Kinect Chapter 3. A Point Cloud for Depths

[**Note**: the code for this chapter is online at http://fivedots.coe.psu.ac.th/~ad/kinect/; only important fragments are described here.]

The PointCloud application renders the Kinect's changing depth map as a point cloud in a Java 3D scene (see Figure 1).

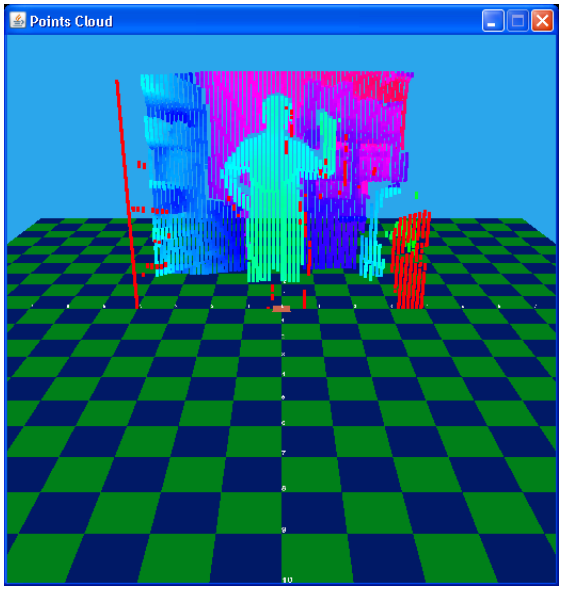

Figure 1. The Depth Map as a Point Cloud.

The points have varying coordinates and colors depending on their depths, which is easier to see if the user's viewpoint is moved, as in Figure 2.

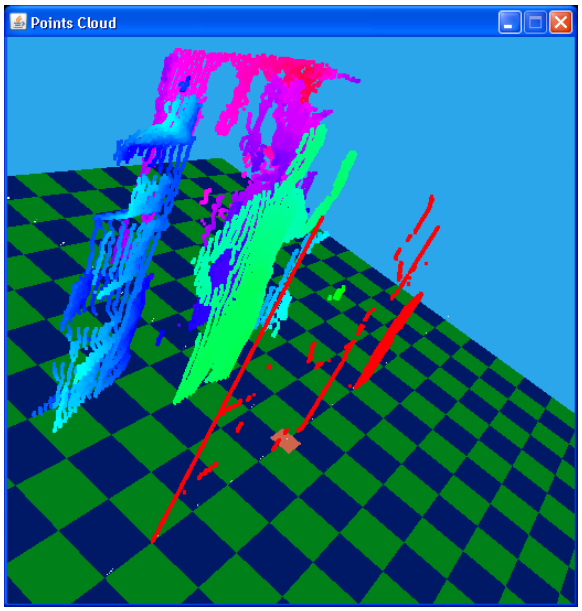

Figure 2. The Depth Map from Another Viewpoint.

I'll be utilizing Java 3D in two chapters altogether, so I'm **not** going to explain all the intricacies of 3D programming. It would take up a lot of space and time, and isn't directly relevant to NUI programming. However, I will overview the more important Java 3D concepts in this chapter, and point you to sources of more information.

For example, almost all of the 3D scene in Figures 1 and 2, aside from the point cloud, is based on the Checkers3D example in chapter 15 of my book "Killer Game Programming in Java", published by O'Reilly. Of course, I'd love you to buy a copy, but you can also find a draft of that chapter (and almost 20 others on 3D programming) at http://fivedots.coe.psu.ac.th/~ad/jg/. Please read it if my overview here isn't enough.

The class diagrams in Figure 3 show the public data and methods for the PointCloud application.

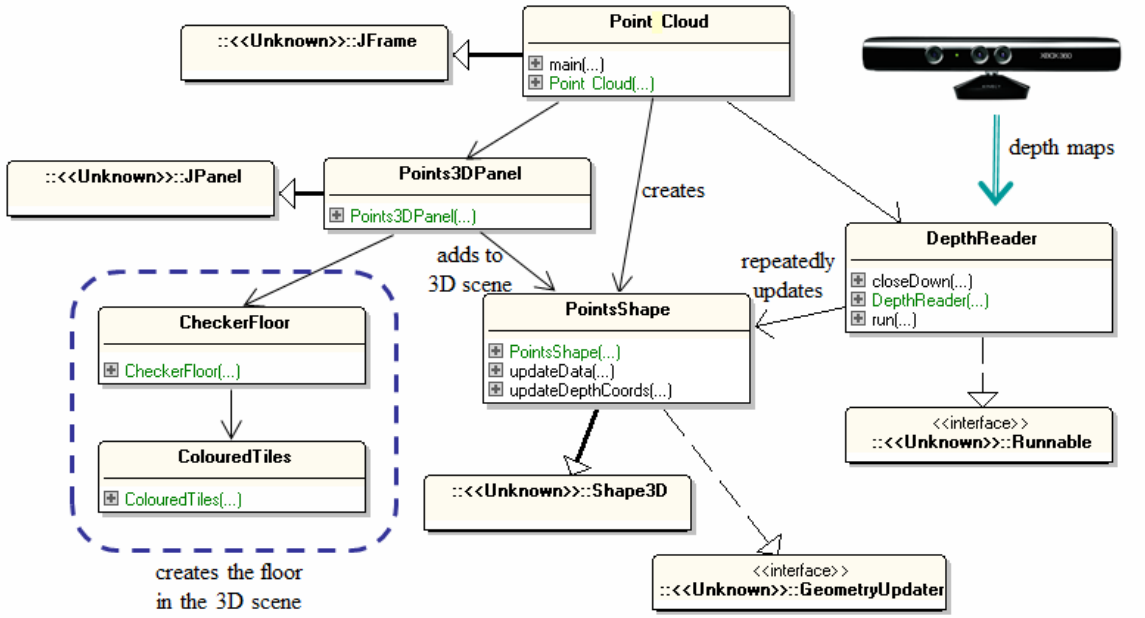

Figure 3. Class Diagrams for PointCloud.

PointCloud is the top-level JFrame for the application, and invokes three important objects:

1. Points3DPanel: a JPanel for displaying the 3D scene, which also creates most of it, apart from the floor and the point cloud.
2. PointsShape: a subclass of Java 3D's Shape3D, which initializes and manages updates to the 3D point cloud.
3. DepthReader: a reader of Kinect depth maps, which passes them over to PointsShape for rendering.

The PointCloud class invokes the three objects like so:

```
PointsShape ptsShape = new PointsShape();
DepthReader depthReader = new DepthReader(ptsShape);
```



```
Points3DPanel panel3d = new Points3DPanel(ptsShape);
```

A reference to the PointsShape object is passed to both DepthReader and Points3DPanel.

CheckerFloor in Figure 3 constructs the 3D floor (the blue and green checkerboard visible in Figures 1 and 2), with the colored tiles drawn by ColouredTiles objects. I won't be describing CheckerFloor and ColouredTiles, since they have nothing to do with the point cloud. Please refer to chapter 15 of "Killer Game Programming in Java" for the gory details.

Before I can explain Points3DPanel, I need to give a quick overview of Java 3D and its scene graph concept.

1. What is Java 3D?

The Java 3D API provides a collection of high-level constructs for creating, rendering, and manipulating a 3D scene composed of geometry, materials, lights, sounds, and more. Java 3D was developed by Sun Microsystems (now, Oracle), and the most recent stable release is version 1.5.2 (http://java3d.java.net/).

Java 3D uses the *scene graph* to organize and manage a 3D application. The underlying graphics pipeline is hidden, replaced 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 (e.g. for a shape) may have node components, specifying color, reflectivity, and other attributes.

The scene graph can contain behaviors – nodes holding code that can affect 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 4 shows how the scene graph idea can be applied to defining the contents of an office.

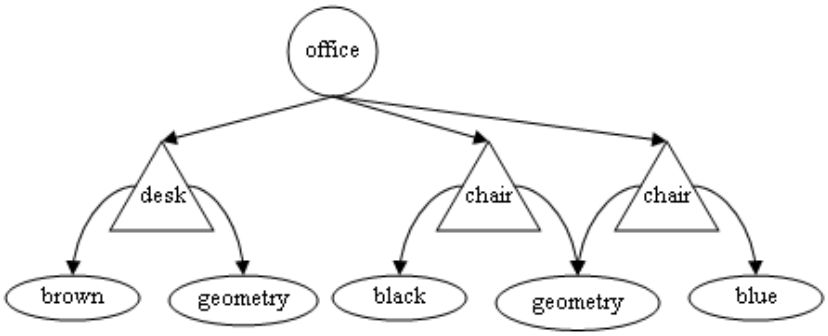

Figure 4. Scene Graph for an Office.

The office Group node is the parent of Leaf nodes representing a desk 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 colored differently.

The choice of symbols in Figure 4 comes from a standard set shown in Figure 5, employed for all my scene graph diagrams. I'll explain the VirtualUniverse and Locale nodes in due course.

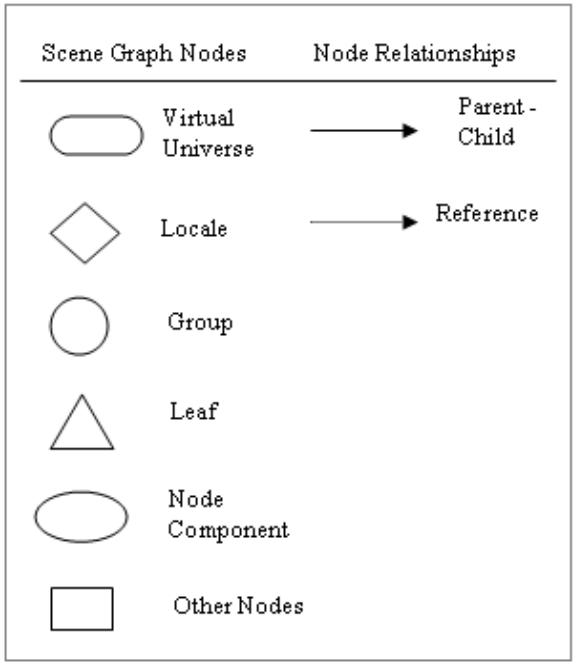

Figure 5. Scene Graph Symbols.

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).

Shape3D is the main shape-related class, utilizing 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, the BranchGroup class allows children to be added or removed from a graph at run time, while TransformGroup permits the position and orientation of its children to be changed.

1.1. The HelloUniverse Scene Graph

The usual first example for Java 3D programmers is HelloUniverse (it appears in Chapter 1 of Sun's Java 3D tutorial). The application displays a rotating colored cube, as in Figure 6.

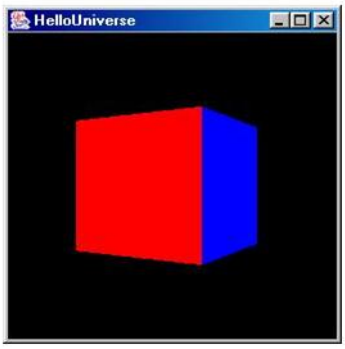

Figure 6. A Rotating Colored Cube.

Its scene graph appears in Figure 7.

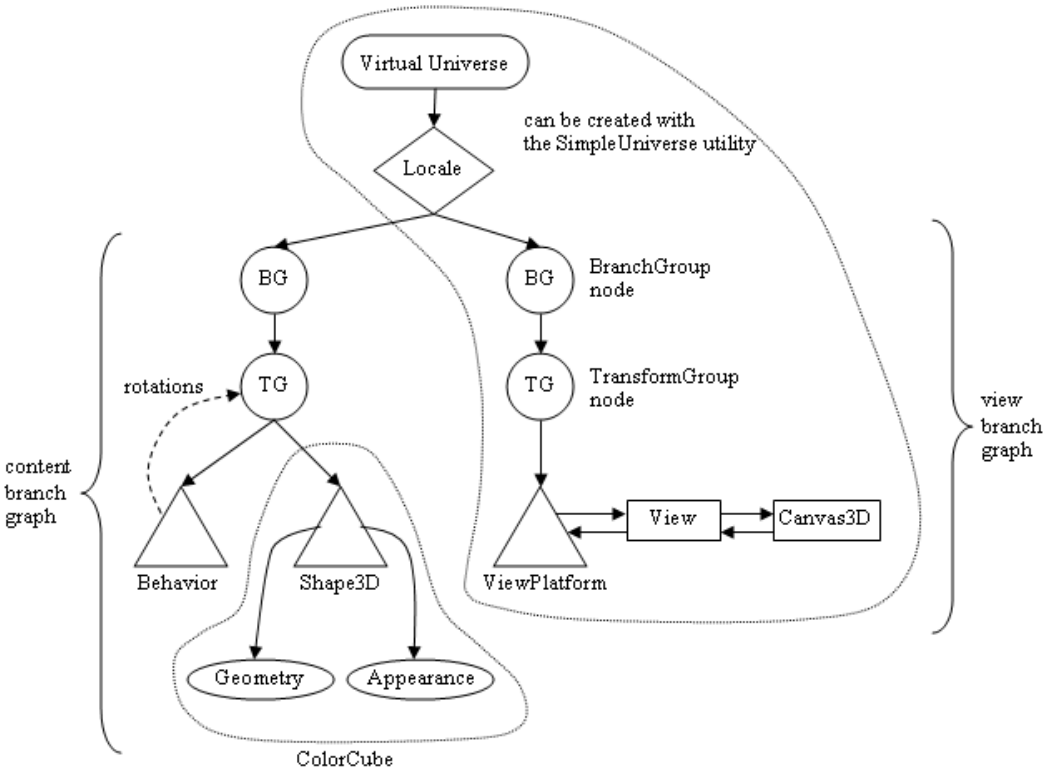

Figure 7. Scene Graph for HelloUniverse.

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. There are always two subgraphs below the Locale node – the *content branch* graph holds 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 with associated Geometry and Appearance components. Its rotation is driven by a Behavior node which affects the TransformGroup parent of the ColorCube's shape.

The other 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 these applications, the relevant nodes can be created with Java 3D's SimpleUniverse utility class, relieving the programmer of a lot of graph construction work.

1.2. Documentation and Examples

The Java 3D distribution comes with over 40 small-to-medium examples. They're a great help, but somewhat lacking in documentation. Fortunately, there's a lot more resources online. Sun's Java 3D tutorial is available at http://java.sun.com/developer/onlineTraining/java3d/. The tutorial is a good introduction to Java 3D but sometimes rather confusing for beginners.

I recommend three Java 3D textbooks as supplemental reading:

- *Killer Game Programming in Java*, by Andrew Davison (i.e. me) (O'Reilly)
- *Computer Graphics Using Java 2D and 3D*, by Hong Zhang and Y. Daniel Liang (Pearson)
- *Java 3D Programming*, by Daniel Selman (Manning)

The Davison text is quite simply amazing. I've never found a better cure for insomnia and constipation (often at the same time).

The Zhang and Liang text is an excellent overview of Java 3D, and includes material on Java 2D which is useful.

The Selman book is more games-oriented, including a Doom-like world, utilizing first-person perspective keyboard navigation, and scene creation from a 2D map.

2. Creating the 3D Scene for PointCloud

After that high-speed tour of Java 3D, I can switch back my PointCloud application and its Points3DPanel class. Points3DPanel starts by initializing a Java 3D drawing surface, a Canvas3D object, which is embedded inside the panel:

```
// globals
private SimpleUniverse su;
private BranchGroup sceneBG;


public Points3DPanel(PointsShape ptsShape)
{
  setLayout( new BorderLayout() );
  setOpaque( false );
  setPreferredSize( new Dimension(PWIDTH, PHEIGHT));

  GraphicsConfiguration config =
            SimpleUniverse.getPreferredConfiguration();
  Canvas3D canvas3D = new Canvas3D(config);
  add("Center", canvas3D);          // add to the panel
  canvas3D.setFocusable(true);      // give focus to the canvas
  canvas3D.requestFocus();

  su = new SimpleUniverse(canvas3D);

  createSceneGraph(ptsShape);
  initUserPosition();         // set user's viewpoint
  orbitControls(canvas3D);    // controls for moving the viewpoint

  su.addBranchGraph( sceneBG );
}   // end of Points3DPanel()
```

The Canvas3D object is initialized with a configuration obtained from SimpleUniverse.getPreferredConfiguration(), which queries the hardware for rendering information.

The su SimpleUniverse object creates a standard view branch graph and the VirtualUniverse and Locale nodes of the scene graph. My createSceneGraph() method sets up the lighting, the sky background, the floor, and positions the point cloud, while initUserPosition() and orbitControls() handle viewer issues. The resulting BranchGroup is added to the scene graph at the end of the constructor.

In other chapters, most of my panel drawing classes start a thread whose run() method performs an update/draw/sleep loop repeatedly. No such coding is needed in Points3DPanel since Java 3D contains it's own threaded mechanism for monitoring changes in the scene and initiating rendering.

The code for createSceneGraph():

```
// globals
private BranchGroup sceneBG;
private BoundingSphere bounds;


private void createSceneGraph(PointsShape ptsShape)
{
  sceneBG = new BranchGroup();  // top of content branch
```



```
  bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

  lightScene();          // add the lights
  addBackground();       // add the sky
  sceneBG.addChild( new CheckerFloor().getBG() );  // add floor

  addPointsShape(ptsShape);  // add the point cloud

  sceneBG.compile();   // fix the scene
} // end of createSceneGraph()
```

Various methods add subgraphs to sceneBG to build up the content branch graph. sceneBG is compiled once the graph has been finalized, to allow Java 3D to optimize it. The optimizations may involve reordering the graph, and regrouping and combining nodes.

bounds is a global BoundingSphere used to specify the influence of environment nodes for lighting, background, and the OrbitBehavior object. The bounding sphere is placed at the center of the scene, and affects everything within a BOUNDSIZE units radius.

The scene graph by the end of Points3DPanel() is shown in Figure 8.

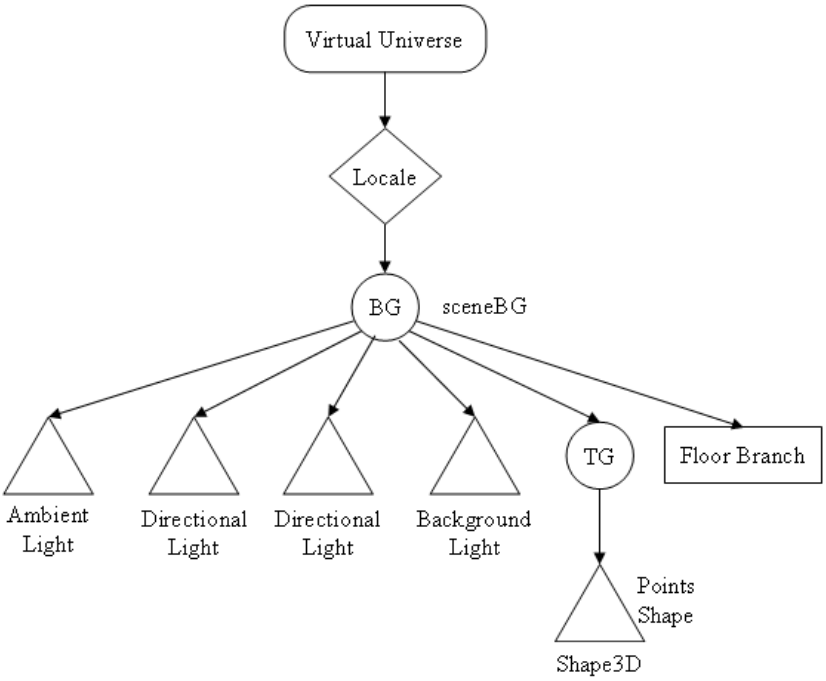

Figure 8. Partial Scene Graph for PointCloud.

BG and TG are short forms for BranchGroup and TransformGroup. The "Floor Branch" is more complicated than shown, but I've drawn it as a box to hide unnecessary details. Figure 8 is also missing the view branch part of the scene graph, which I'll get to in Figure 10.

2.1. Lighting the Scene

One ambient and two directional lights are added to the scene by lightScene(). An ambient light reaches every corner of the world, illuminating everything equally.

```
Color3f white = new Color3f(1.0f, 1.0f, 1.0f);
// Set up the ambient light
AmbientLight ambientLightNode = new AmbientLight(white);
ambientLightNode.setInfluencingBounds(bounds);
sceneBG.addChild(ambientLightNode);
```

The color of the light is set, the ambient source is created along with bounds, and added to the scene. The Color3f() constructor takes Red/Green/Blue values between 0.0f and 1.0f (1.0f being 'full on').

A directional light mimics a light from a distant source, hitting the surfaces of objects from a specified direction. The main difference from an ambient light is the requirement for a direction vector.

```
Vector3f light1Direction  = new Vector3f(-1.0f, -1.0f, -1.0f);
   // left, down, backwards
DirectionalLight light1 =  new DirectionalLight(white,
light1Direction);
light1.setInfluencingBounds(bounds);
sceneBG.addChild(light1);
```

The direction is the vector between (0, 0, 0) and (-1, -1, -1); the light can be imagined to be multiple parallel lines with that direction, originating at infinity.

Point and spot lights are the other forms of Java 3D lighting. Point lights position the light in space, emitting in all directions. Spot lights are focused point lights, aimed in a particular direction.

2.2. The Scene's Background

A background for a scene can be specified as a constant color (as here), a static image, or a texture-mapped geometry such as a sphere:

```
Background back = new Background();
back.setApplicationBounds( bounds );
back.setColor(0.17f, 0.65f, 0.92f);    // sky color
sceneBG.addChild( back );
```

2.3. Positioning the Points Cloud

Positioning a shape is almost always done by placing it's Shape3D node below a TransformGroup (see the Shape3D in Figure 8). A TransformGroup can be used to position, rotate, and scale the nodes which lie beneath it, with the transformations defined with Java 3D Transform3D objects:

```
TransformGroup posnTG = new TransformGroup();
Transform3D t3d = new Transform3D();
// t3d.setScale(0.5);
t3d.setTranslation( new Vector3d(-4.0f, 0.0f, 0.0f) );
```



```
posnTG.setTransform(t3d);
posnTG.addChild(ptsShape);
sceneBG.addChild( posnTG );
```

Unlike some 3D drawing packages, the y-axis in Java 3D is in the vertical direction, while the 'ground' is defined by the XZ plane, as in Figure 9.

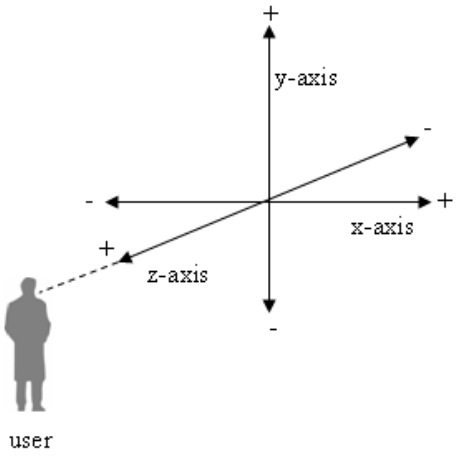

Figure 9. Axes in Java 3D

The position of the point cloud is set to be (-4, 0, 0), which places the midpoint of its base roughly at (0, 0, 0) (as shown in Figure 1).

2.4. Viewer Positioning

The scene graph in Figure 8 doesn't include a view branch graph; that branch is shown in Figure 10.

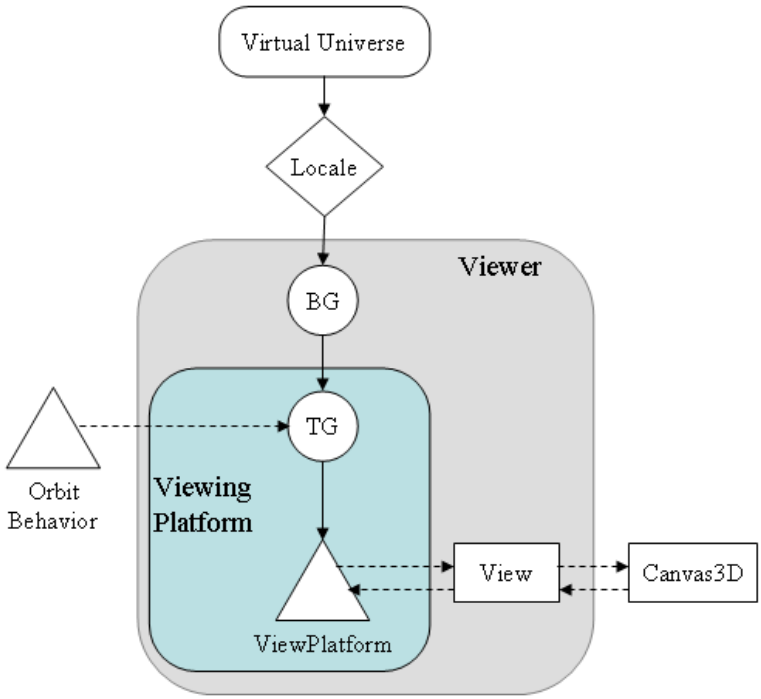

Figure 10. The View Branch Graph.

The branch is created by a call to the SimpleUniverse constructor in the Points3DPanel() constructor:

```
su = new SimpleUniverse(canvas3D);
```

SimpleUniverse offers simplified access to the view branch graph via the ViewingPlatform and Viewer classes, which are mapped to the graph (shown as rounded rectangles in Figure 10).

ViewingPlatform is employed in initUserPosition() to access the TransformGroup above the ViewPlatform node:

```
ViewingPlatform vp = su.getViewingPlatform();
TransformGroup steerTG = vp.getViewPlatformTransform();
```

steerTG corresponds to the TG node in Figure 10. Its Transform3D component is extracted and changed with the lookAt() and invert() methods:

```
Transform3D t3d = new Transform3D();
steerTG.getTransform(t3d);
t3d.lookAt( USERPOSN, new Point3d(0,0,0), new Vector3d(0,1,0));
t3d.invert();
steerTG.setTransform(t3d);
```

lookAt() is a convenient way to set the viewer's position in the virtual world. The method requires the viewer's intended position, the point that he is looking at, and a

vector specifying the upward direction. In this application, the viewer's position is USERPOSN (the coordinate (0, 7, 17)); he is looking towards the origin (0, 0, 0), and 'up' is along the positive y-axis, as illustrated by Figure 11.

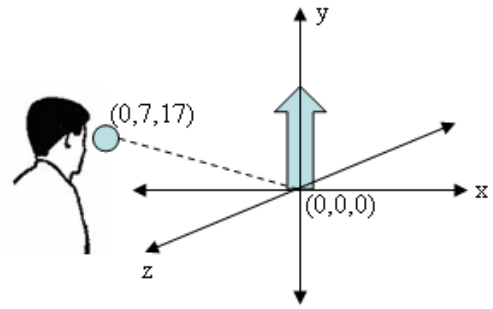

Figure 11. lookAt() Depicted Graphically.

invert() is required since the position is relative to the viewer rather than an object in the scene.

2.5. Viewer Movement

The user can move through the scene by utilizing the Java 3D OrbitBehavior utility class in the view graph. A combination of control keys and mouse button presses move and rotates (or orbits) the viewer's position.

The behavior is set up in orbitControls() in Points3DPanel:

```
OrbitBehavior orbit = new OrbitBehavior(canvas3D,
                                  OrbitBehavior.REVERSE_ALL);
orbit.setSchedulingBounds(bounds);
ViewingPlatform vp = su.getViewingPlatform();
vp.setViewPlatformBehavior(orbit);
```

The REVERSE_ALL flag ensures that the viewpoint moves in the same direction as the mouse.

There are numerous other flags and methods for affecting rotation, translation, and zooming characteristics, explained in the OrbitBehavior class documentation.

3. Points Clouds in Java 3D

PointsShape is a Java 3D shape (a Shape3D subclass) which is drawn as a collection of colored points derived from the Kinect's current depth map.

Internally, PointsShape stores its points as a Java 3D PointArray data structure, with each point represented by a (x, y, z) coordinate and a RGB color. The coordinates and colors are stored in separate arrays, and the PointArray is linked to them *by reference*, meaning that it doesn't hold copies of the points data itself. Only the arrays need to be

updated when the depth map changes, and the PointArray will be automatically changed and redrawn.

The trickiest aspect of PointsShape is converting a depth map into 3D coordinates. The (x, y) pixel positions in the map are modified to become (x, y) coordinates so the point cloud rests on the XZ plane (see Figure 1). The map's depth values are transformed into negative z- axis values, so the points stretch out along the z-axis away from the user's viewpoint.

The coloring of points is based on their depth (i.e. their z-axis value), and uses a color map generated by the ColorUtils library (http://code.google.com/p/colorutils/), in a similar way to version 5 of the ViewerPanel from the last chapter.

3.1. Initializing the PointArray

The PointsShape constructor creates a reference version of the PointArray:

```
// globals
// dimensions of the depth map
private static final int IM_WIDTH = 640;
private static final int IM_HEIGHT = 480;

/* display volume for points inside the 3D scene;
  arrived at by trial-and-error testing to see
  what looked 'good' in the scene
*/
private static final int X_WIDTH = 8;
private static final int Y_WIDTH = 6;
private static final int Z_WIDTH = 5;

private static final int MAX_POINTS = 32000;
private static final int NUM_COLORS = 256;


private PointArray pointParts;   // Java 3D points

// for coloring the points (created with the ColorUtils library)
private Color[] colorMap;

private float xScale, yScale, zScale;
    // scaling from Kinect coords to 3D scene coords

private Semaphore sem;



public PointsShape()
{
  // BY_REFERENCE PointArray storing coordinates and colors
  pointParts = new PointArray(MAX_POINTS,
         PointArray.COORDINATES | PointArray.COLOR_3 |
                                  PointArray.BY_REFERENCE );

  // the PointArray can be read and written at run time
  pointParts.setCapability(GeometryArray.ALLOW_REF_DATA_READ);
  pointParts.setCapability(GeometryArray.ALLOW_REF_DATA_WRITE);

  colorMap = ColorUtils.getSpectrum(NUM_COLORS, 0, 1);
```



```
  // calculate x- and y- scaling from Kinect coords to
  // 3D scene coords
  xScale = ((float)X_WIDTH)/IM_WIDTH;
  yScale = ((float)Y_WIDTH)/IM_HEIGHT;

  sem = new Semaphore(0);

  // create PointsShape geometry and appearance
  createGeometry();
  createAppearance();
} // end of PointsShape()
```

The constructor calculates scaling factors to convert x- and y- coordinates in the depth map into coordinates in the 3D scene.

createGeometry() creates and initializes the coords[] and colors[] arrays for the points.

```
// globals
private static final int SAMPLE_FREQ = 10;
         // the gap between depth positions being sampled

private float[] coords, colors;
         // holds (x,y,z) and (R,G,B) of the points



private void createGeometry()
{
  if (MAX_POINTS*SAMPLE_FREQ < IM_WIDTH*IM_HEIGHT) {
    System.out.println("Warning: coords[] is too small -");
    System.out.println("  some depth info will be lost");
  }

  coords = new float[MAX_POINTS*3];   // for (x,y,z) coords
  colors = new float[MAX_POINTS*3];   // to store points color (RGB)

  // initialize the two arrays
  int ptsCount = 0;
  for (int dpIdx=0; dpIdx < IM_WIDTH*IM_HEIGHT; dpIdx++) {
    if (dpIdx%SAMPLE_FREQ == 0) {
      // only look at depth index that is to be sampled
      int ptIdx = (dpIdx/SAMPLE_FREQ)*3;   // calc point index
      if (ptIdx < MAX_POINTS*3) {   // is there enough space?
        Point pt = depthIdx2Point(dpIdx);
        coords[ptIdx] = pt.x * xScale;     // x coord
        coords[ptIdx+1] = pt.y * yScale;   // y coord
        coords[ptIdx+2] = 0f;              // z coord (changes later)

        // initial point color is white (changes later)
        colors[ptIdx] = 1.0f;
        colors[ptIdx+1] = 1.0f;
        colors[ptIdx+2] = 1.0f;

        ptsCount++;
      }
    }
  }
  System.out.println("Initialized " + ptsCount + " points");
```



```
  // link the coordinates and colors to the PointArray
  pointParts.setCoordRefFloat(coords);    // use BY_REFERENCE
  pointParts.setColorRefFloat(colors);

  /* PointsShape is drawn as the collection
     of colored points stored in the PointArray. */
  setGeometry(pointParts);   // set the geometry of this Shape3D
}  // end of createGeometry()
```

The simplest way of generating the coords[] array would be to map *every* pixel in the depth map to a coordinate. Unfortunately, this would produce IM_WIDTH*IM_HEIGHT (307,200) points, requiring about 1.8 million floats in the two arrays (6 * 307,200). To reduce the burden, only a selection of pixels are converted to coordinates, using the SAMPLE_FREQ constant to space out the sampling.

All the z-coordinates are set to 0 initially, but these will change later as the map is filled with different depths. The colors[] array is set to white, but will also vary with the depths.

The mapping of the map's (x, y) pixel position to scene coordinates is handled by depthIdx2Point(). The depth map is a linear buffer, so its index can be combined with the map's dimensions (IM_WIDTH and IM_HEIGHT) to generate (x, y) values.

```
// globals
// dimensions of the depth image
private static final int IM_WIDTH = 640;
private static final int IM_HEIGHT = 480;


private Point depthIdx2Point(int depthIdx)
/* convert index position in 1D depth buffer into
   2D (x,yUp) point position, assuming IM_WIDTH*IM_HEIGHT
   image dimensions, with x- axis across and y-axis up
*/
{
  int x = depthIdx%IM_WIDTH;
  int yUp = (IM_HEIGHT-1) - (depthIdx/IM_WIDTH);
     // so y values increase up the axis
  return new Point(x, yUp);
}  // end of depthIdx2Point()
```

The y-axis direction is 'flipped' because the depth map's y-axis runs downwards whereas the 3D scene's y-axis runs upwards (see Figure 9).

Compared to createGeometry(), createAppearance() has little to do – it sets the rendering size of the points.

```
// global
private final static int POINT_SIZE = 3;


private void createAppearance()
{
  Appearance app = new Appearance();
  PointAttributes pa = new PointAttributes();
  pa.setPointSize( POINT_SIZE );  // fix point size
```



```
  app.setPointAttributes(pa);
  setAppearance(app);   // set the appearance of this Shape3D
}  // end of createAppearance()
```

3.2. Updating the Points

The DepthReader object calls PointsShape.updateDepthCoords() whenever it receives a new depth map from the Kinect. The problem is that PointsShape cannot update the coords[] and colors[] array whenever it wants.

Java 3D executes its own update/redraw rendering cycle in a system thread, which means that it will periodically access the arrays to use their information for drawing the points. Synchronization problems may occur if this examination is intertwined with the arrays being changed by code inside PointsShape.

Synchronization worries can be avoided by having PointsShape implement Java 3D's GeometryUpdater interface, which allows the chosen data structures (in my case, the two arrays) to be updated by the system. The system ensures that the rendering cycle isn't using the arrays when it calls GeometryUpdater.updateData().

Let's look at the update sequence in PointsShape. It starts with DepthReader passing a new depth map to PointsShape.updateDepthCoords():

```
// globals
private static final int Z_WIDTH = 5;

private ShortBuffer depthBuf;      // current Kinect depth map

private float zScale;
    // scaling from Kinect coords to 3D scene coords

private PointArray pointParts;     // Java 3D points

private Semaphore sem;
   /* used to make updateDepthCoords() wait until
      GeometryUpdater.updateData() has finished an update
   */


public void updateDepthCoords(ShortBuffer dBuf)
{
  depthBuf = dBuf;
  zScale = ((float)Z_WIDTH)/getMaxDepth(depthBuf);
                       //adjust z-axis scaling

  pointParts.updateData(this);   // request update of geometry
  try {
    sem.acquire();     // wait for update to finish in updateData()
  }
  catch(InterruptedException e) {}
}  // end of updateDepthCoords()
```

Despite its name, updateDepthCoords() doesn't do any updating. Instead it politely requests that the system calls GeometryUpdater.updateData() when there's no chance of synchronization problems. The request is issued by:


```
pointParts.updateData(this);
```

A short time later, the system will call GeometryUpdater.updateData() (which I'll explain in a moment), and that's where the arrays are updated. updateDepthCoords() should wait until these changes have been carried out before returning.

One way of implementing this waiting is to use a *counting semaphore*. The call to Semaphore.acquire() in updateDepthCoords() makes it block until the semaphore receives a 'permit'.

In the meantime, the system calls GeometryUpdater.updateData(), which is implemented by PointsShape as:

```
// globals
private static final int SAMPLE_FREQ = 10;
         // the gap between depth positions being sampled

private ShortBuffer depthBuf;     // current Kinect depth map

private PointArray pointParts;     // Java 3D points
private float[] coords, colors;
          // holds (x,y,z) and (R,G,B) of points

private Semaphore sem;


public void updateData(Geometry geo)
// this method is called by Java 3D
{
  while (depthBuf.remaining() > 0) {
    int dpIdx = depthBuf.position();
    float zCoord = ((float) depthBuf.get())*zScale;
                              // convert to 3D scene coord
    if (dpIdx%SAMPLE_FREQ == 0) {    // update this z-coord?
      int zCoordIdx = (dpIdx/SAMPLE_FREQ)*3 + 2;
      if (zCoordIdx < coords.length) {
        coords[zCoordIdx] = -zCoord;
            // negate so depths are spread out along -z axis,
            // away from camera
        updateColour(zCoordIdx-2, zCoord);
      }
    }
  }
  sem.release();  // wake-up updateDepthCoords()
}  // end of updateData()
```

The code cycles through the sampled depths in the depth map, and updates the z-coordinates in the coords[] array and the colors in colors[]. When this is finished, Semaphore.release() issues a permit, which the blocked Semaphore.acquire() consumes, waking up updateDepthCoords() so it can finish.

updateColor() uses the red-to-violet color map created back in the constructor to color the points, based on their z-axis values. Points in the foreground, which have small z values are colored red, while those in the background (further along the axis) get a color more towards violet.

Comparing the coloring of Figure 1 with the coloring of the depth map in the previous chapter (e.g. in Figure 20 of chapter 2) reveals that the color schemes are similar but

not identical. The reason is that the coloring in ViewerPanel is applied to accumulated depth values. Something similar could be done here, but coloring the scaled depth values is simpler.

4. Obtaining the Depth Map

DepthReader waits for an update to the Kinect's depth map, then passes it to PointsShape via updateDepthCoords(). It connects to the Kinect in the same way as the ViewerPanel classes from the last chapter.

```
// globals
/* resolution of depth image; the same values
   should be used in PointsShape */
private static final int IM_WIDTH = 640;
private static final int IM_HEIGHT = 480;

// OpenNI
private Context context;
private DepthMetaData depthMD;


private void configOpenNI()
// create context and depth generator
{
  try {
    context = new Context();

    // add the NITE License
    License license = new License("PrimeSense",
                                  "0KOIk2JeIBYClPWVnMoRKn5cdY4=");
    context.addLicense(license);
    DepthGenerator depthGen = DepthGenerator.create(context);
    MapOutputMode mapMode = new MapOutputMode(IM_WIDTH,IM_HEIGHT,30);
                                                  // xRes, yRes, FPS
    depthGen.setMapOutputMode(mapMode);

    // set Mirror mode for all
    context.setGlobalMirror(true);
    context.startGeneratingAll();
    System.out.println("Started context generating...");

    depthMD = depthGen.getMetaData();
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

A hidden dependencies between DepthReader and PointsShape is the depth map's dimensions (specified in IM_WIDTH and IM_HEIGHT). PointsShape must use the same values or it will incorrectly convert the map's pixels to 3D coordinates.

DepthReader employs a thread to wait for the context to be updated.


```
// globals
private volatile boolean isRunning;

// OpenNI
private Context context;
private DepthMetaData depthMD;

private PointsShape ptsShape;
          // renders the depth points in the Java 3D scene


public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAnyUpdateAll();
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }

    ShortBuffer depthBuf = depthMD.getData().createShortBuffer();
    ptsShape.updateDepthCoords(depthBuf);
          // this call will not return until the
          // 3D scene has been updated
  }
  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(0);
}  // end of run()
```

PointsShape.updateDepthCoords() will not return until the point cloud has been updated, so there's no chance that updateDepthCoords() can be called again while the previous map is still being processed by Java 3D.

