg(), and linked to the sceneBG BranchGroup node.

The NyARMarkersBehavior must communicate with the background node, and with the model branches (see Figure 13). Instead of passing TransformGroup references to NyARMarkersBehavior, the behavior is supplied with a pointer to the DetectMarkers object, which acts as their manager.

The Background node is initialized in the standard manner inside makeBackground(), except that its capability bits are set to allow its image to be updated at run time. The background will keep being changed to show the current webcam view.

```
private Background makeBackground()
{
  Background bg = new Background();
  BoundingSphere bounds = new BoundingSphere();
  bounds.setRadius(10.0);

  bg.setApplicationBounds(bounds);
  bg.setImageScaleMode(Background.SCALE_FIT_ALL);
  bg.setCapability(Background.ALLOW_IMAGE_WRITE);  // to change image
  return bg;
}  // end of makeBackground()
```

The User's Viewpoint

The view branch shown in Figure 14 is built by createView() since its View node has to be initialized with the camera's properties. The content branch generated by createSceneGraph() is drawn as a dotted triangle in the figure.

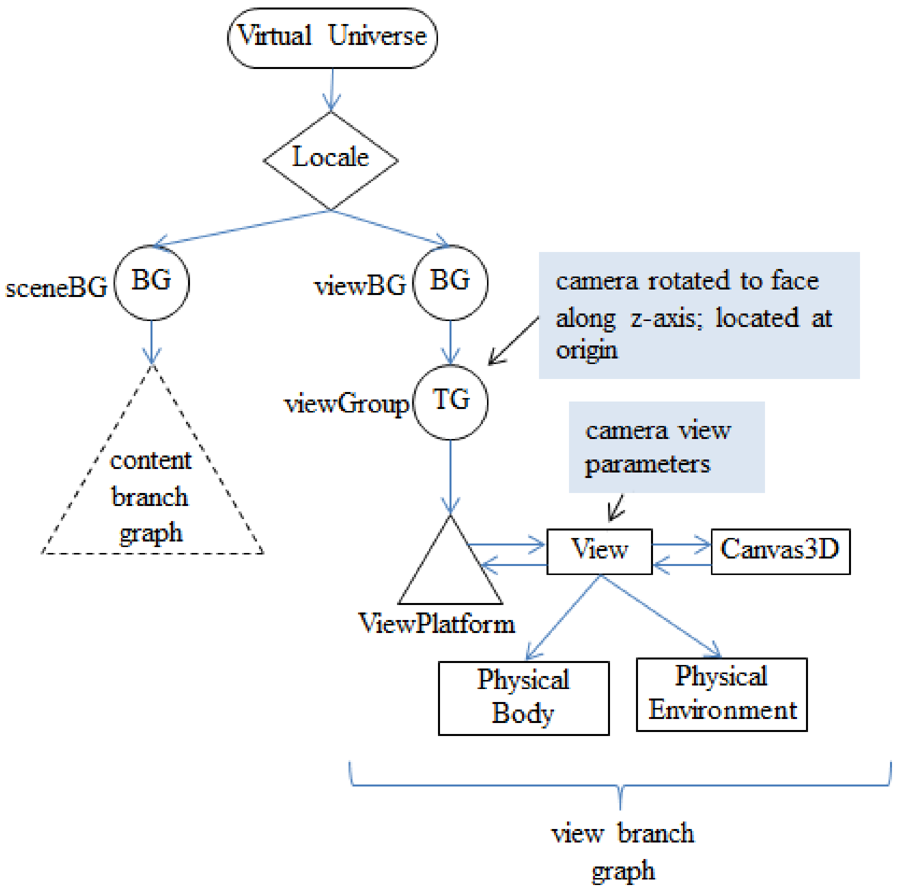


Figure 14. The View Branch of the 3D Scene.

The createView() method:

```
// globals
private J3dNyARParam cameraParams;


private BranchGroup createView(Canvas3D c3d)
// create a view graph using the camera parameters
{
  View view = new View();
  ViewPlatform viewPlatform = new ViewPlatform();
  view.attachViewPlatform(viewPlatform);
  view.addCanvas3D(c3d);

  view.setPhysicalBody(new PhysicalBody());
  view.setPhysicalEnvironment(new PhysicalEnvironment());

  view.setCompatibilityModeEnable(true);
  view.setProjectionPolicy(View.PERSPECTIVE_PROJECTION);
  view.setLeftProjection( cameraParams.getCameraTransform() );
                                // camera projection

  TransformGroup viewGroup = new TransformGroup();
  Transform3D viewTransform = new Transform3D();
  viewTransform.rotY(Math.PI);   // rotate 180 degrees
  viewTransform.setTranslation(new Vector3d(0.0, 0.0, 0.0)); //origin
  viewGroup.setTransform(viewTransform);
  viewGroup.addChild(viewPlatform);

  BranchGroup viewBG = new BranchGroup();
  viewBG.addChild(viewGroup);

  return viewBG;
}  // end of createView()
```

The user's viewpoint is placed at the origin and faces along the positive z-axis, as illustrated by Figure 15.

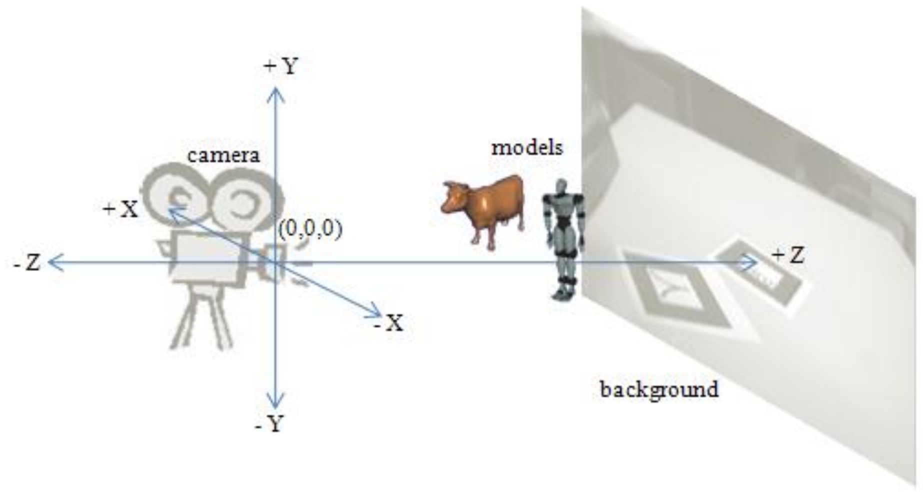


Figure 15. The User's Viewpoint.

The Background node is rendered at the back of the scene, and the models positioned so they appear to be resting on top of the markers shown in the background picture.

The y-axis rotation of the camera means that the positive x-axis runs to the left.

5. The Scene Graph's Behavior

The NyARMarkersBehavior object is a time-based behavior which grabs a image from the webcam, pastes it into the background node, and also passes it to DetectMarkers. Inside DetectMarkers, a NyARToolkit detector examines the image and updates the 3D positions of the models drawn above the markers.

The NyARMarkersBehavior constructor initializes the FrameGrabber, and sets the 'wakeup' period for the behavior to 1000/30 milliseconds, so it will fire roughly 30 times/second.

```
// globals
private final double FPS = 30.0;

private FrameGrabber grabber;
private NyARBufferedImageRaster raster;
private ImageComponent2D imc2d;
private DetectMarkers detectMarkers;

private Background bg;
private WakeupCondition wakeup;
private boolean cameraStopped = false;


public NyARMarkersBehavior(Background bg,DetectMarkers ms)
{
  super();
  this.bg = bg;
  detectMarkers = ms;

  wakeup = new WakeupOnElapsedTime((int)(1000.0/FPS));
  setSchedulingBounds( new BoundingSphere(new Point3d(), 100.0) );

  grabber = initGrabber(CAMERA_ID);
  BufferedImage im = (picGrab(grabber,CAMERA_ID)).getBufferedImage();
  try {
    raster = new NyARBufferedImageRaster(im);
    imc2d = new ImageComponent2D(ImageComponent2D.FORMAT_RGB, im,
                                                   true, false);
    imc2d.setCapability(ImageComponent.ALLOW_IMAGE_WRITE);
  }
  catch(NyARException e)
  {  System.out.println(e);
     System.exit(1);
  }
}  // end of NyARMarkersBehavior()
```

An image is snapped by the FrameGrabber in this method so that a Java 3D ImageComponent2D object can be initialized. ImageComponent2D is an image-wrapper used by Java 3D to update the background during execution.

The behavior’s processStimulus() method is called every 'wakeup' milliseconds, grabs an image, updates the background (using the imc2d global), and passes the BufferedImage to the detector.

```
// globals
private FrameGrabber grabber;
private NyARBufferedImageRaster raster;
private ImageComponent2D imc2d;
private DetectMarkers detectMarkers;

private Background bg;
private WakeupCondition wakeup;
private boolean cameraStopped = false;


public void processStimulus(Enumeration criteria)
{
  if (cameraStopped)
```

```
    return;

  try {
    BufferedImage im = (picGrab(grabber, CAMERA_ID)).
                                       getBufferedImage();
    raster.wrapImage(im);
    if (raster.hasBuffer()) {
      if (bg != null) {
        imc2d.set(im);
        bg.setImage(imc2d);
      }
      detectMarkers.updateModels(raster);
    }
    wakeupOn(wakeup);
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}  // end of processStimulus()
```

The NyARBufferedImageRasterclass is a NyARToolkit wrapper around BufferedImage. processStimulus() uses it's wrapImage() method to update the raster.

6. Markers and Models

The MarkerModel constructor is supplied with the filenames of the marker pattern (e.g. "patt.hiro") and the 3D model (e.g. "robot.3ds"), and loads them. For example:

```
MarkerModel mm1 =
       new MarkerModel("patt.hiro", "robot.3ds", 0.15, false);
```

Loading the model is the harder of the two tasks, but most of the work is handled by the PropManager class which uses the NCSA Portfolio library. A sequence of three nodes are attached below the sceneBG BranchGroup node as shown in Figure 19.

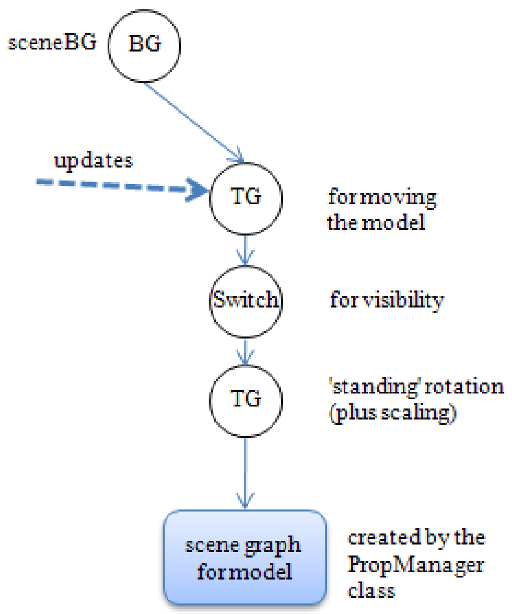


Figure 16. The Scene Graph for a Model.

Most of the code for building the branch in Figure 16 is located in MarkerModel(), as shown below:

```
// globals
private String markerName, modelName;
private NyARCode markerInfo = null;  // NyARToolkit marker details

private TransformGroup moveTg;       // for moving the marker model

private Switch visSwitch;    // for changing the model's visibility
private boolean isVisible;

private SmoothMatrix sMat;
                // for smoothing the transform applied to the model



public MarkerModel(String markerFnm, String modelFnm,
                             double scale, boolean hasCoords)
{
  markerName = markerFnm;
  modelName = modelFnm.substring(0, modelFnm.lastIndexOf('.'));
                                  // remove filename extension

  // build a branch for model: TG --> Switch --> TG --> model

  // load the model, with scale and coords info
  TransformGroup modelTG = loadModel(modelFnm, scale, hasCoords);

  // create switch for model visibility
  visSwitch = new Switch();
  visSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);
  visSwitch.addChild( modelTG );
  visSwitch.setWhichChild( Switch.CHILD_NONE );    // make invisible
  isVisible = false;

  // create transform group for positioning the model
  moveTg = new TransformGroup();
  moveTg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
                                      // so this tg can change
  moveTg.addChild(visSwitch);

  try {   // load marker info
      markerInfo = NyARCode.createFromARPattFile(
          new FileInputStream(MARKER_DIR+markerName),16, 16);
  }
  catch(NyARException e)
  {  System.out.println(e);
     markerInfo = null;
  }

  sMat = new SmoothMatrix();
}  // end of MarkerModel()
```

The MarkerModel constructor takes four arguments: the filenames of the marker pattern (markerFnm) and the 3D model (modelFnm), a scale factor, and a coordinates file boolean. The scale factor is used to tweak the size of the model on top of the marker. The coordinates file boolean specifies whether the model comes with optional coordinates information in a separate file, which is used by PropManager when it loads the model (I'll explain this in a little more detail when I overview the PropManager class).

The scale factor and coordinates boolean are passed to loadModel(), which calls PropManager, and creates the 'standing' rotation and scaling TransformGroup node at the bottom of Figure 16.

```
private TransformGroup loadModel(String modelFnm,
                                   double scale, boolean hasCoords)
// load the model, rotating and scaling it
{
  PropManager propMan = new PropManager(modelFnm, hasCoords);

  // get the TG for the model
  TransformGroup propTG = propMan.getTG();

  // rotate and scale the model
  Transform3D modelT3d = new Transform3D();
  modelT3d.rotX( Math.PI/2.0 );
       // the model lies flat on the marker;
       // rotate forwards 90 degrees so it is standing
  Vector3d scaleVec = calcScaleFactor(propTG, scale);
  modelT3d.setScale( scaleVec );  // scale the model

  TransformGroup modelTG = new TransformGroup(modelT3d);
  modelTG.addChild(propTG);

  return modelTG;
}  // end of loadModel()
```

The modelTG node created by loadModel() transforms the loaded model in two ways – it scales the model using the scale argument and rotates it 90 degrees around the x-axis. The latter operation deals with the way that NyARToolkit positions a model on top of a marker. By default, it is lying 'on its back' on the marker; the rotation turns the model so it is 'standing'.

Back in MarkerModel(), the two other nodes shown in Figure 16 are added. The Switch node allows the branch to be made invisible when the model's marker is not found in the image. The top-most TransformGroup (moveTG) is utilized to position the model so it stays with its marker.

6.1. Moving a Model

Updates to the model's moveTg TransformGroup node are handled by moveModel(). DetectMarker passes the method the position of the marker (transMat) specified in the camera's coordinate system. It is converted into a transformation in the scene's coordinate system and applied to moveTg.

```
// globals
private TransformGroup moveTg;      // for moving the marker model

private Switch visSwitch;     // for changing the model's visibility
private boolean isVisible;

private SmoothMatrix sMat;
       // for smoothing the transforms applied to the model


public void moveModel(NyARDoubleMatrix44 transMat)
// update the model's moveTG
{
  visSwitch.setWhichChild( Switch.CHILD_ALL );   // make visible
  isVisible = true;

  sMat.add(transMat);
  Matrix4d mat = sMat.get();
  Transform3D t3d = new Transform3D(mat);

  int flags = t3d.getType();
  if ((flags & Transform3D.AFFINE) == 0)
    System.out.println("Ignoring non-affine transformation");
  else {
    if (moveTg != null)
      moveTg.setTransform(t3d);
    calcPosition(mat);
    calcEulerRots(mat);
  }
}  // end of moveModel()
```

If moveModel() is called at all, then it means that DetectMarkers has found the model's marker in the image. In that case, the model must be made visible, and moved to the marker's position. The conversion of the transMat value from camera coordinates to scene coordinates is dealt with by an instance of SmoothMatrix (sMat). which also reduces the shaking of a model. The SmoothMatrix.get() method returns an averaged transform based on the value just added (with SmoothMatrix.add()), and several values from previous frame captures.

The transformation, mat, is 4 x 4 homogenous matrix like the one shown in Figure 17.

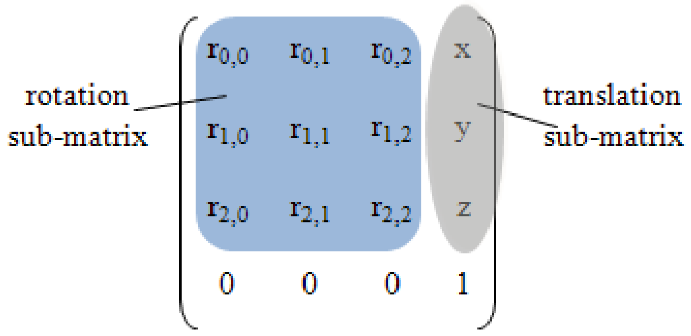

rotation sub-matrix
translation sub-matrix
$r_{0,0}$ $r_{0,1}$ $r_{0,2}$ x
$r_{1,0}$ $r_{1,1}$ $r_{1,2}$ y
$r_{2,0}$ $r_{2,1}$ $r_{2,2}$ z
0 0 0 1

Figure 17. The Transformation Matrix.

The top-left 3 x 3 sub-matrix contains the rotational parts of the transformations, while the top 3 elements of the fourth column are the translation component. calcPosition() and calcEulerRots() extracts this information, storing it in globals for future use.

```
// global
private Point3d posInfo = null;   // the model's current position


private void calcPosition(Matrix4d mat)
{
  // convert to larger units and round to 1 dp
  double x = roundToNumPlaces( mat.getElement(0,3)*100, 1);
  double y = roundToNumPlaces( mat.getElement(1,3)*100, 1);
  double z = roundToNumPlaces( mat.getElement(2,3)*100, 1);
  posInfo = new Point3d(x, y, z);
}  // end of reportPosition()


private double roundToNumPlaces(double val, int numPlaces)
{
  double power = Math.pow(10, numPlaces);
  long temp = Math.round(val*power);
  return ((double)temp)/power;
}
```

calcPosition() stores the (x, y, z) position in a global Point3d object, after multiplying the values by 100, and rounding that value to 1 decimal place with roundToNumPlaces(). This data is accessible via a getPos() method, and is printed in the status area of the GUI (e.g. see Figures 2 and 3).

Probably the most user-friendly way of understanding a model's orientation is in terms of Euler rotation angles around the x-, y-, and z- axes. Extracting these from the rotational sub-matrix in Figure 17 is a little tricky mathematically, as calcEulerRots() demonstrates.

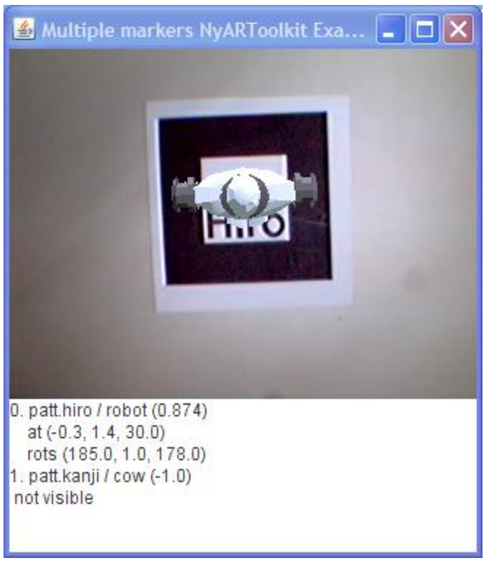
```
// global
private Point3d rotsInfo = null;
             // the model's current orientation (in degrees)
```

```
private void calcEulerRots(Matrix4d mat)
/* calculate the Euler rotation angles from the upper 3x3 rotation
   components of the 4x4 transformation matrix. */
{
  rotsInfo = new Point3d();

  rotsInfo.y = -Math.asin(mat.getElement(2,0));
  double c = Math.cos(rotsInfo.y);

  double tRx, tRy, tRz;
  if(Math.abs(rotsInfo.y) > 0.00001) {
    tRx = mat.getElement(2,2)/c;
    tRy = -mat.getElement(2,1)/c;
    rotsInfo.x = Math.atan2(tRy, tRx);

    tRx = mat.getElement(0,0)/c;
    tRy = -mat.getElement(1,0)/c;
    rotsInfo.z = Math.atan2(tRy, tRx);
  }
  else {
    rotsInfo.x  = 0.0;

    tRx = mat.getElement(1,1);
    tRy = mat.getElement(0,1);
    rotsInfo.z = Math.atan2(tRy, tRx);
  }

  rotsInfo.x = -rotsInfo.x;
  rotsInfo.z = -rotsInfo.z;

  // ensure the values are positive by adding 2*PI if necessary...
  if(rotsInfo.x < 0.0)
    rotsInfo.x += 2*Math.PI;
  if(rotsInfo.y < 0.0)
    rotsInfo.y += 2*Math.PI;
  if(rotsInfo.z < 0.0)
    rotsInfo.z += 2*Math.PI;

  // convert to degrees and round
  rotsInfo.x = roundToNumPlaces( Math.toDegrees(rotsInfo.x), 0);
  rotsInfo.y = roundToNumPlaces( Math.toDegrees(rotsInfo.y), 0);
  rotsInfo.z = roundToNumPlaces( Math.toDegrees(rotsInfo.z), 0);
}  // end of calcEulerRots()
```

calcEulerRots() is based on code written by Daniel Selman, which was derived from pseudo-code in Question 37 of the "Matrix and Quaternion FAQ" at http://www.j3d.org/matrix_faq/matrfaq_latest.html. A good discussion of the maths can be found in “Computing Euler angles from a rotation matrix” by Gregory G. Slabaugh at https://truesculpt.googlecode.com/hg-history/38000e9dfece971460473d5788c235fbbe82f31b/Doc/rotation_matrix_to_euler.pdf.

These angles are accessible via a getRots() method, and are printed in the status area of the GUI (e.g. see Figures 18, 20, and 21 below).

Figure 18 shows a view of the robot model with the camera directly above the “Hiro” marker, with the marker writing straight up.



```
0. patt.hiro / robot (0.874)
   at (-0.3, 1.4, 30.0)
   rots (185.0, 1.0, 178.0)
1. patt.kanji / cow (-1.0)
 not visible
```

Figure 18. The “Hiro” Marker Upwards, Facing the Camera.

The rotation values should be understood in terms of the z-, y-, and z- axes relative to the camera, which are shown in Figure 19.

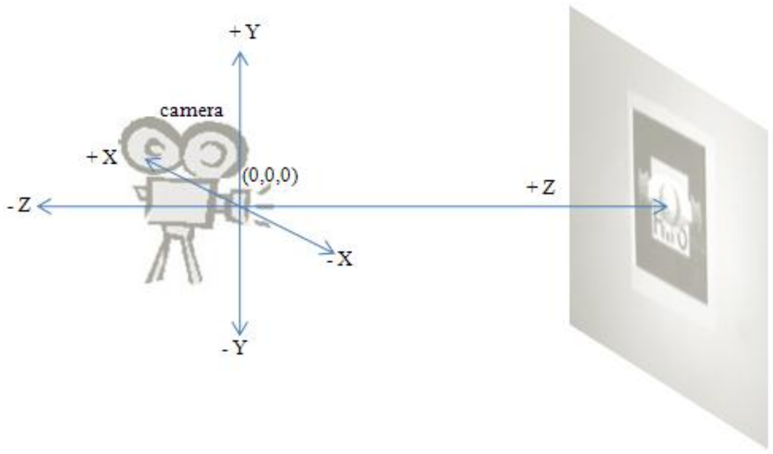
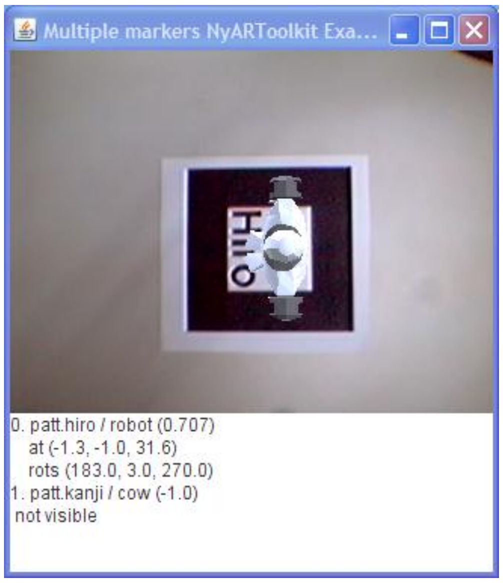


Figure 19. The Axes Relative to the Camera in Figure 21.

The positive z-axis heads forwards from the camera, which in the case of Figure 18 is straight down onto the table top.

The rotation angles shown in Figure 18 are (185, 1, 178), which roughly corresponds to a half rotation of the model around the x-axis and another half turn around the z-axis.

Figure 20 shows the “Hiro” marker after being turned 90 degrees clockwise relative to the camera.

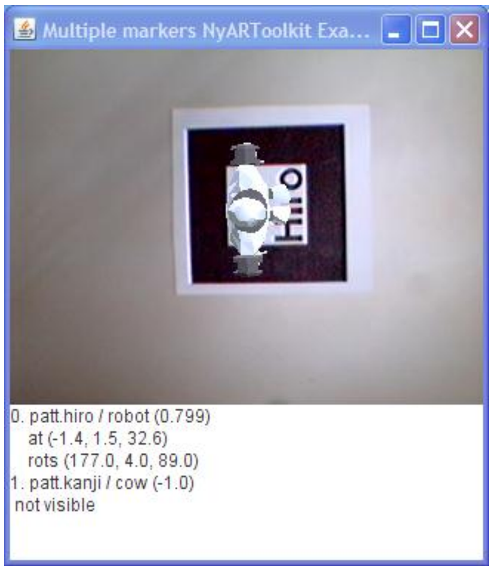


```
0. patt.hiro / robot (0.707)
   at (-1.3, -1.0, 31.6)
   rots (183.0, 3.0, 270.0)
1. patt.kanji / cow (-1.0)
 not visible
```

Figure 20. The "Hiro" Marker Turned to the Right, Facing the Camera.

The rotations shown in Figure 20 are (183, 3, 270) which is roughly a 90 degree rotation around the z-axis change from the values reported in Figure 18.

Figure 21 shows the "Hiro" marker after being turned 90 degrees anti-clockwise from its position in Figure 18.



```
0. patt.hiro / robot (0.799)
   at (-1.4, 1.5, 32.6)
   rots (177.0, 4.0, 89.0)
1. patt.kanji / cow (-1.0)
 not visible
```

Figure 21. The "Hiro" Marker Turned to the Left, Facing the Camera.

The rotation values confirm this change from Figure 18, reporting (177, 4, 89), which is roughly a -90 degree turn around the z-axis.

6.2. Model Visibility

Figures 18, 20, and 21 also illustrate what happens when a marker cannot be found in the webcam image. The "kanji" marker is missing, so the cow model is not added to the scene. This is noted in the status area as the marker and model being "not visible".

Visibility is controlled by a Switch node (see Figure 16) and a boolean, which are initialized in the MarkerModel constructor:

```
// globals
private Switch visSwitch;   // for changing the model's visibility
private boolean isVisible;


// in MarkerModel()
// create switch for model visibility
visSwitch = new Switch();
visSwitch.setCapability(Switch.ALLOW_SWITCH_WRITE);
visSwitch.addChild( modelTG );
visSwitch.setWhichChild( Switch.CHILD_NONE );   // make invisible
isVisible = false;
```

When DetectMarker calls moveModel() to move the model, it is also made visible:

```
// in makeModel()
visSwitch.setWhichChild( Switch.CHILD_ALL );   // make visible
isVisible = true;
```

It is possible to test a model’s visibility by calling isVisible(), and turn a model invisible with hideModel().

```
public boolean isVisible()
{  return isVisible; }


public void hideModel()
{
  visSwitch.setWhichChild( Switch.CHILD_NONE );  // make invisible
  isVisible = false;
}
```

The tricky aspect of hideModel() is deciding *when* to call it. As Figures 18, 20, and 21 suggest, if a marker cannot be found then the model should be hidden. However, in practice this leads to a model ‘flickering’ in and out of sight since a marker is usually only ‘lost’ briefly, as occurs when a user’s fingers obscure part of the marker's black border. A better strategy is to hide the model only after the marker has gone missing for several frames.

This delayed hiding technique is implemented in DetectMarkers, and so will be described in more detail later. However, it relies on a numTimesLost counter maintained in each MarkerModel object.

```
// global
private int numTimesLost = 0;
   // number of times marker for this model not detected


public void resetNumTimesLost()
{  numTimesLost = 0;  }
```

```
public void incrNumTimesLost()
{  numTimesLost++;  }

public int getNumTimesLost()
{  return numTimesLost;  }
```

7. Smoothing a Matrix Transform

SmoothMatrix reduces model shaking caused by slight variations in the calculated rotations and positions of the transformation matrix in each video frame.

The current transformation matrix is passed to SmoothMatrix as a NyARDoubleMatrix44 object, and converted to a Java 3D Matrix4d object. In the process the coordinates, which are specified relative to the camera, are converted into scene coordinates.

The resulting Java 3D matrix is added to a list which stores up to the MAX_SIZE matrices.

```
// globals
private final static int MAX_SIZE = 10;

private ArrayList<Matrix4d> matsStore;
private int numMats = 0;


public boolean add(NyARDoubleMatrix44 transMat)
{
  Matrix4d mat = new Matrix4d(-transMat.m00, -transMat.m01,
                                    -transMat.m02, -transMat.m03,
                              -transMat.m10, -transMat.m11,
                                    -transMat.m12, -transMat.m13,
                               transMat.m20,  transMat.m21,
                                     transMat.m22,  transMat.m23,
                               0,     0,      0,     1 );
  Transform3D t3d = new Transform3D(mat);

  int flags = t3d.getType();
  if ((flags & Transform3D.AFFINE) == 0) {
    System.out.println("Not adding a non-affine matrix");
    return false;
  }
  else {
    if (numMats == MAX_SIZE) {
      matsStore.remove(0);   // remove oldest
      numMats--;
    }
    matsStore.add(mat);  // add at end of list
    numMats++;
    return true;
  }
}  // end of add()
```

The coordinates conversion is done as the Matrix4d object is filled with values from the NyARDoubleMatrix44 object, by negating the first two rows of data. I copied this code, with some simplifications, from the NyARToolkit Java 3D example, NyARJava3D.java, but I have to admit to being baffled by it (although it ‘works’). The conversion does not seem to match the documentation on the coordinate systems employed in ARToolkit, as described at http://www.hitl.washington.edu/artoolkit/documentation/cs.htm.

SmoothMatrix’s get() method calculates an average of the matrices in its list, thereby reducing any errors caused by incorrect readings.

```
public Matrix4d get()
// average matrices in store
{
  if (numMats == 0)
    return null;

  Matrix4d avMat = new Matrix4d();
  for(Matrix4d mat : matsStore)
    avMat.add(mat);
  avMat.mul( 1.0/numMats );

  return avMat;
}  // end of get()
```

The maximum size of the list is defined in MAX_SIZE (10), and the larger this value, the more smoothed will be the transformation returned by get(). The downside is that a marker movement will take longer to have a visible effect on a displayed model.

The timing of NyARMarkersBehavior is controlled by the constant FPS (30), which is utilized as the millisecond time 1000/FPS. This means that the behavior fires roughly 30 times per second, and so the ten transforms stored in SmoothMatrix span about a third of a second. This will be the maximum lag time before a marker movement is reflected in its model’s movement.

8. Loading a Model

The loadModel() method in MarkerModel uses the PropManager class to do the hard work of loading a model.

```
// in loadModel()
PropManager propMan = new PropManager(modelFnm, hasCoords);
TransformGroup propTG = propMan.getTG();
```

The filename of the 3D model (modelFnm) and a coordinates file boolean are passed to PropManager’s constructor. The coordinates file boolean specifies whether the model comes with optional coordinates information in a separate file.

PropManager is described in detail in chapter 16 of *Killer Game Programming in Java* (http://fivedots.coe.psu.ac.th/~ad/jg/ch9/). It utilizes the NCSA Portfolio library which supports many formats, including 3D Studio Max (3DS files), AutoCAD (DXF), Digital Elevation Maps (DEM), TrueSpace (COB), and VRML 97 (WRL). The drawbacks of Portfolio are its very advanced age (the current version is 1.3, from 1998), and its relatively simple support of the formats: often only the geometry and shape colors are loaded, without textures, behaviors, or lights. Aside from Portfolio, there's a wide range of Java 3D loaders for different file formats, written by third party developers; try searching for "Java 3D loaders".

I utilize three features of PropManager:

1. It loads the specified model, scales it to a standard size, and rotates the model if it's been saved as a 3DS (3D Studio Max) file.
2. It loads a coords data file if requested, and applies the translations, rotations, and scaling values in that file to the model.
3. It makes the top-level TransformGroup for the model available.

In Figure 16 a model loaded by PropManager is represented by a rounded rectangular box, labeled as “scene graph for model”. Figure 22 shows some of the hidden detail in that box.

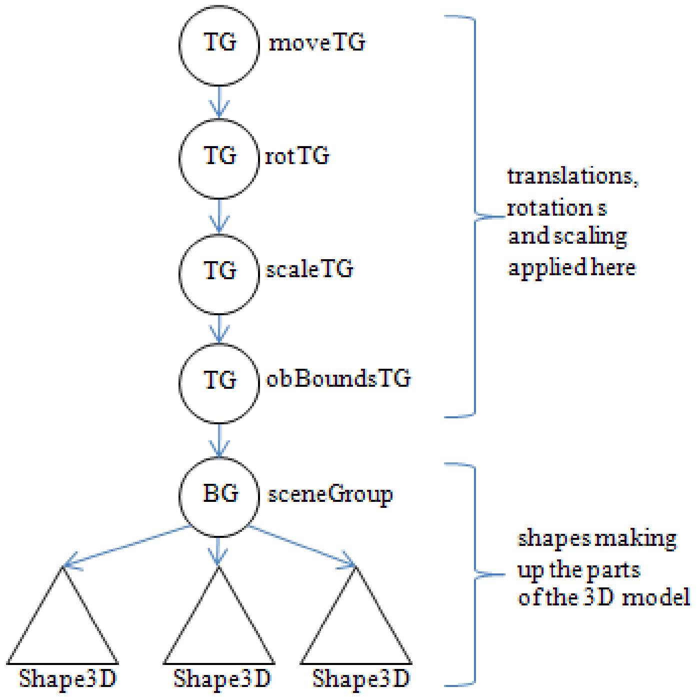

TG
moveTG
TG
rotTG
TG
scaleTG
TG
obBoundsTG
BG
sceneGroup
Shape3D
Shape3D
Shape3D
translations, rotations and scaling applied here
shapes making up the parts of the 3D model

Figure 22. Scene Graph for a Model.

PropManager creates a chain of four TransformGroup nodes, and utilizes the Portfolio library to convert the 3D model into the BranchGroup and its children. The TransformGroups deal with different aspects of the model’s configuration:

- moveTG handles the translations;
- rotTG for rotations;
- scaleTG for scaling;
- objBoundsTG carries out the scaling and possible rotation of the model when it is first loaded.

The reason for this separation is to process distinct operations in different nodes in the graph. This reduces the overall complexity of the coding, because I can take advantage of the hierarchy of local coordinate systems used by the TransformGroup nodes.

A coords data file is created with the Loader3D application, which is described in chapter 16 of *Killer Game Programming in Java* (http://fivedots.coe.psu.ac.th/~ad/jg/ch9/). The robot model doesn't need one, but the cow's data file is

```
cow.obj
-p 0 0.6 0.2
-r 34444444444
-s 1
```

stored in cowCoords.txt, in the same model/ subdirectory as cow.obj.

The -p line gives a (x, y, z) translation, the -r line contains a series of rotations, and the -s value is for scaling.

The aim of using a coordinates file is to ensure that the model is positioned so it is standing over the origin on the XZ plane, with its front facing the positive z-axis. There's no real need to scale the model, since that aspect can be tweaked with the scale factor parameter passed to the MarkerModel constructor.

No coordinates file is needed for the robot since it's already standing on the XZ plane, and facing forward. This positioning can be done with any 3D modeling tool; a simple one that I've used in the past is AccuTrans3D (http://www.micromouse.ca/), which is mainly a 3D file conversion program, but can also be used for simple model re-orientation tasks.

Alternatives to PropManager and NCSA Portfolio

There's no need to use PropManager and Portfolio. Any loader that can create a scene branch like the one in Figure 16 is fine.

Inspector3ds is a feature-rich 3DS loader, developed by John Wright at Starfire Research (http://www.starfireresearch.com/services/java3d/inspector3ds.html). The loader handles geometry, materials, textures, and normals.

Programmers wishing to utilize a modern VRML loader should consider the Xj3D loader (http://www.xj3d.org/), which covers most of the VRML 2.0 standard. The actual aim is to load X3D files, which extend VRML with XML functionality.

For the artistically-impaired (e.g. yours truly), there are a profusion of Web sites that offer 3D models. Two sites with many free models are Archive3d (http://archive3d.net/) and 3Dxtras (http://www.3dxtras.com/).

9. Detecting Markers

The DetectMarkers object maintains two data structures – a list of MarkerModel objects and a NyARDetectMarker detector. At execution time, its updateShapes() method is repeatedly called with the current image from the webcam. The picture is analyzed by the detector, markers found, and their positions and orientations used to update the models.

The creation of the detector and its search for multiple markers are two important differences between this example and the NyARJava3D sample in the NyARToolkit download. (Another difference is that my code employs JavaCV rather than JMF.)

NyARJava3D utilizes a utility class called NyARSingleMarkerBehaviourHolder which creates a detector by instantiating the NyARSingleDetectMarker class. As the name suggests, this detector only looks for a single marker (via a call to NyARSingleDetectMarker.detectMarkerLite()). The code surrounding this call assumes that the marker is either found or missed, and doesn't deal with multiple results or confidence levels.

An explanation of the extra complexity of dealing with multiple markers can be found in the ARToolkit documentation "Developing your First Application, Part 2", (http://www.hitl.washington.edu/artoolkit/documentation/devmulti.htm), but it utilizes a C-like interface to OpenGL.

NyARToolkit does include a multiple marker example, but coded in JOGL (JavaSimpleLite2.java).

9.1. Creating a Detector

A multiple markers detector is created by instantiating the NyARDetectMarker class, which requires details about the markers and camera parameters. This is done by createDetector() in DetectMarkers:

```
// globals
private ArrayList<MarkerModel> markerModels;
private int numMarkers;
private NyARDetectMarker detector;


public void createDetector(NyARParam params)
{
  NyARCode[] markersInfo = new NyARCode[numMarkers];
  double[] widths = new double[numMarkers];
  int i = 0;
  for (MarkerModel mm : markerModels) {
     markersInfo[i] = mm.getMarkerInfo();
     widths[i] = mm.getMarkerWidth();
     i++;
  }

  try {
    detector = new NyARDetectMarker(params, markersInfo,
                                        widths, numMarkers);
    detector.setContinueMode(false);
              // no history stored; use SmoothMatrix instead
  }
  catch(NyARException e)
  {  System.out.println("Could not create markers detector");
     System.exit(1);
  }
}  // end of createDetector()
```

createDetector() is called after all the MarkerModel objects have been added to the markerModels list. The camera parameters (params) is passed in as an argument, and the markers information is extracted from the MarkerModels.

The built-in detector has a “continue” mode which maintains a history of previous detections to help it find markers in the current image, set by NyARDetectMarker.setContinueMode(). When I tried using it, I usually found that marker detection became less reliable, causing models to be positioned and rotated all around the image. For that reason, I decided to write my own SmoothMatrix class to help reduce model ‘shake’, and so setContinueMode() is false in createDetector().

9.2. Updating Shapes

Pseudo-code for updateShapes() roughly corresponds to the first three stages of Figure 1:

```
for each MarkerModel
  find marker in image frame that best matches the MarkerModel;
  get transformation for the found marker;
  update the MarkerModel’s model with the transformation;
```

The loop becomes a bit more complicated when we factor in what to do when no marker is found for a given MarkerModel, or the found marker has a low likelihood of matching the MarkerModel.

I deal with the first issues by having each MarkerModel keep a count of how many times no suitable marker is found when updateShapes() is called. When the number of missed detections reaches a given maximum, the model is made invisible, until it’s marker is found again. This means that if a marker is lost only for a short time, then the model will stay on screen at its last position, but eventually the model will disappear if the marker remains undetected.

The full code for updateShapes():

```
// globals
private final static double MIN_CONF = 0.3;
        // smallest confidence accepted for finding a marker

private final static int CONF_SIZE = 1000;
       // for converting confidence level double <--> integer

private final static int MAX_NO_DETECTIONS = 50;
      /* number of times a marker goes undetected
         before being made invisible */

private ArrayList<MarkerModel> markerModels;
private int numMarkers;

private MultiNyAR top;   // for reporting MarkerModel status info
private NyARDetectMarker detector;

private NyARDoubleMatrix44 transMat = new NyARDoubleMatrix44();
   // transformation matrix for moving a model
```

```
public void updateModels(NyARBufferedImageRaster raster)
{
  int numDetections = getNumDetections(detector, raster);
  try {
    StringBuffer statusInfo = new StringBuffer();

    // find the best detected match for each marker
    for (int mkIdx = 0; mkIdx < numMarkers; mkIdx++) {
      MarkerModel mm = markerModels.get(mkIdx);

      int[] detectInfo =
         findBestDetectedIdx(detector, numDetections, mkIdx);
                          // look for marker mkIdx in image
      int bestDetectedIdx = detectInfo[0];
      double confidence = ((double)detectInfo[1])/CONF_SIZE;
                // (hacky) conversion back to a double

      if (bestDetectedIdx == -1)    // marker not found
        mm.incrNumTimesLost();
      else  {   // marker found
        if (confidence >= MIN_CONF) {
             // detected a marker for mkIdx with high confidence
          mm.resetNumTimesLost();
             // apply transformation from detected marker
             // to the marker's model
          detector.getTransmationMatrix(bestDetectedIdx, transMat);
          mm.moveModel(transMat);
        }
      }

      if (mm.getNumTimesLost() > MAX_NO_DETECTIONS)
                           // marker not detected too many times
        mm.hideModel();    // so make its model invisible

      statusInfo.append(mkIdx + ". " + mm.getNameInfo() +
                           " (" + confidence + ")\n");
      addToStatusInfo(mm, statusInfo);
    }
    top.setStatus( statusInfo.toString());  // display status in GUI
  }
  catch(NyARException e)
  {  System.out.println(e);  }
}  // end of updateModels()
```

One of the quirks of updateModels() is that it first does a quick analysis of the raster image to find the number of detected markers. This is used later to speed up the search for more detailed marker information inside the image.

The number of detected markers is returned by getNumDetections():

```
private int getNumDetections(NyARDetectMarker detector,
                               NyARBufferedImageRaster raster)
{
  int numDetections = 0;
  try {
    synchronized (raster) {
      if (raster.hasBuffer())
        numDetections = detector.detectMarkerLite(raster, 100);
    }
  }
  catch(NyARException e)
  {  System.out.println(e);  }

  return numDetections;
}  // end of getNumDetections()
```

The call to NyARDetectMarker.detectMarkerLite() is surrounded by a synchronized block to prevent the image being updated by the Java 3D behavior while it is being examined here.

The number of MarkerModels is stored in numMarkers, which is used by updateModels()'s for-loop to increment a marker index, mkIdx. findBestDetectedIdx() is called to find the most likely marker in the image that matches that particular marker.

```
// global
private final static int CONF_SIZE = 1000;
          // for converting confidence level double <--> integer


private int[] findBestDetectedIdx(NyARDetectMarker detector,
                             int numDetections, int markerIdx)
{ int iBest = -1;
  double confBest = -1;

  for (int i = 0; i < numDetections; i++) { //check detected markers
    int codesIdx = detector.getARCodeIndex(i);
    double conf = detector.getConfidence(i);

    if ((codesIdx == markerIdx) && (conf > confBest)) {
      iBest = i;  // detected marker index with highest confidence
      confBest = conf;
    }
  }

  /* return best detected marker index, and its confidence
     value as an integer */
  int[] detectInfo = {iBest, (int)(confBest*CONF_SIZE)};
  return detectInfo;
}  // end of findBestDetectedIdx()
```

I'm a little bit embarrassed by the hacky nature of findBestDetectedIdx() which returns two values in an integer array – the first is the index of the best marker in the image (iBest), and the second is its confidence value, converted from a double (confBest) to an integer. The conversion multiples confBest by CONF_SIZE and then casts it to an integer, an unnatural transformation that has to be subsequently reversed (with some loss of precision) back in updateShapes().

If a suitable marker is not found in the image then iBest will be -1, a situation that is tested for in updateShapes(), and results in an increment of the MarkerModel's numTimesLost counter (via a call to MarkerModel.incrNumTimesLost()).

If a marker has been found, we must still check its confidence value to see if it exceeds some arbitrary minimum (MIN_CONF). If it doesn't then no transformation is applied to the model, meaning that it stays in its current position.

If the confidence level exceeds MIN_CONF then NyARDetectMarker.getTransmationMatrix() is called to retrieve its transformation matrix, which is passed to the MarkerModel for applying to the model.

At the end of an iteration of updateModels()'s for-loop, the numTimesLost counter for a MarkerModel is checked to see if it's been incremented too many times. In that case, the model is rendered invisible. Also, various details about the model are added to a string in addToStatusInfo() and written to the text box in the GUI.

```
private void addToStatusInfo(MarkerModel mm, StringBuffer statusInfo)
{
  if (!mm.isVisible())
    statusInfo.append(" not visible\n");
  else {   // model is visible, so report position and orientation
    Point3d pos = mm.getPos();
    if (pos != null)
      statusInfo.append("    at (" + pos.x + ", " +
                                  pos.y + ", " + pos.z + ")\n");
    else
      statusInfo.append("    at an unknown position\n");

    Point3d rots = mm.getRots();
    if (rots != null)
      statusInfo.append("    rots (" + rots.x + ", " +
                                    rots.y + ", " + rots.z + ")\n");
    else
      statusInfo.append("    with unknown rotations\n");
  }
}  // end of addToStatusInfo()
```

A model's name, current visibility, position and rotation could be used in more complex ways. For example, by maintaining a history of a model's positions across multiple frames, it would be possible to plot its trajectory and perhaps predict its future location.