Kinect Chapter 5. Viewing Users in 3D

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

This chapter revisits skeletal tracking but this time renders the users in 3D, as shown in Figure 1.

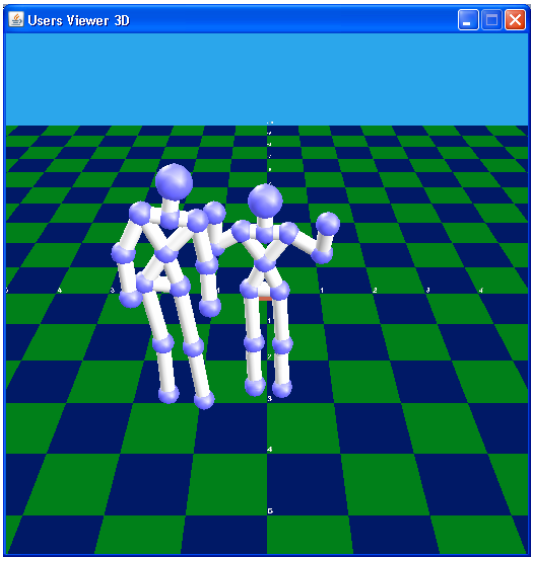

Figure 1. 3D Skeletal Tracking of Two Users.

The OpenNI aspects of this application (called UsersTracker3D) are much the same as in the last chapter – the pose detection and skeletal tracking capabilities of a UserGenerator node are utilized to obtain the users' joint coordinates. In fact, the OpenNI part of the code is simpler than previously because I've no need to collect and render depth map information. The main new features are:

- the joints and the limbs are Java 3D shapes (sphere and cylinders respectively), which move and rotate to match the movement of the users);
- OpenNI observers (listeners) deal with a user temporarily moving out of range of the Kinect sensor and returning (the user's skeleton disappears during that time);
- joints are positioned using averaged coordinate values, collected over several sensor updates. This reduces the 'shuddering' of joints, at the cost of reducing the responsiveness of a skeleton to sudden user movement;
- each joint and limb that be made invisible independently of the rest of the skeleton. This allows the application to deal with joints going out of sensor range by making only those parts of the skeleton invisible;

the 3D scene is essentially unchanged from the Java 3D code used in chapter 3 for the point cloud (i.e. a checkerboard floor, blue sky, lighting, and a moveable camera). Figure 2 shows a user standing facing the Kinect, but with the Java 3D camera rotated to the left.

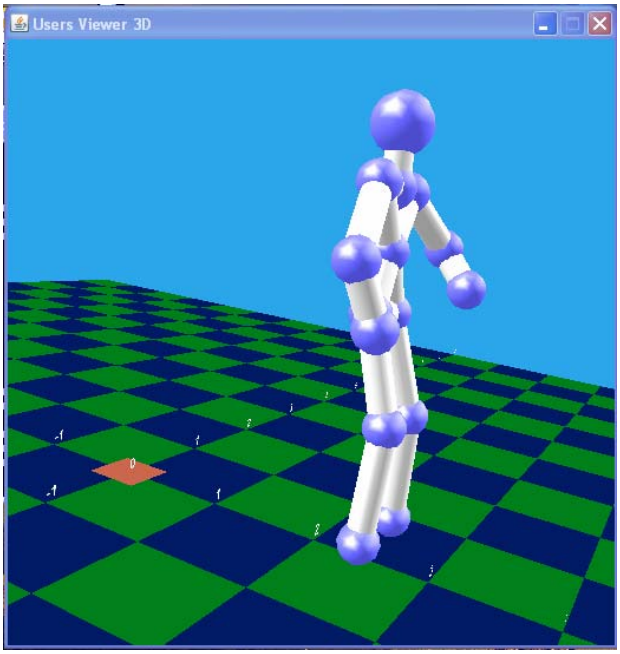

Figure 2. A Single User Viewed from the Left.

Figure 3 shows the class diagrams for the UsersViewer3D application, with only the class names listed.

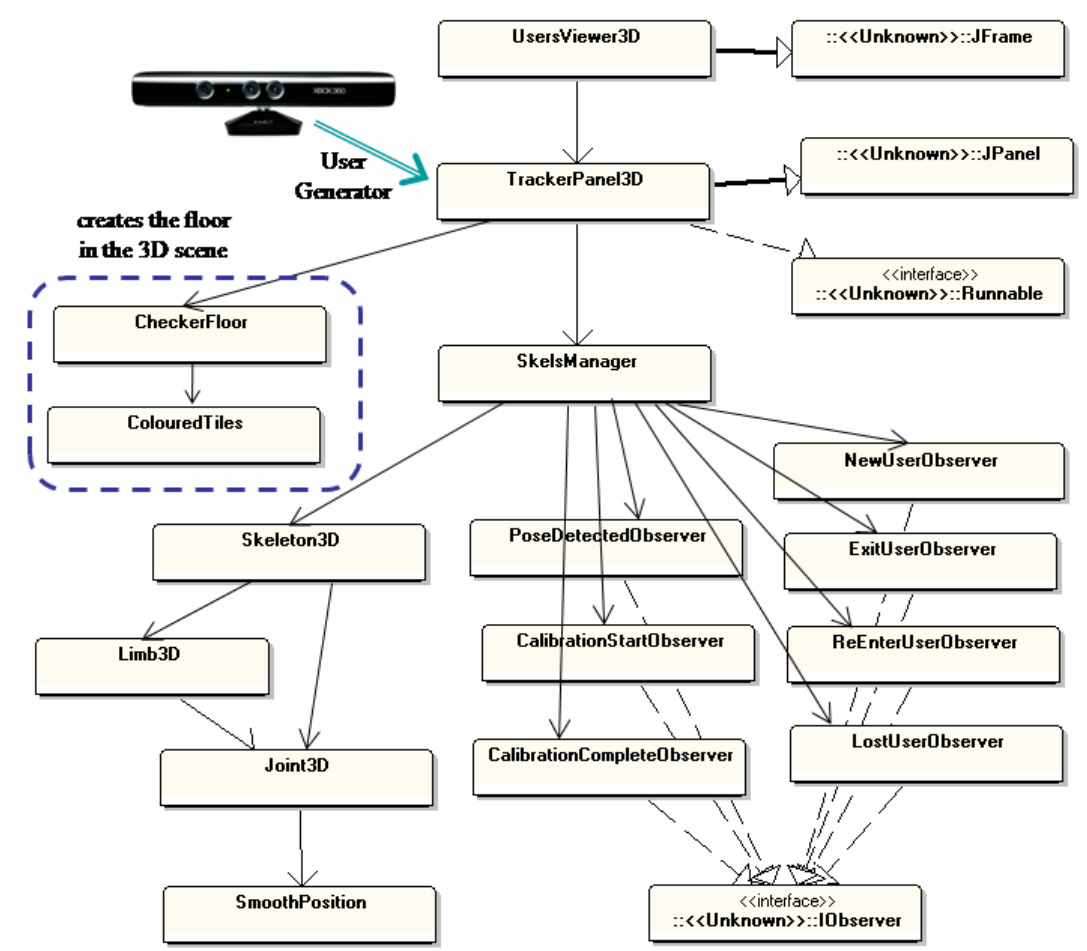

Figure 3. Class Diagrams for UsersViewer3D.

UsersViewer3D builds the application window by employing an instance of TrackerPanel3D as a panel. TrackerPanel3D's main task is to create the Java 3D scene, in a very similar way to Points3DPanel in the points cloud example in chapter 3. It builds the scene graph in Figure 4.

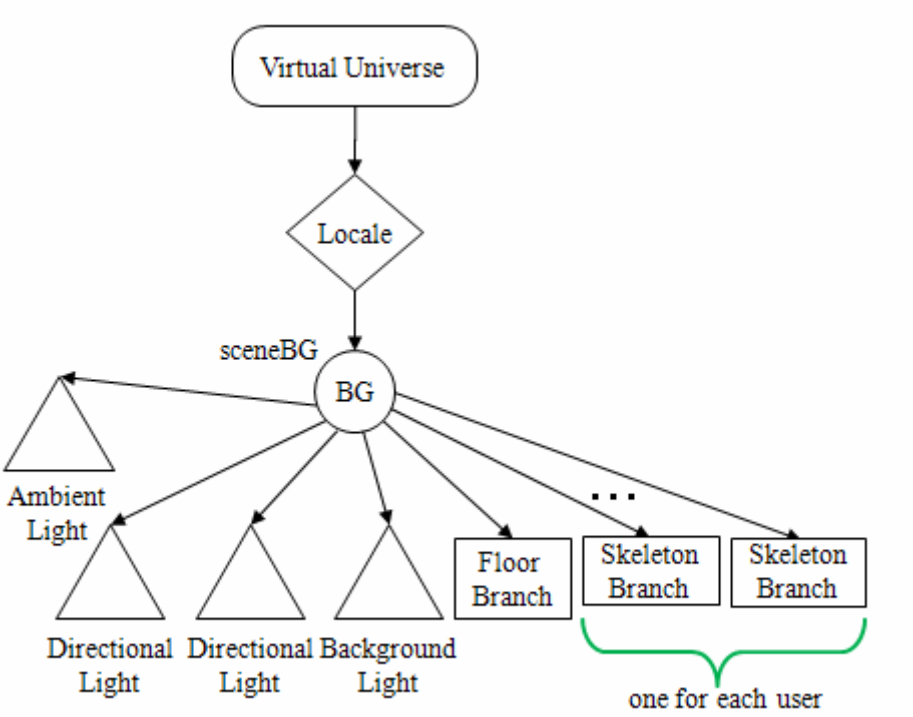

Figure 4. Scene Graph Created by TrackerPanel3D.

TrackerPanel3D employs the CheckerFloor and ColouredTiles classes to create the floor (as in chapter 3), and passes a sceneBG reference to SkelsManager.

TrackerPanel3D also creates a Kinect sensor context and passes user generator information down to SkelsManager, which manages the creation, deletion, and the visibility of user skeletons. Figure 3 shows that SkelsManager uses seven observers, three more than TrackerPanel in the ArniesTracker application. The additional observers are for noticing when a user temporarily exits the scene (ExitUserObserver), when the user re-enters (ReEnterUserObserver), and when skeleton calibration starts (CalibrationStartObserver).

The creation of a 3D Skeleton (a Skeleton Branch box in Figure 4) is handled by a Skeleton3D object, which creates the 3D joints and limbs (i.e. spheres and cylinders) using instances of the Joint3D and Limb3D classes.

One of the novel elements of this application is the use of average joint positions to make the skeletons less prone to shuddering. The averaging is carried out by the SmoothPosition class.

1. The Tracker Panel

TrackerPanel3D has two main tasks: to create the Java 3D scene graph shown in Figure 4 and to set up the OpenNI context.

As in the points cloud example, the scene consists of a dark green and blue tiled surface with labels along the X and Z axes, a blue background, lit from two directions.

The user can guide a camera through the scene by moving the mouse. I won't be explaining these details again; please look back at Points3DPanel in chapter 3.

The new 3D features are the user skeletons; they're managed by the SkelsManager class which I explain below.

The OpenNI context, the UserGenerator node, and a SkelsManager object are created inside TrackerPanel3D.configOpenNI():

```
// globals
private Context context;
private UserGenerator userGen;
private BranchGroup sceneBG;
private SkelsManager skels;   // the skeletons manager


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

    // add the NITE Licence
    License licence = new License("PrimeSense",
                           "0KOIk2JeIBYClPWVnMoRKn5cdY4=");
    context.addLicense(licence);
    context.setGlobalMirror(true);    // set mirror mode

    userGen = UserGenerator.create(context);
    skels = new SkelsManager(userGen, sceneBG);

    context.startGeneratingAll();
    System.out.println("Started context generating...");
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

Depth information isn't needed, so there's no need to create DepthGenerator and DepthMapData objects in configOpenNI().

The SkelsManager object is passed two arguments: the UserGenerator so that its pose detection and skeleton tracking capabilities can be accessed, and the sceneBG BranchGroup so that skeletons can be attached to the scene graph (as in Figure 4).

TrackerPanel3D is threaded so that it can execute a wait-update loop for Kinect sensor information:

```
// globals
private volatile boolean isRunning;
private Context context;
private SkelsManager skels;   // the skeletons manager


public void run()
```



```
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAnyUpdateAll();
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    skels.update();   // skeletons manager carries out updates
  }
  // close down
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(0);
}  // end of run()
```

2. Managing the Skeletons

The SkelsManager class has three main duties:

> it initializes the user generator 's pose and skeleton detection capabilities;
>
> it sets up seven observers (listeners) which manage the creation, deletion and visibility of a user skeleton
>
> it's update() method is periodically called by TrackerPanel3D so the position and orientation of the user skeletons can be refreshed with the Kinect's skeleton data.

2.1. Configuring the UserGenerator

The initialization of the pose and skeleton detection capabilities, and the invocation of the listeners, is carried out by SkelsManager.configure():

```
// globals
private UserGenerator userGen;

// capabilities used by UserGenerator
private SkeletonCapability skelCap;
    // to output skeletal data, including location of joints

private PoseDetectionCapability poseDetectionCap;
 // to recognize when the user is in a specific position

private String calibPose = null;


private void configure()
{
  try {
    // setup UserGenerator pose and skeleton detection caps;
    poseDetectionCap = userGen.getPoseDetectionCapability();
```



```
    skelCap = userGen.getSkeletonCapability();
    calibPose = skelCap.getSkeletonCalibrationPose(); // 'psi' pose
    skelCap.setSkeletonProfile(SkeletonProfile.ALL);

    // set up 7 observers
    userGen.getNewUserEvent().addObserver(new NewUserObserver());
                          // new user found

    userGen.getLostUserEvent().addObserver(new LostUserObserver());
                          // lost a user

    userGen.getUserExitEvent().addObserver(new ExitUserObserver());
                          // user has exited (but may re-enter)

    userGen.getUserReenterEvent().addObserver(
                      new ReEnterUserObserver());
                          // user has re-entered

    poseDetectionCap.getPoseDetectedEvent().addObserver(
                      new PoseDetectedObserver());
                           // for when a pose is detected

    skelCap.getCalibrationStartEvent().addObserver(
                      new CalibrationStartObserver());
                           // calibration is starting

    skelCap.getCalibrationCompleteEvent().addObserver(
                      new CalibrationCompleteObserver());
            // for when skeleton calibration is completed,
            // and tracking starts
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configure()
```

2.2. The Observers

The purpose of the seven observers can best be understood by looking at the stages in Figure 5 that constitute user pose detection and skeleton tracking.

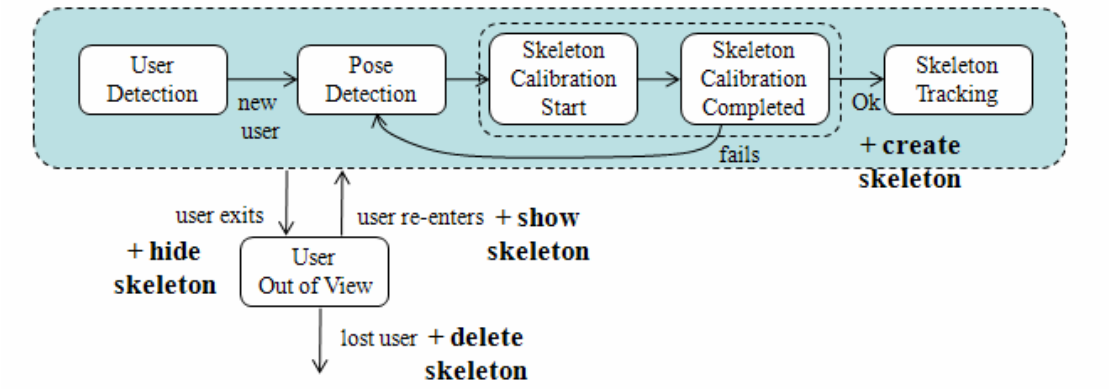

Figure 5. Pose Detection and Skeletal Tracking Stages.

When a new user is detected, the standard user pose is detected so the user's skeleton can be calibrated and tracked.

The start of tracking triggers the creation of a new Java 3D skeleton, and its attachment to the scene graph.

The skeleton is made invisible when the user leaves the Kinect's view, and visible again when it returns. If the user remains out of sight for several seconds, then the user is considered lost and his skeleton is deleted from the scene graph.

A New User

When a new user is detected, the NewUserObserver listener is invoked:

```
// globals
private PoseDetectionCapability poseDetectionCap;
private String calibPose = null;


class NewUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                                          UserEventArgs args)
  {
    System.out.println("Detected new user " + args.getId());
    try {
      // try to detect a pose for the new user
      poseDetectionCap.StartPoseDetection(calibPose, args.getId());
    }
    catch (StatusException e)
    { e.printStackTrace(); }
  }
}  // end of NewUserObserver inner class
```

NewUserObserver starts looking for the pose stored in calibPose (the 'psi' position, where the user has both hands raised).

Pose Detection

When pose detection comes to an end, PoseDetectedObserver is invoked:

```
// globals
private PoseDetectionCapability poseDetectionCap;
private SkeletonCapability skelCap;


class PoseDetectedObserver implements
                        IObserver<PoseDetectionEventArgs>
{
  public void update(IObservable<PoseDetectionEventArgs> observable,
                                  PoseDetectionEventArgs args)
  {
```



```
    int userID = args.getUser();
    System.out.println(args.getPose() +
               " pose detected for user " + userID);
    try {
      // finished pose detection; switch to skeleton calibration
      poseDetectionCap.StopPoseDetection(userID);
      skelCap.requestSkeletonCalibration(userID, true);
    }
    catch (StatusException e)
    {  e.printStackTrace(); }
  }
}  // end of PoseDetectedObserver inner class
```

The pose detector is stopped, and skeleton calibration started.

The Start of Skeleton Calibration

When calibration commences, CalibrationStartObserver is called:

```
class CalibrationStartObserver implements
                          IObserver<CalibrationStartEventArgs>
{
  public void update(
           IObservable<CalibrationStartEventArgs> observable,
                                 CalibrationStartEventArgs args)
  { System.out.println("Calibration started for user " +
                                                 args.getUser());  }
}  // end of CalibrationStartObserver inner class
```

To be honest, this method does nothing useful; I included it to show how skeletal calibration can be monitored.

The End of Skeletal Calibration

When the calibration comes to a close, CalibrationCompleteObserver is invoked:

```
// globals
private SkeletonCapability skelCap;

private HashMap<Integer, Skeleton3D> userSkels3D;
    // maps user IDs --> a 3D skeleton

private BranchGroup sceneBG;    // the scene graph


class CalibrationCompleteObserver implements
                        IObserver<CalibrationProgressEventArgs>
{
  public void update(
             IObservable<CalibrationProgressEventArgs> observable,
                                CalibrationProgressEventArgs args)
  {
    int userID = args.getUser();
    System.out.println("Calibration status: " +
                args.getStatus() + " for user " + userID);
```



```
    try {
      if (args.getStatus() == CalibrationProgressStatus.OK) {
        // calibration succeeeded; move to skeleton tracking
        System.out.println("Starting tracking user " + userID);
        skelCap.startTracking(userID);

        // create skeleton3D in userSkels3D, and add to scene
        Skeleton3D skel = new Skeleton3D(userID, skelCap);
        userSkels3D.put(userID, skel);
        sceneBG.addChild( skel.getBG() );
      }
      else   // calibration failed; return to pose detection
        poseDetectionCap.StartPoseDetection(calibPose, userID);
    }
    catch (StatusException e)
    {  e.printStackTrace(); }
  }
}  // end of CalibrationCompleteObserver inner class
```

As Figure 5 indicates, there are two paths out of this state: if the calibration was successful then skeletal tracking is started by calling SkeletonCapability.startTracking(). Calibration failure means a return to pose detection for another attempt at finding the psi pose.

When tracking starts, a 3D skeleton is created by instantiating a Skeleton3D object. This is stored in a HashMap, using the user ID as a key. A reference to the top BranchGroup node of the skeleton graph is obtained via Skeleton3D.getBG(), and attached to the scene as a child of sceneBG (see Figure 4 for an illustration).

A User Exits

If the Kinect loses track of a user, OpenNI wakes up the ExitUserObserver:

```
// global
private HashMap<Integer, Skeleton3D> userSkels3D;


class ExitUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                                          UserEventArgs args)
  {
    int userID = args.getId();
    System.out.println("Exit of user " + userID);

    // make 3D skeleton invisible when user exits
    Skeleton3D skel = userSkels3D.get(userID);
    if (skel == null)
      return;
    skel.setVisibility(false);
  }
} // end of ExitUserObserver inner class
```

The user's 3D skeleton is made invisible by calling Skeleton3D.setVisibility(). The correct Skeleton3D object is found by looking in the userSkels3D HashMap using the user ID as a key.

A User Re-enters

If the user reappears within a certain time period, then ReEnterUserObserver is called:

```
// global
private HashMap<Integer, Skeleton3D> userSkels3D;

class ReEnterUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                                          UserEventArgs args)
  {
    int userID = args.getId();
    System.out.println("Reentry of user " + userID);

    // make 3D skeleton visible when user re-enters
    Skeleton3D skel = userSkels3D.get(userID);
    if (skel == null)
      return;
    skel.setVisibility(true);
  }
} // end of ReEnterUserObserver inner class
```

ReEnterUserObserver looks up the user's Skeleton3D object and makes it visible.

A User is Lost

If a user stays out of range of the Kinect for too long, then LostUserObserver is woken up:

```
// global
private HashMap<Integer, Skeleton3D> userSkels3D;


class LostUserObserver implements IObserver<UserEventArgs>
{
  public void update(IObservable<UserEventArgs> observable,
                                          UserEventArgs args)
  {
    int userID = args.getId();
    System.out.println("Lost track of user " + userID);

    // delete skeleton from userSkels3D and the scene graph
    Skeleton3D skel = userSkels3D.remove(userID);
    if (skel == null)
      return;
    skel.delete();
  }
} // end of LostUserObserver inner class
```

The user's Skeleton3D object is removed from the userSkels3D HashMap, and Skeleton3D.delete() called. As we'll see, delete() detaches the skeleton's subgraph from the scene.

2.3. Updating the Skeletons

CalibrationCompleteObserver starts tracking a new user by creating a UserID/Skeleton3D entry in the userSkel3D HashMap. When SkelsManager.update() is called, it's task is to update all those skeletons which are still being tracked.

```
// globals
private UserGenerator userGen;
private SkeletonCapability skelCap;
private HashMap<Integer, Skeleton3D> userSkels3D;


public void update()
// update skeleton of each user being tracked
{
  try {
    int[] userIDs = userGen.getUsers();   // may be many users
    for (int i = 0; i < userIDs.length; i++) {
      int userID = userIDs[i];
      if (skelCap.isSkeletonCalibrating(userID))
        continue;
      if (skelCap.isSkeletonTracking(userID))
        userSkels3D.get(userID).update();
    }
  }
  catch (StatusException e)
  {  System.out.println(e); }
}  // end of update()
```

The UserGenerator is queried to obtain a list of IDs, and the SkeletonCapability object is employed to find which of those are being tracked, and Skeleton3D.update() is called for their skeletons.

3. Bringing a Skeleton to Life

The primary tasks of a Skeleton3D object are to:

create a scene graph for a skeleton;

update the position and orientation of the skeleton when update() is called;

make the skeleton invisible or visible (when a user exits or re-enters the scene);

delete the skeleton from the scene when the user is deemed lost.

3.1. Creating a Skeleton Scene Graph

The skeletons in Figures 1 and 2 appear quite complex, but they're only made from two types of shape, spheres representing skeletal joints, and cylinders for the limbs connecting the joints.

The joints and limbs are represented by Joint3D and Limb3D objects, which manage the creation of the Java 3D spheres and cylinders, and their movement and rotation.

The scene graph for a skeleton is shown in Figure 6.

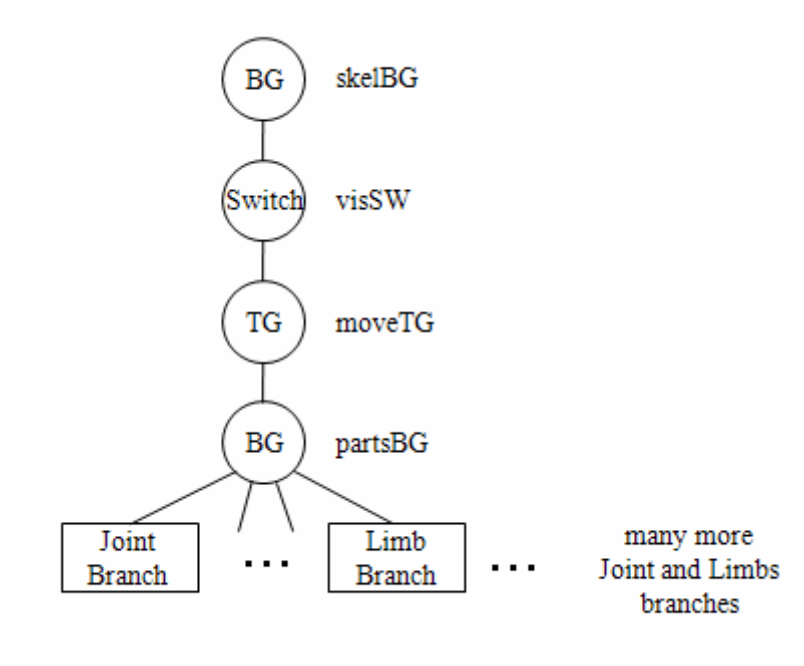

Figure 6. The Scene Graph for a Skeleton.

Figure 6 corresponds to a "Skeleton Branch" box in Figure 4, with the skelBG BranchGroup being connected to the sceneBG node of the 3D scene.

The Switch node (called visSW) controls the rendering of the subgraph below it, allowing the skeleton to be made invisible.

The moveTG TransformGroup is employed to position the skeleton so its 'feet' rest on the checkerboard floor.

The partsBG BranchGroup is used to group the limbs and joint subgraphs. The details of "Joint Branch" and "Limb Branch" will be explained when I discuss the Joint3D and Limb3D classes.

It is possible to use less graph nodes to create the skeleton. However, multiple nodes allow a cleaner separation of concerns, with one node doing one task (e.g. visibility, positioning, grouping). The subgraph is compiled after it is constructed, which allows Java 3D to optimize the graph, perhaps combining several nodes together.

The number of joint and limb branches depends on the skeletal model. I'll be reusing the one from the previous chapter, shown again in Figure 7.

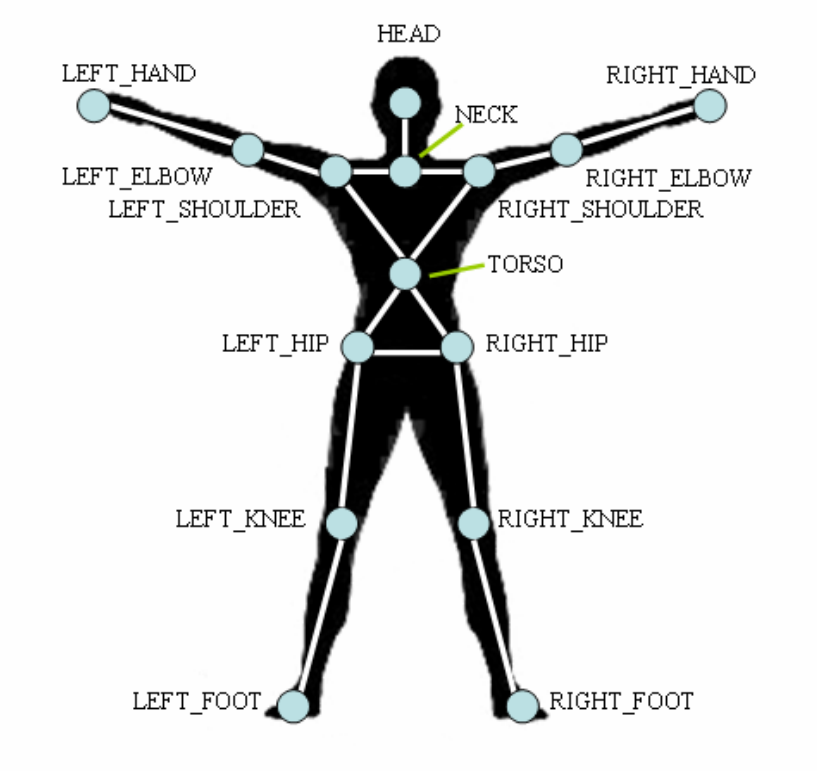

Figure 7. The Skeletal Model of Joints and Limbs.

Figure 7 utilizes 15 joints (the circles) and 16 limbs (the lines connecting the circles), replicated in the code with 15 Joint3D and 16 Limb3D objects. The resulting scene graph for a skeleton appears in Figure 8.

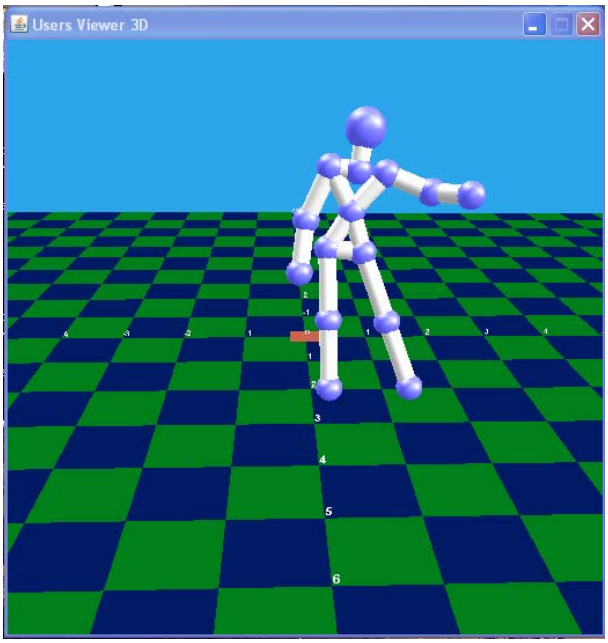

Figure 8. A Java 3D Skeleton.

Note that the head joint in Figure 8 is a little larger than the other joints to make it look more like a head.

The Skeleton3D constructor builds the scene graph in Figure 6.

```
// globals
// collections of joints and limbs making up the skeleton
private ArrayList<Joint3D> joints3D;
private ArrayList<Limb3D> limbs;


public Skeleton3D(int userID, SkeletonCapability skelCap)
{
  // create top of scene graph for the skeleton
  BranchGroup partsBG = buildSkelGraph();
       // all the joints and limbs are attached to partsBG

  HashMap<SkeletonJoint, Joint3D> jointsMap =
                      new HashMap<SkeletonJoint, Joint3D>();
      // used for accessing joints when I create the limbs

  // build joints
  joints3D = new ArrayList<Joint3D>();
  buildJoints(joints3D, userID, skelCap, partsBG, jointsMap);

  // build limbs
  limbs = new ArrayList<Limb3D>();
  buildLimbs(limbs, partsBG, jointsMap);

  skelBG.compile();
} // end of Skeleton3D()
```

Three data structures are created in the constructor – a list of Joint3D objects (joints3D) for the skeleton's joints, a list of Limb3D objects (limbs) for the skeleton's limbs, and a HashMap. The HashMap is a temporary structure which maps OpenNI SkeletonJoints to their corresponding Joint3D objects. The map is built by buildJoints(), and used inside buildLimbs().

The three highlighted methods in the constructor are buildSkelGraph(), buildJoints() and buildLimbs(). buildSkelGraph() creates the top part of the scene graph in Figure 6, buildJoints() creates the Joint branches, and buildLimbs() the Limb branches.

The buildSkelGraph() code:

```
// globals
private static final float Y_OFFSET = 2.5f;
private static final float Z_OFFSET = 5.0f;

private BranchGroup skelBG;      // top of the skeleton graph
private Switch visSW;            // for skeleton visibility
private boolean isVisible;


private BranchGroup buildSkelGraph()
{
  // skelBG-->visSW-->moveTG-->partsBG
  BranchGroup partsBG = new BranchGroup();
  partsBG.setCapability(BranchGroup.ALLOW_CHILDREN_READ);
  partsBG.setCapability(BranchGroup.ALLOW_CHILDREN_WRITE);

  Transform3D t3d = new Transform3D();
```



```
  t3d.set(new Vector3f(0, Y_OFFSET, Z_OFFSET));   // so feet on floor
  TransformGroup moveTG = new TransformGroup(t3d);
  moveTG.addChild(partsBG);

  // create switch for visibility
  visSW = new Switch();
  visSW.setCapability(Switch.ALLOW_SWITCH_WRITE);
  visSW.addChild( moveTG );
  visSW.setWhichChild( Switch.CHILD_ALL);   // visible initially
  isVisible = true;

  skelBG = new BranchGroup();
  skelBG.setCapability(BranchGroup.ALLOW_DETACH);
            // so skeleton can be deleted from the scene
  skelBG.addChild(visSW);

  return partsBG;    // where all the joints and limbs are attached
}  // end of buildSkelGraph()
```

Of special note are the calls to SceneGraphObject.setCapability() which assign important properties to the nodes.

The skelBG BranchGroup is made detachable (ALLOW_DETACH) which permits the skeleton subgraph to be removed from the scene, effectively deleting it. This feature is employed when LostUserObserver decides that the skeleton's user is lost.

The visSW Switch node is made writable (ALLOW_SWITCH_WRITE) allowing rendering to be switched off and on as often as necessary. This capability is utilized by ExitUserObserver and ReEntryUserObserver when the user exits and re-enters the scene.

The partsBG node's children are made readable and writable (ALLOW_CHILDREN_READ, ALLOW_CHILDREN_WRITE). Its children are the joint and limb subgraphs which need to be translated, rotated, scaled, and perhaps turned invisible at run time.

The moveTG TransformGroup is used to move the skeleton up the y-axis by an offset of Y_OFFSET (2.5 units) to make all of the skeleton visible, and have its feet joints rest on the floor. This value was arrived at by a process of trial-and-error. The Z_OFFSET is used to move the skeletons nearer the camera.

3.2. Building Joints

The joints are built by iterating through the SkeletonJoint enumeration, and creating a Joint3D object for each value. The Joint3D scene graphs are added to the skeleton at partsBG.

```
// global
private ArrayList<Joint3D> joints3D;


private void buildJoints(ArrayList<Joint3D> joints3D, int userID,
                  SkeletonCapability skelCap,
                  BranchGroup partsBG,
                  HashMap<SkeletonJoint, Joint3D> jointsMap)
{
```



```
  Joint3D j3d;
  for(SkeletonJoint joint : SkeletonJoint.values()) {
    j3d = buildJoint3D(joint, userID, skelCap);
    if (j3d != null) {
      partsBG.addChild( j3d.getTG() );
      joints3D.add(j3d);
      jointsMap.put(joint, j3d);   // build map for later
    }
  }
}  // end of buildJoints()
```

The jointsMap is used as a handy way of remembering which SkeletonJoint is represented by which Joint3D object. Each Joint3D object is also added to the global joints3D list.

buildJoint3D() creates a Joint3D object after checking that the SkeletonJoint value is valid (in the current version of OpenNI some joints named in SkeletonJoint aren't accessible).

```
// globals
// scaling from Kinect coords to 3D scene coords
private static final float XY_SCALE = 1/500.0f;
private static final float Z_SCALE = -1/1000.0f;


private Joint3D buildJoint3D(SkeletonJoint joint, int userID,
                        SkeletonCapability skelCap)
{
  if (!skelCap.isJointAvailable(joint) ||
      !skelCap.isJointActive(joint)) {
    /* To deal with the absence of WAIST,
       LEFT_COLLAR, LEFT_WRIST, LEFT_FINGER_TIP, LEFT_ANKLE,
       RIGHT_COLLAR, RIGHT_WRIST, RIGHT_FINGER_TIP, RIGHT_ANKLE
    */
    return null;
  }

  Joint3D j3d;
  if (joint == SkeletonJoint.HEAD)
    j3d = new Joint3D(joint, 0.22f, XY_SCALE, Z_SCALE,
                              userID, skelCap);   // bigger head
  else
    j3d = new Joint3D(joint, XY_SCALE, Z_SCALE, userID, skelCap);

  return j3d;
}  // end of buildJoint3D()
```

The Joint3D object is passed scaling factors (XY_SCALE, Z_SCALE) which are employed to convert Kinect (x, y, z) coordinates into values suitable for plotting inside the Java 3D scene. These scale factors were arrived at by trial-and-error. Note that the Z_SCALE scale is twice that used in the x- and y- directions. It's also negative so that the Java 3D z-values become negative, and so are placed along the z-axis going away from the camera).

3.3. Building Limbs

Each limb connects two joints (see Figure 7), and so limb creation requires a mapping from two SkeletonJoints (e.g. HEAD and NECK) to their corresponding Joint3D objects. References to these are stored in each Limb3D object so that when the joints change (e.g. move, turn), then the limb can be modified as well.

The mapping from SkeletonJoint to Joint3D uses two data structures. First the pair of SkeletonJoints for a limb are looked up inside the jointPairs[] array, and then each one is mapped to its Joint3D version using the jointsMap HashMap.

```
// globals
private static final float LIMB_RADIUS = 0.1f;

/*  the skeleton is a series of limbs;
    a limb is defined by a pair of joints which are
    listed in jointPairs[]
*/
private SkeletonJoint[] jointPairs =
  { SkeletonJoint.LEFT_SHOULDER, SkeletonJoint.LEFT_ELBOW,
    SkeletonJoint.LEFT_ELBOW, SkeletonJoint.LEFT_HAND,  // left arm

    SkeletonJoint.RIGHT_SHOULDER, SkeletonJoint.RIGHT_ELBOW,
    SkeletonJoint.RIGHT_ELBOW, SkeletonJoint.RIGHT_HAND,//right arm

    SkeletonJoint.HEAD, SkeletonJoint.NECK,
    SkeletonJoint.NECK, SkeletonJoint.LEFT_SHOULDER,
    SkeletonJoint.NECK, SkeletonJoint.RIGHT_SHOULDER,  // upper body

    SkeletonJoint.LEFT_SHOULDER, SkeletonJoint.TORSO,
    SkeletonJoint.RIGHT_SHOULDER, SkeletonJoint.TORSO,    // torso
    SkeletonJoint.LEFT_HIP, SkeletonJoint.TORSO,
    SkeletonJoint.RIGHT_HIP, SkeletonJoint.TORSO,

    SkeletonJoint.LEFT_HIP, SkeletonJoint.RIGHT_HIP, // across hips

    SkeletonJoint.LEFT_HIP, SkeletonJoint.LEFT_KNEE,
    SkeletonJoint.LEFT_KNEE, SkeletonJoint.LEFT_FOOT,  // left leg

    SkeletonJoint.RIGHT_HIP, SkeletonJoint.RIGHT_KNEE,
    SkeletonJoint.RIGHT_KNEE, SkeletonJoint.RIGHT_FOOT  //right leg
  };

private ArrayList<Limb3D> limbs;


private void buildLimbs(ArrayList<Limb3D> limbs,
                    BranchGroup partsBG,
                    HashMap<SkeletonJoint, Joint3D> jointsMap)
{
  Limb3D limb;
  Joint3D startJ3d, endJ3d;
  for (int i=0; i < jointPairs.length; i=i+2) {
    startJ3d = getJoint3D(jointPairs[i], jointsMap);
    endJ3d = getJoint3D(jointPairs[i+1], jointsMap);
    if ((startJ3d != null) && (endJ3d != null)) {
      limb = new Limb3D(startJ3d, endJ3d, LIMB_RADIUS);
      partsBG.addChild(limb.getTG());
      limbs.add(limb);
    }
```



```
  }
}  // end of buildLimbs()


private Joint3D getJoint3D(SkeletonJoint joint,
            HashMap<SkeletonJoint, Joint3D> jointsMap)
{
  Joint3D j3d = jointsMap.get(joint);
  if (j3d == null) {
    System.out.println("Undefined " + joint);
    return null;
  }
  return j3d;
}  // end of getJoint3D()
```

The resulting Limb3D object is stored in a global list for later updating, and the limbs Java 3D subgraph is attached to the partsBG BranchGroup.

3.4. Updating the Skeleton

A skeleton is updated via calls to its update() method which first updates all the Joint3D objects (with positions obtained from SkeletonCapability), and then updates all the limbs (which get their coordinates from their Joint3D endpoints).

```
// globals
private ArrayList<Joint3D> joints3D;
private ArrayList<Limb3D> limbs;


public void update()
{
  if (!isVisible)
    return;

  // update joints
  for(Joint3D j3d : joints3D)
    j3d.update();

  // update limbs
  for(Limb3D limb : limbs)
    limb.update();
}  // end of update()
```

There's no need to update anything if the skeleton is currently invisible.

3.5. Modifying a Skeleton's Visibility

The exit of a user from the scene causes the skeleton to become invisible, via a call to Skeleton3D.setVisibility(). If the user re-enters the scene soon enough then setVisibility() will be called again to make the skeleton visible.

```
// globals
private Switch visSW;
private boolean isVisible;
```



```
public void setVisibility(boolean toVisible)
{
  if (toVisible) {
    visSW.setWhichChild(Switch.CHILD_ALL);   // make visible
    isVisible = true;
  }
  else {   // make invisible
    visSW.setWhichChild( Switch.CHILD_NONE );   // invisible
    isVisible = false;
  }
}  // end of setVisibility()
```

Toggling rendering is a matter of setting the visSW Switch node; the visibility status is recorded in the isVisible boolean.

3.6. Deleting the Skeleton

If the user is absent from the scene for long enough then the LostUserObserver will call Skeleton.delete(). This detaches the skeleton (skelBG) from the main scene graph.

```
// global
private BranchGroup skelBG;

public void delete()
{  skelBG.detach();  }
```

4. Creating a 3D Joint

Each Joint3D object manages a joint in the 3D scene. which involves three main tasks:

initially creating the joint's subgraph;

updating the joint by moving it to the position supplied by the SkeletonCapability object;

making the joint invisible or visible. This may seem a bit superfluous since I've already got skeleton-level visibility, but it allows finer-level control over what parts of the skeleton are visible.

4.1. Creating the Joint Subgraph

The joint subgraph created by Joint3D is shown in Figure 9.

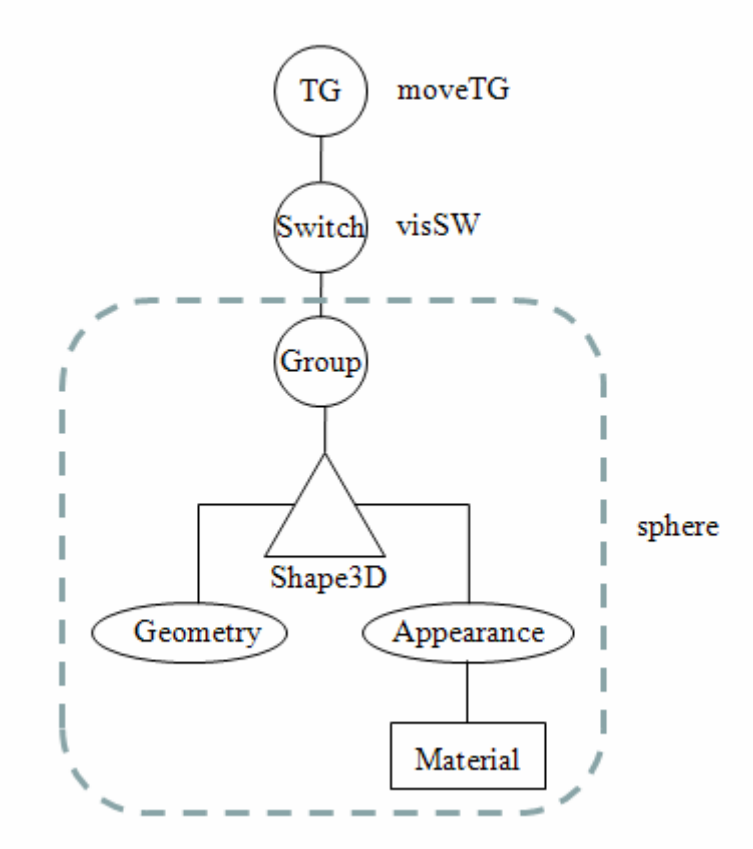

Figure 9. A Joint Branch.

Figure 9 corresponds to the "Joint Branch" box in Figure 6. The dashed rounded rectangle is a Sphere object. Sphere is a utility class from the com.sun.j3d.utils.geometry package, consisting of a Group node with a Shape3D child. A typical invocation is:

```
Sphere s = new Sphere(radius, flags, appearance)
```

The Appearance node can hold many kinds of appearance information, including coloring, transparency, and texture attributes. The Material node allows the lights in the scene to affect the color of the shape.

The moveTG TransformGroup permits the joint to be moved, and is the connection point to the 3D scene. The visSW Switch enables the subgraph to be made invisible.

The Joint3D constructor builds the subgraph:

```
// globals
// sphere colors
private static final Color3f BLACK = new Color3f(0.0f, 0.0f, 0.0f);
private static final Color3f WHITE = new Color3f(0.9f, 0.9f, 0.9f);
private static final Color3f BLUE = new Color3f(0.3f, 0.3f, 0.8f);


private SkeletonCapability skelCap;
private SkeletonJoint joint;

private int userID;   // ID of skeleton containing this joint
```



```
// the joint's scene graph: moveTG-->visSW-->sphere
private TransformGroup moveTG;
private Transform3D t3d;         // for accessing a TG's transform
private Switch visSW;            // for joint visibility
private boolean isVisible;

private SmoothPosition smoothPosns;

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


public Joint3D(SkeletonJoint j, float radius,
                      float xyScale, float zScale,
                      int userID, SkeletonCapability skelCap)
{
  joint = j;
  this.xyScale = xyScale;
  this.zScale = zScale;
  this.userID = userID;
  this.skelCap = skelCap;

  smoothPosns = new SmoothPosition();

  Appearance app = new Appearance();

  // assign blue material with lighting
  Material blueMat= new Material(BLUE, BLACK, BLUE, WHITE, 25.0f);
        // sets ambient, emissive, diffuse, specular, shininess
  blueMat.setLightingEnable(true);
  app.setMaterial(blueMat);

  // make the sphere with normals for lighting
  Sphere sphere = new Sphere(radius, Sphere.GENERATE_NORMALS, app);

  // create switch for visibility
  visSW = new Switch();
  visSW.setCapability(Switch.ALLOW_SWITCH_WRITE);
  visSW.addChild( sphere );
  visSW.setWhichChild( Switch.CHILD_ALL);   // visible initially
  isVisible = true;

  // create a transform group for moving the sphere
  t3d = new Transform3D();
  moveTG = new TransformGroup(t3d);
  moveTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  moveTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  moveTG.addChild(visSW);
}  // end of Joint3D()
```

For light to affect the sphere's color, three conditions must be met:

1. The shape's geometry must include normals;
2. The shape's Appearance node must have a Material component;
3. The Material component must enable lighting effects with setLightingEnable().

The Sphere class can automatically creates normals by including the Sphere.GENERATE_NORMALS flag in its constructor, so (1) is easily satisfied. The Material component controls what color the shape exhibits when lit by different kinds of lights, and is created like so:

```
Material mat = new Material(ambientColor, emissiveColor,
                   diffuseColor, specularColor, shininess);
```

The ambient argument specifies the shape's color when lit by ambient light: this gives the object a uniform color.

The emissive argument contributes the color that the shape produces itself (as in a light bulb); frequently, the value is set to black (off).

The diffuse value is the color of the object when lit, with its intensity depending on the angle the light beams make with the shape's surface. Often the diffuse and ambient colors are the same.

The intensity of the specular color parameter is related to how much the shape reflects from it's shiny areas. This is combined with the shininess argument which controls the size of the reflective highlights. Often the specular color is white.

The Joint3D constructor creates the following Material node:

```
Material blueMat= new Material(BLUE, BLACK, BLUE, WHITE, 25.0f);
```

The sphere's ambient and diffuse colors are blue, not emissive, and reflects a small amount of white light. These lighting effects can be seen in the close-up of the head joint in Figure 10.

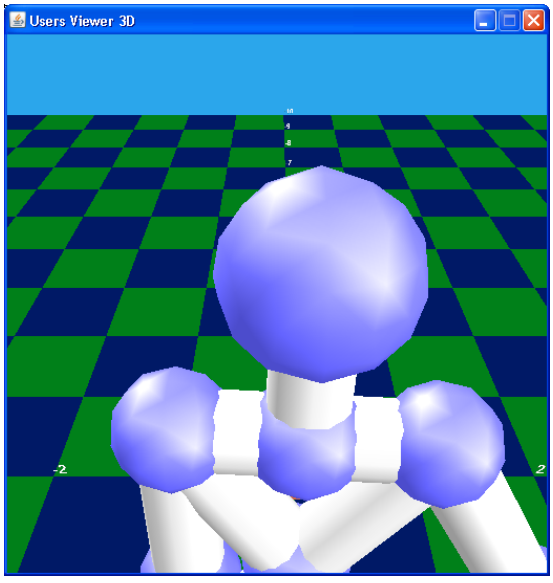

Figure 10. Lighting on the Head Joint.

There are two patches of shininess on the head (and the other joints) because the scene is lit by two directional lights (see Figure 4).

4.2. Updating the Joint's Position

Joint3D.update() is called when the SkeletonCapability object has new joint position information. The getKinectPos() method retrieves the data.

```
// globals
private SkeletonCapability skelCap;
private SkeletonJoint joint;
private int userID;   // ID of skeleton containing this joint


public void update()
// called by Skeleton3D
{
  org.OpenNI.Point3D pos = getKinectPos();
  setPos(pos);
}


private org.OpenNI.Point3D getKinectPos()
// query Kinect for position of joint (may be null)
{
  if (!skelCap.isJointAvailable(joint) ||
      !skelCap.isJointActive(joint)) {
    System.out.println(joint + " not available");
    return null;
  }

  SkeletonJointPosition pos = null;
  try {
    pos = skelCap.getSkeletonJointPosition(userID, joint);
  }
  catch(StatusException e) {}
  if (pos == null) {
    System.out.println("No update for " + joint);
    return null;
  }

  if (pos.getConfidence() == 0) {
    // System.out.println("No confidence in " + joint);
    return null;
  }

  return pos.getPosition();
}  // end of getKinectPos()
```

The setPos() method shown next converts the Kinect (x, y, z) coordinate into one suitable for the 3D scene, and updates the moveTG TransformGroup; the joint sphere will then be moved to that location by Java 3D

One problem with this approach is the occasional 'shuddering' of joints. This is caused by the Kinect being uncertain about the position of joints which are partially (or completely) obscured. Since the joints are being updated so rapidly (perhaps 10 times per second), any incorrect positions will be corrected quickly, but the user may see the joints rapidly moving by small amounts (i.e. shuddering).

My solution is to store recent Kinect positions in a SmoothPosition object, and update the sphere using an average of those positions. This smoothes away small, brief movements of the joint.

```
// globals
private TransformGroup moveTG;
private Transform3D t3d;      // for accessing a TG's transform
private SmoothPosition smoothPosns;
private float xyScale, zScale;


private void setPos(org.OpenNI.Point3D pos)
{
  if (pos == null)
    smoothPosns.addPosition(null);
  else {   // store a scaled position
    double x = (double) pos.getX()*xyScale;
    double y = (double) pos.getY()*xyScale;
    double z = (double) pos.getZ()*zScale;
    smoothPosns.addPosition( new Vector3d(x, y, z));
  }

  // use the average position to translate the sphere
  Vector3d sPos = smoothPosns.getPosition();
  if (sPos != null) {
    setVisibility(true);
    t3d.set(sPos);
    moveTG.setTransform(t3d);
  }
  else     // joint has no position
    setVisibility(false);
}  // end of setPos()
```

A new position is added to the SmoothPosition object, after being suitably scaled. The scaling values are passed to Joint3D in its constructor, and stored as the globals xyScale and zScale. The sphere is updated with an average position obtained from SmoothPosition.

If no position value is available (i.e. sPos == null) then the joint is turned invisible.

4.3. Smoothing a Position

The SmoothPosition class stores a list of Vector3d positions, up to a maximum of MAX_POSNS.

```
// globals
private final static int MAX_POSNS = 10;
private ArrayList<Vector3d> posns;


public SmoothPosition()
{  posns = new ArrayList<Vector3d>();  }
```

I feel a little guilty about using an ArrayList for storing the positions, because the storage is a bounded buffer which is better encoded as a fixed-size array. But a list is easier to manipulate.

A new element is added to the end of the list, and the first element (the 'oldest' value) is deleted if the MAX_POSN limit has been reached.

```
public void addPosition(Vector3d p)
{
  if (p == null) {
    if (!posns.isEmpty())
      posns.remove(0); //remove oldest element when null 'added'
  }
  else {
    if (posns.size() == MAX_POSNS)
      posns.remove(0);     // remove oldest
    posns.add(p);
  }
}  // end of addPosition()
```

A complication is that the Vector3d object being added to SmoothPosition may be null, indicating that there's some problem with the Kinect position. addPosition() deletes the oldest element in this case, which means that if the error continues over several updates, that the list will gradually empty.

getPosition() returns the average of its list positions, or null if the list is empty.

```
public Vector3d getPosition()
{
  if (posns.isEmpty())
    return null;  // since no positions to average
  else {
    double xSum = 0;
    double ySum = 0;
    double zSum = 0;
    int count = 0;
    for(Vector3d v : posns) {
      xSum += v.getX();
      ySum += v.getY();
      zSum += v.getZ();
      count++;
    }
    return new Vector3d(xSum/count, ySum/count, zSum/count);
  }
}  // end of getPosition()
```

4.4. Joint Visibility

If getPosition() returns null then there's a problem with the joint position, perhaps because the joint is no longer within range of the Kinect sensor. As a consequence, the joint is made invisible in setPos(), by calling setVisibility(false):

```
// globals
private Switch visSW;            // for joint visibility
private boolean isVisible;
```



```
private void setVisibility(boolean toVisible)
/* toggle the visibility of the joint
  (called by a limb connected to this joint) */
{
  if (toVisible) {
    visSW.setWhichChild(Switch.CHILD_ALL);   // make visible
    isVisible = true;
  }
  else {   // make invisible
    visSW.setWhichChild( Switch.CHILD_NONE );   // invisible
    isVisible = false;
  }
}  // end of setVisibility()
```

5. Creating a 3D Limb

Each Limb3D object manages a limb in the 3D scene, which involves two tasks:

initially creating the limb subgraph;

updating the limb by setting its visibility, position, orientation, and length.

5.1. Creating the Limb Subgraph

The limb subgraph created by Limb3D is shown in Figure 11.

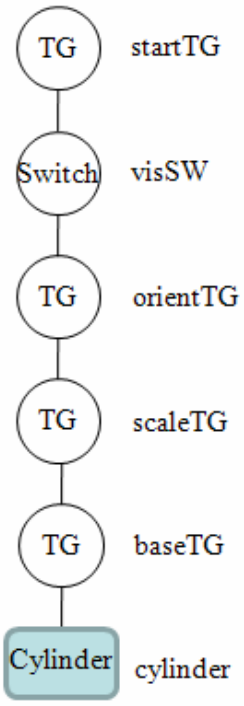

Figure 11. A Limb Branch.

Figure 11 corresponds to the "Limb Branch" box in Figure 6. startTG holds the limb's starting joint location; the visSW Switch manages visibility; orientTG sets the limb's orientation so it points from its start joint towards the end joint; scaleTG is used to scale the limb's length.

Cylinder is a Java 3D shape utility, implemented in a similar way to Sphere. When a cylinder is created, its center is located at the origin, which isn't suitable for our purposes because I want to rotate and position the cylinder relative to its base. So the baseTG node is employed to lift the cylinder up the y-axis by half its length, so the center of its base is at the origin

The long chain of nodes in Figure 11 may seem excessive, especially when four of them are TransformGroups. It's possible to implement the subgraph with less nodes, but it would make their updating more complicated. A chain of nodes splits the update into parts, and imposes an update ordering. The updates are performed starting from the leaf node (the cylinder in Figure 11), working up the branch.

First the cylinder's base is moved to the origin (by baseTG), then its y-axis length is stretched (by scaleTG), then it's rotated (by orientTG) so it's facing in the direction defined by its joints, then its visibility is set (by visSW), and finally it's translated (by startTG) so its base coincides with the start joint.

The Limb3D constructor builds the subgraph in Figure 11:

```
// globals
private static final double LIMB_LEN = 1;

// the joints connected to this limb
private Joint3D startJ3d, endJ3d;
private SkeletonJoint startJoint, endJoint;

// for positioning the limb
private TransformGroup startTG;
private Transform3D startT3d;

// for limb visibility
private Switch visSW;
private boolean isVisible;

private TransformGroup orientTG;
private TransformGroup scaleTG;


public Limb3D(Joint3D startJ3d, Joint3D endJ3d, float radius)
{
  this.startJ3d = startJ3d;
  this.endJ3d = endJ3d;
  startJoint = startJ3d.getJoint();
  endJoint = endJ3d.getJoint();

  // limb's position at the start joint
  startT3d = new Transform3D();
  startTG = new TransformGroup(startT3d);
  startTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
  startTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

  // create switch for visibility
  visSW = new Switch();
```



```
  visSW.setCapability(Switch.ALLOW_SWITCH_WRITE);
  visSW.setWhichChild( Switch.CHILD_ALL);   // visible initially
  isVisible = true;

  // limb's orientation
  orientTG = new TransformGroup();
  orientTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
  orientTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

  // for changing the length of a limb
  scaleTG = new TransformGroup();
  scaleTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
  scaleTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

  TransformGroup baseTG = makeLimb(radius, LIMB_LEN);

  // limb subgraph's sequence of nodes:
  // startTG -->visSW-->orientTG -->scaleTG --> baseTG --> cylinder
  startTG.addChild(visSW);
  visSW.addChild(orientTG);
  orientTG.addChild(scaleTG);
  scaleTG.addChild(baseTG);
} // end of Limb3D()
```

The cylinder is created by makeLimb():

```
// globals
// colors for limb material
private static final Color3f BLACK = new Color3f(0.0f, 0.0f, 0.0f);
private static final Color3f WHITE = new Color3f(0.9f, 0.9f, 0.9f);
private static final Color3f GRAY = new Color3f(0.6f, 0.6f, 0.6f);

private Transform3D currTrans = new Transform3D();


private TransformGroup makeLimb(float radius, double len)
// a gray cylinder whose base is at the origin
{
  // fix limb's start position
  TransformGroup baseTG = new TransformGroup();
  currTrans.setTranslation( new Vector3d(0, len/2, 0) );
                              // move up length/2
  baseTG.setTransform(currTrans);

  Appearance app = new Appearance();
  Material limbMaterial =
     new Material(GRAY, BLACK, GRAY, WHITE, 150.0f);
       // sets ambient, emissive, diffuse, specular, shininess
  limbMaterial.setLightingEnable(true);
  app.setMaterial( limbMaterial );

  Cylinder cyl = new Cylinder( radius, (float)len, app);
  baseTG.addChild( cyl );
  return baseTG;
}  // end of makeLimb()
```

The Cylinder constructor needs a radius, length, and Appearance node. As with the sphere, I utilize a Material node so the cylinder will be affected by the scene's lights.

5.2. Updating the Limb

When the limb is updated by a call to Limb3D.update(), four attributes may need to be modified: visibility, position, orientation, and limb length.

The limb should only be updated if the joints at its two ends have valid positions, which update() confirms by calling isLimbUpdatable(). The method performs two checks:

```
// global
private static final double MIN_DIST = 0.00001;
          // small distance that probably indicates an error

private Joint3D startJ3d, endJ3d;


private boolean isLimbUpdatable(Vector3d startPos, Vector3d endPos)
// can the limb be updated given these joint positions?
{
  if (!startJ3d.isVisible() || !endJ3d.isVisible())
    return false;

  /* small diff between the (x,z) coordinates of the joints
     indicates a likely error */
  if ((Math.abs(endPos.x - startPos.x) < MIN_DIST) &&
      (Math.abs(endPos.z - startPos.z) < MIN_DIST))
    return false;

  return true;
}  // end of isLimbUpdatable()
```

If either of the joints are invisible, then one or both do not have a valid position. Also, if the joints are very close together in the XZ plane then there's probably a problem with the limb's direction vector, as explained at the end of this section.

If isLimbUpdatable() returns false, the limb is made invisible, as shown in the fragment of update() below:

```
// globals
// the joints connected to this limb
private Joint3D startJ3d, endJ3d;


public void update()
{
  // get start and end joint positions
  Vector3d startPos = startJ3d.getPos();
  Vector3d endPos = endJ3d.getPos();

  if (!isLimbUpdatable(startPos, endPos)) {  // hide the limb
    setVisibility(false);
    return;
  }

  // both joints are ok, so make the limb visible
  if (!isVisible)
    setVisibility(true);
```



```
  // more code not shown. . .

}  // end of update()
```

The setVisibility() method affects the Switch node, visSW:

```
// globals
private Switch visSW;
private boolean isVisible;


private void setVisibility(boolean toVisible)
// change limb visibility
{
  if (toVisible) {
    visSW.setWhichChild(Switch.CHILD_ALL);   // make visible
    isVisible = true;
  }
  else {   // make invisible
    visSW.setWhichChild( Switch.CHILD_NONE );   // invisible
    isVisible = false;
  }
}  // end of setVisibility()
```

This approach means that a problem with a limb's joints will cause that limb to become invisible. Figure 12 shows the effect on a skeleton.

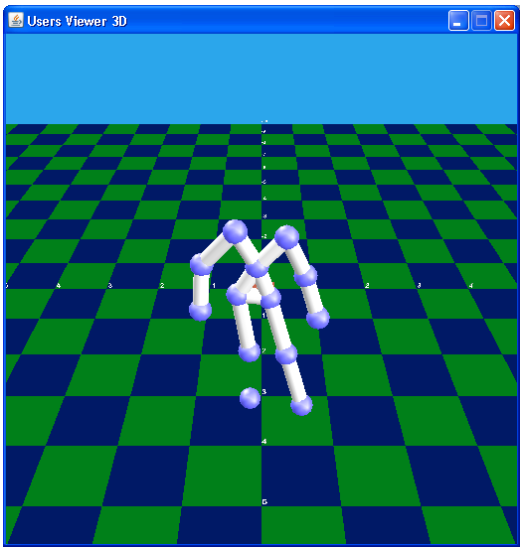

Figure 12. A Skeleton with Some Invisible Joints and Limbs.

The neck and neck joints are invisible, along with their connecting limbs, because the user's head and neck are too close to the Kinect for their positions to be calculated. As soon as the user moves back into range of the Kinect, the joints and limbs will reappear.

In Figure 12, the limb connecting the left knee to the foot is invisible because the (x, z) difference between the left knee and foot joint is almost zero. I'll explain why this is a problem at the end of this section.

Another way of dealing with a joint with an invalid position is to have it use its last correct value instead. This doesn't look so nice in practice because, as the skeleton moves, the limbs connected to the static joint will become elongated and twisted.

The best answer is probably to use interpolation to calculate a likely position for a joint based on its previous positions and rotations.

5.3. Adjusting Limb Position and Length

A limb's new position is the location of its base's start joint, which is set by adjusting the startTG TransformGroup. The limb's length also needs to be changed since the Kinect does not reliably keep joints a fixed distance apart (unlike the joints in a real body). The current distance between the two joints must be calculated, and used to scale the limb's y-axis via the scaleTG TransformGroup.

The relevant code in update() is shown below:

```
// globals
private Joint3D startJ3d, endJ3d;

// for positioning the limb
private TransformGroup startTG;
private Transform3D startT3d;


// code fragments from Limb3D.update(). . .

// get start and end joint positions
Vector3d startPos = startJ3d.getPos();
Vector3d endPos = endJ3d.getPos();

// update limb position
startT3d.set(startPos);
startTG.setTransform(startT3d);

Vector3d lengthVec =
    new Vector3d( (double)(endPos.x - startPos.x),
                  (double)(endPos.y - startPos.y),
                  (double)(endPos.z - startPos.z));
double len = lengthVec.length();

rotateLimb(lengthVec);
setLength(len);    // change length of limb
```

The setLength() method changes the cylinder's length by modifying the scaling factor used in scaleTG():

```
// globals for scaling
private TransformGroup scaleTG;
private Vector3d scaleLimb = new Vector3d(1,1,1); // only y changes
private Transform3D currTrans = new Transform3D();


private void setLength(double len)
// change the cylinder's length to len (by changing the scaling)
{
  double lenChange = len / (LIMB_LEN * scaleLimb.y);
```



```
  scaleLimb.y *= lenChange;   // only y scale changes

  scaleTG.getTransform(currTrans);
  currTrans.setScale(scaleLimb);
  scaleTG.setTransform(currTrans);
}  // end of setLength()
```

5.4. Changing the Limb's Orientation

rotateLimb() rotates the limb about the start joint to point at the end joint. The problem is that a simple rotation about the x-, y-, or z- axis is insufficient, since the joints can be anywhere in the 3D space. Instead, an AxisAngle4d rotation is utilized, which can define a rotation about any vector. rotateLimb()'s algorithm is illustrated in Figure 13.

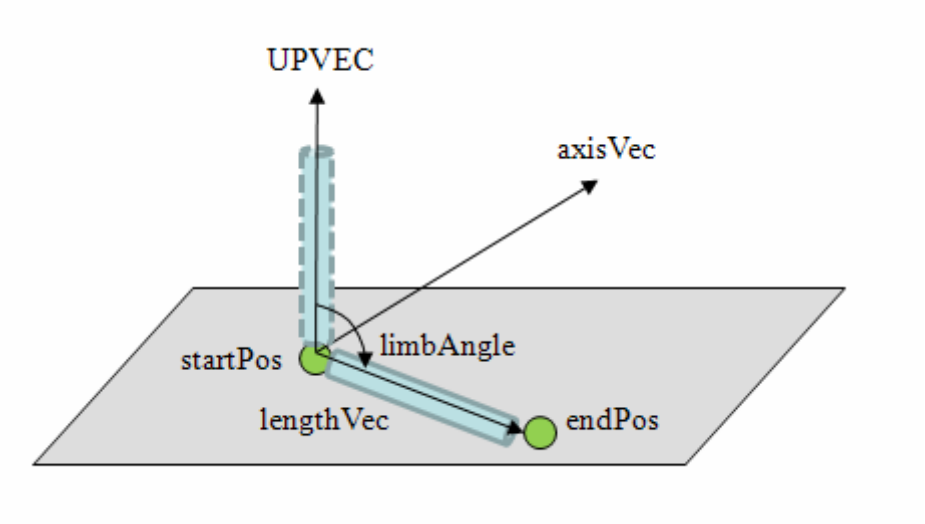

Figure 13. Rotating a Limb.

The start and end joints are located at startPos and endPos respectively, and the limb begins by pointing in the UPVEC direction with its base at startPos. It must be rotated to point in the lengthVec direction, a rotation of limbAngle radians.

The rotation should be carried out around the axisVec vector, which is normal to the plane defined by the two vectors UPVEC and lengthVec, and is known as their cross product. The Vector3d class contains a cross() method which calculates the vector, given normalized values for UPVEC and lengthVec.

The rotation angle, limbAngle, between UPVEC and lengthVec, can also be easily obtained with Vector3d's angle() method. An AxisAngle4d object requires a vector and rotation, which is supplied in the calcRotation() method:

```
// globals
private static final Vector3d UPVEC = new Vector3d(0.0, 1.0, 0.0);
            // initial orientation of limb: straight up

private Vector3d axisVec = new Vector3d();
private AxisAngle4d rotAxisAngle = new AxisAngle4d();
private double limbAngle = 0;
```

```
private AxisAngle4d calcRotation(Vector3d lengthVec)
 {
  lengthVec.normalize();
  axisVec.cross(UPVEC, lengthVec);   // calculate axisVec

  limbAngle = UPVEC.angle(lengthVec);    // get rotation

  rotAxisAngle.set(axisVec, limbAngle);  // build axis angle
  return rotAxisAngle;
} // end of calcRotation()
```

A complication is that rotateLimb() assumes that the limb starts in the UPVEC direction, which is only true when a limb is initially created. Before subsequent rotations, the limb must first be rotated back to the vertical. A negative AxisAngle4d rotation is calculated in rotateLimb() using the limbAngle and the negative of the axisVec vector before calling calcRotation():

```
// globals
private Vector3d negAxisVec = new Vector3d();
private AxisAngle4d negRotAxisAngle = new AxisAngle4d();
private boolean firstRotation = true;
           // to flag the need to undo the previous rotation


private void rotateLimb(Vector3d lengthVec)
{
  if (!firstRotation) {   // calculate neg of previous rotation
    negAxisVec.negate(axisVec);
    negRotAxisAngle.set(negAxisVec, limbAngle);
  }

  // update limb orientation
  AxisAngle4d rotAxisAngle = calcRotation(lengthVec);
  doRotation(rotAxisAngle, negRotAxisAngle);
  firstRotation = false;
}  // end of rotateLimb()
```

doRotation() applies the two AxisAngle4d rotations to the orientTG TransformGroup:

```
// globals
private Transform3D orientT3d = new Transform3D();
private Transform3D rotT3d = new Transform3D();
private Transform3D negRotT3d = new Transform3D();
private TransformGroup orientTG;

private void doRotation(AxisAngle4d rotAxisAngle,
                        AxisAngle4d negRotAxisAngle)
{
  orientTG.getTransform(orientT3d);   // get current transform

  if (!firstRotation) {   // undo previous rotation first
    negRotT3d.setRotation(negRotAxisAngle);
    orientT3d.mul(negRotT3d);
  }

  // apply new rotation
```



```
  rotT3d.setRotation(rotAxisAngle);
  orientT3d.mul(rotT3d);
  orientTG.setTransform(orientT3d);
} // end of doRotation()
```

5.5. An Interpolation and Cross Product Problem

The isLimbUpdatable() method includes a rather mysterious test to see if two joints are very close together in the XZ plane. In that case, the method returns false and the limb between the joints isn't drawn. If it was drawn then there's a good chance that it will point straight up the y-axis.

The test is mysterious partly because I'm not quite sure why it's needed :). It appears that the UserGenerator node interpolates the positions of some joints which the Kinect cannot see. For example, in Figure 12, the left foot joint (the one on the left of the picture with no limb) is not actually visible to the Kinect sensor. The joint's coordinates appears to be generated by using the (x, z) coordinate of the left knee joint plus a y-axis offset. Another example is when a joint is obscured by another, such as when the user turns sideways to the Kinect. The obscured joint appears to be assigned a similar (x, z) value to the joint that's in front of it.

LengthVec is the vector between two adjacent joints (see Figure 13). If the XZ component of lengthVec is very close to zero, then the AxisAngle4d object can rotate correctly by limbAngle radians. However, if the two joints have *identical* (x, z) values then the normalized lengthVec will be parallel to UPVEC, and the cross product returns a null vector (i.e. (0, 0, 0)). When that happens, the axis angle ignores the limbAngle value, and the limb will end up pointing in the same direction as UPVEC, up the y-axis.

isLimbUpdatable() checks for XZ values very close to 0, and makes a limb invisible before it flips into an upright position.