Chapter 11.5. An Articulated, Moveable Figure

This chapter describes the implementation of an articulated figure, composed of rotatable limbs, which can be moved around a checkboard floor in a similar manner to the 3D sprites in chapters 10 and 11.

This work is based on the first part of Thana Konglikhit's student project (`s4310170@maliwan.psu.ac.th`). He is expected to finish in February/March of 2004.

Figure 1 shows the figure in its initial stance, and figure 2 after the following commands have been processed:

```
  urLeg f 40, lrLeg f -40, ulArm f 20, llArm f 20, chest t 10, head t -10
```

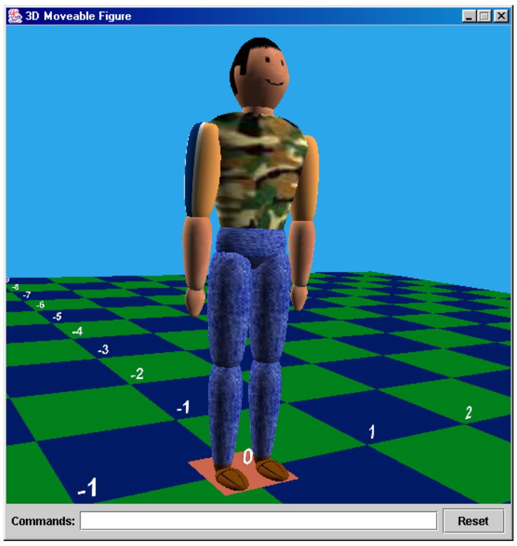
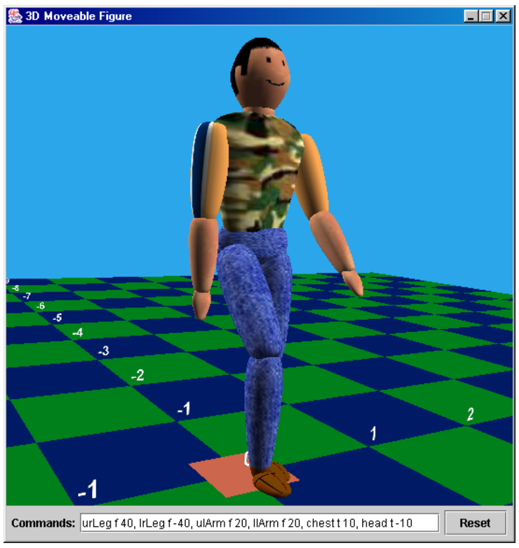


Figure 1. Initial Position.



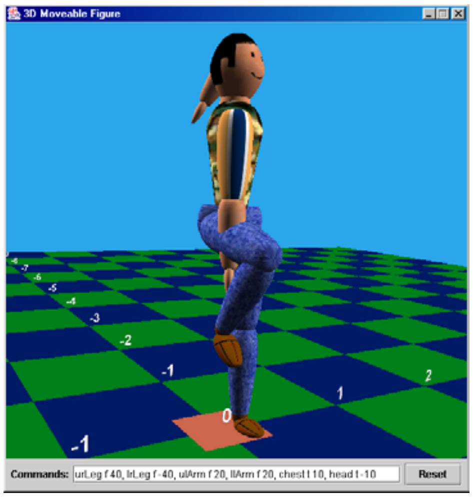
urLeg f 40, lrLeg f -40, ulArm f 20, llArm f 20, chest t 10, head t -10

Figure 2. After Limb Movement.

The first four commands specify forward (f) rotations of the limbs representing the upper part of the right leg (urLeg), the lower right leg (lrLeg), the upper left arm (ulArm), and the lower part of the left arm (llArm). The chest and head are turned (t) left and right respectively, so that the head stays facing forward.

All the operations are carried out as a group, causing a single re-orientation of the figure.

Pressing <enter> repeats the commands, although when a limb reaches its predefined maximum or minimum rotation, operations which would rotate it beyond these limits are ignored. Figure 3 shows the result of executing the commands several times.



Commands: urLeg f 40, lrLeg f -40, ulArm f 20, llArm f 20, chest t 10, head t -10

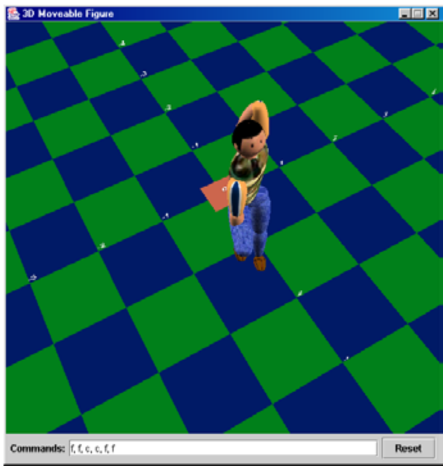
Figure 3. Repeated Limb Movements.

Note that the right arm passes through the right leg; The Mover3D application does not employ collision avoidance to prevent limbs intersecting.

The user can move the entire figure about the floor by typing commands into the text field or by pressing arrow keys on the keyboard. Figure 4 displays the outcome of the commands:

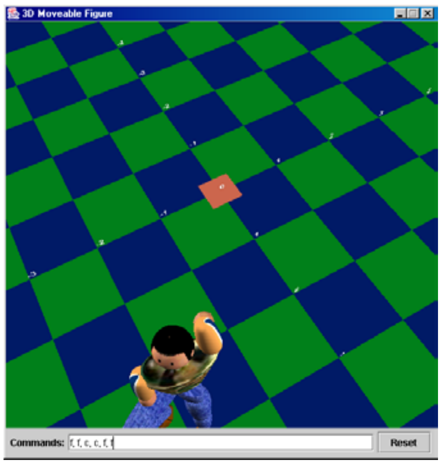
```
f, f, c, c, f, f
```

They cause the figure to move from its starting position at (0,0) on the floor: forward 0.6 units, 22.5 degrees to its right, and forward another 0.6 units. Figure 5 is a view of the scene after repeating these commands three times. Sixteen repetitions will cause the figure to return to its starting position at (0,0).



Commands: f, f, c, c, f, f

Figure 4. Figure Movement.



Commands: f, f, c, c, f, f

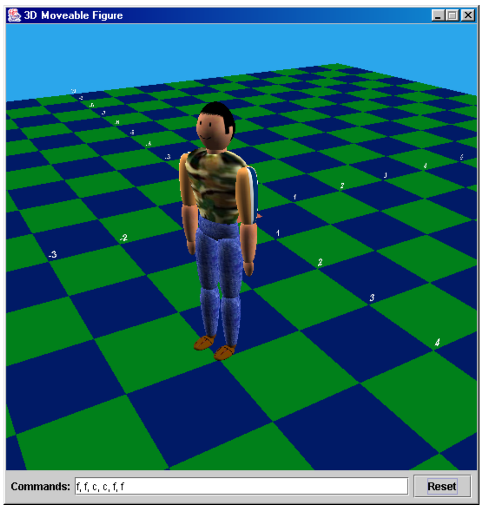
Figure 5. Repeated Figure Movement

These commands are reminiscent of the turtle geometry constructs found in languages like Logo.

As with the limb operations, all movements entered into the textfield update the figure at once. The operations are carried out in the order specified by reading the input sequence left-to-right.

A close look at the figure in Figures 4 and 5 shows that its limbs are unaffected by the movement: the entire figure is moved and rotated as a single 'unit'.

Figure 6 illustrates the result of pressing the "reset" button in the GUI: the figure's limbs are rotated back to their starting position, but the figure remains at its current position and orientation on the floor.



Commands: f, f, c, c, f, f

Figure 6. Reset Limbs.

Other features:

- the articulated figure is created by connecting instances of our Limb class and its subclasses. These classes are sufficiently general to build most kinds of articulated shape;
- each limb can be given an initial orientation relative to its 'parent' limb, and can be rotated around its x-, y-, and z-axes at run time. The rotation ranges can be pre-set;
- the shape of a limb is described using a LatheShape3D object (see chapter 9.5) which allows a typical limb shape to be specified with just a few coordinates;
- a limb's cross-sectional shape can be modified by using subclasses of LatheShape3D when defining the limb;
- the appearance of a limb is derived from a texture, and reflects light;
- a limb may be invisible, which enables it to be used as a connector between other limbs without being rendered;
- the command language for limbs allows each limb to be individually rotated;
- the command language for the figure permits the figure to be moved around the board, rotated around the y-axis, and lifted into the air. The figure cannot be

lowered below the floor or rotated around its x- or z- axes. This means, for example, that it is not possible to make the figure lie on its back on the floor.

1. Forward and Inverse Kinematics

Java 3D's scene graph imposes a parent-child relationship on (most of) its nodes. This hierarchy is particularly important for sequences of TransformGroups.

Figure 7 shows a simple hierarchy made up of a parent and a child TransformGroup. The parent holds a translation of (1,1,2), the child a translation of (2, 3, 1). However, from the world's viewpoint, the child's translation will be (3, 4, 3), a combination of the parent and child values.

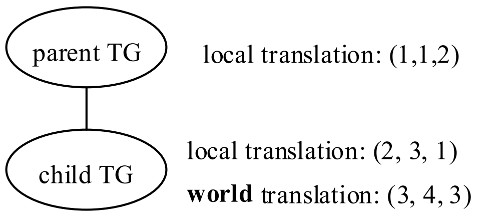

parent TG
local translation: (1,1,2)
child TG
local translation: (2, 3, 1)
world translation: (3, 4, 3)

Figure 7. A Hierarchy of TransformGroups.

In general, the world (or scene) view of a TransformGroup is a combination of its translation, rotations, and scaling with those of its ancestors (parent, grandparent, and so on).

This hierarchy is important when developing an articulated figure, since each limb contains several TransformGroups, and the connection of limbs to make the complete figure creates a large hierarchy of TransformGroups. The consequence is that when a limb is moved (by affecting one of its TransformGroups), the limbs linked to it as children will also move.

This top-down behaviour is at the heart of *forward kinematics*, one of the standard approaches to animating articulated figures. For example, the rotation of a figure's chest can cause its arms and head to turn, while the bottom and legs remain stationary. From a programming point of view, this means much less explicit manipulation of TransformGroups, but requires that the arms and head are connected as children to the chest's TransformGroup.

Forward kinematics is especially useful for movements which originate at the top-level of a figure and 'ripple down' to the lower-level components. An everyday example is moving a figure: the translation is applied to the top-most TransformGroup, and all the other nodes will move as well.

Forward kinematics is a lot less satisfactory for operations which start at lower-level limbs and should then 'ripple up'. For instance, the natural way of having a figure touch an object in the scene is to move the hand to the object's location. As the hand is moved, the arm and torso should follow. Unfortunately, this would require that a child TransformGroup be able to influence its ancestors, which is not possible in the parent-child hierarchy used by Java 3D.

The 'ripple up' animation technique is called *inverse kinematics*, and is a staple of professional animation packages such as Poser, Maya, and 3D Studio Max. Important low-level nodes are designated as *end-effectors*, and these influence higher-level nodes as they are manipulated. Typically end-effectors for an articulated human are its hands, feet, and head.

Inverse kinematics has problems specifying top-down effects, and so is often combined with *constraints* which link end-effectors to other nodes. For instance, when the body moves, the end-effectors can be constrained to always stay within a certain distance of the torso.

Our Mover3D application utilizes Java 3D's scene graph, with its parent-child relationship, and so specifies movement in terms of forward kinematics.

A good non-technical introduction to forward and inverse kinematics is:

> *Character Animation: Skeletons and Inverse Kinematics*
> Steve Pizel, Intel Developer Service
> `http://www.intel.com/cd/ids/developer/asmo-na/eng/segments/`
> `games/resources/modeling/20433.htm`

2. UML Diagrams for Mover3D

Figure 8 shows the UML diagrams for all the classes in the Mover3D application. The class names and public/protected methods and data are shown.

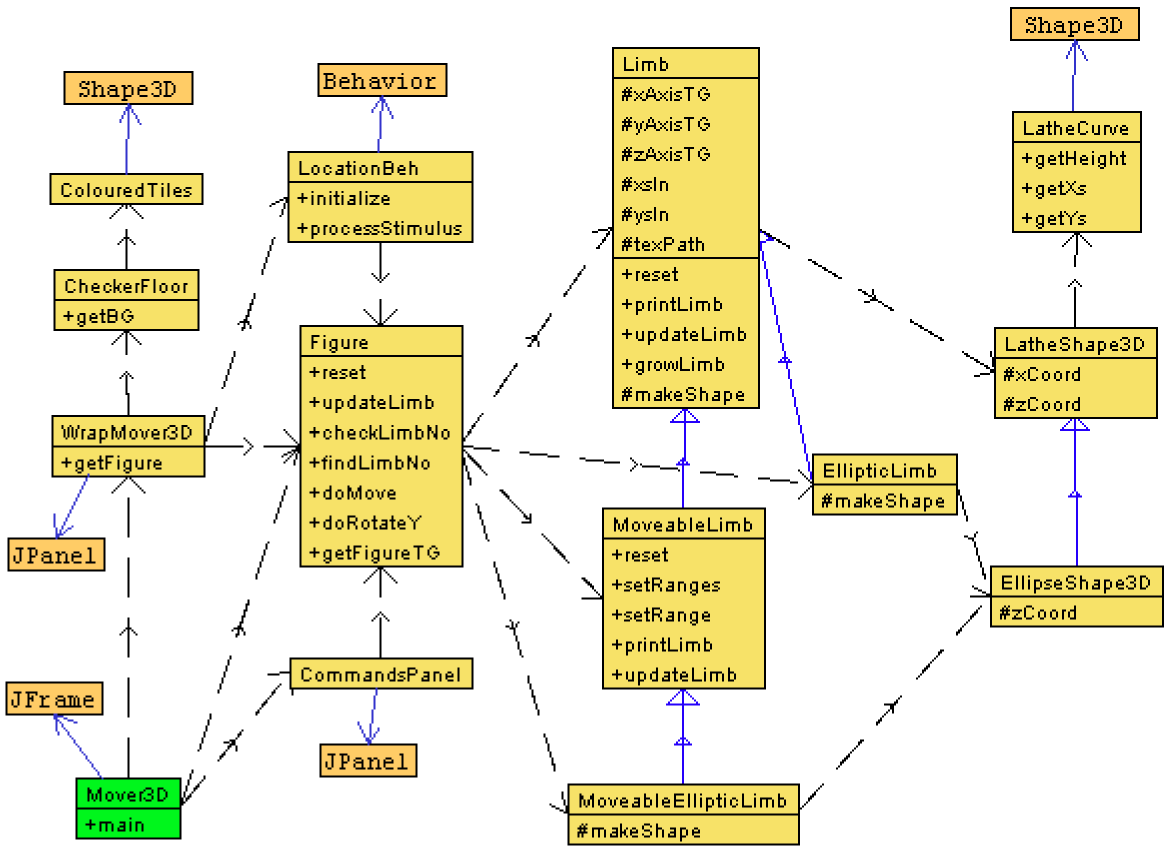


Figure 8. UML Class Diagrams for Mover3D.

Mover3D is the top-level JFrame for the application, containing two JPanels. The 3D scene is created in WrapMover3D and displayed in its panel, while the commands textfield and reset button are managed by CommandsPanel. The LocationBeh behavior deals with user input via the keyboard, and enables the figure to be moved and rotated.

WrapMover3D creates the usual checkboard scene, using the CheckerFloor and ColouredTiles classes (first described in chapter 9) to create the floor.

The scene contains a single Figure object which represents the figure as a series of connected shapes created from the Limb class and its subclasses (MoveableLimb, MoveableEllipticLimb, and EllipticLimb).

The shape of a limb is specified using a LatheShape3D or EllipseShape3D object, which were described in chapter 9.5.

3. The WrapMover3D Class

WrapMover3D is just like previous 'wrap' classes: it creates a 3D scene inside a JPanel, made up of a checkboard floor, blue sky, lighting, and an OrbitBehavior node to allow the user to adjust the viewpoint. Much of this is done in the createSceneGraph() method:

```
private void createSceneGraph()
{
  sceneBG = new BranchGroup();
  bounds = new BoundingSphere(new Point3d(0,0,0), BOUNDSIZE);

  lightScene();          // add the lights
  addBackground();       // add the sky
  sceneBG.addChild( new CheckerFloor().getBG() );  // add the floor
  addFigure();
  sceneBG.compile();   // fix the scene
}
```

The code which distinguishes WrapMover3D from earlier 'wrap' classes is mostly contained in addFigure():

```
// global var: the multi-limbed figure
private Figure figure;
   :

private void addFigure()
// add the figure and its behaviour to the scene
{
  figure = new Figure();
  sceneBG.addChild( figure.getFigureTG() );  // add figure's TG

  // add behavior
  LocationBeh locBeh = new LocationBeh( figure );
  locBeh.setSchedulingBounds(bounds);
  sceneBG.addChild(locBeh);
}
```

The Figure object constructs the articulated figure, and its top-level TransformGroup is added to the scene. The LocationBeh object converts user key presses into figure commands.

Mover3D uses the getFigure() method to obtain a reference to the Figure object, which it passes to the CommandsPanel.

```
public Figure getFigure()
{ return figure; }
```

4. The LocationBeh Class

The figure can be moved forward, back, left, right, up, or down, and turned clockwise or anti-clockwise around the y-axis via the arrow keys. The code is very similar to that in the TourControls class in Tour3D (chapter 10), but simpler in many cases since there is no viewpoint manipulation.

The figure reference is passed in through the constructor:

```
private Figure figure;  // global
  :

public LocationBeh(Figure fig)
{ figure = fig;
      :
}
```

If a key is pressed along with <alt> then altMove() is called, otherwise standardMove(). Both methods are quite similar in style; for instance, the standardMove() method:

```
private void standardMove(int keycode)
// moves figure forwards, backwards, rotates left or right
{
  if(keycode == forwardKey)
    figure.doMove(FWD);
  else if(keycode == backKey)
    figure.doMove(BACK);
  else if(keycode == leftKey)
    figure.doRotateY(CLOCK);   // clockwise
  else if(keycode == rightKey)
    figure.doRotateY(CCLOCK);    // counter-clockwise
} // end of standardMove()
```

The constants (FWD, BACK, etc.) are integers.

The keys are processed by calling doMove() and doRotateY() in the Figure object.

5. The CommandsPanel Class

CommandsPanel creates the panel at the bottom of the GUI containing the textfield and reset button. Much of the code deals with the parsing of the input from the textfield, which takes two forms. A *limb command* has the format:

(<limbName> | <limbNo>) (`fwd` | `f` | `turn` | `t` | `side` | `s`) [angleChg]

A *figure command* has the form:

(`fwd` | `f` | `back` | `b` | `left` | `l` | `right` | `r` | `up` | `u` | `down` | `d` | `clock` | `c` | `cclock` | `cc`)

Each moveable limb is assigned a name and number, and either ID can be used to refer to it. As a convenience, the name/number mappings are printed to standard output when Mover3D is started.

The rotation operations refer to the three axes:

- `fwd` (or `f`) = positive rotation around x-axis
- `turn` (or `t`) = positive rotation around y-axis
- `side` (or `s`) = positive rotation around z-axis

If an angleChg value is not included, a default angle of 5 degrees is used. angleChg can be negative. A angle which would take a limb outside of its specified range is ignored (and an error message printed).

The figure commands all have keyboard equivalents, processed by LocationBeh.

An advantage of textfield input is the ability to group several limb and/or figure commands together, separated by ','s. These are all processed before the figure is redrawn. By pressing <enter>, a complex sequence of commands is easily repeated.

The string entered in the textfield is pulled apart in processComms() which separates out the individual commands and extracts the 2 or 3 argument limb action or single argument figure operation.

```
private void processComms(String input)
  { if (input == null)
      return;

    String[] commands = input.split(",");  // split into commands
    StringTokenizer toks;
    for (int i=0; i < commands.length; i++) {
      toks = new StringTokenizer( commands[i].trim() );
      if (toks.countTokens() == 3)         // 3-arg limb command
        limbCommand( toks.nextToken(), toks.nextToken(),
                                      toks.nextToken() );
      else if (toks.countTokens() == 2)    // 2-arg limb command
        limbCommand( toks.nextToken(), toks.nextToken(), "5");
      else if (toks.countTokens() == 1)    // 1-arg figure command
        figCommand( toks.nextToken() );
      else
        System.out.println("Illegal command: " + commands[i]);
    }
  }
```

limbCommand() must extract the limb number, the axis of rotation, and the rotation angle from the command string. If a limb name has been entered, then the corresponding number is obtained by querying the Figure object.

```
private void limbCommand(String limbName, String opStr,
                                          String angleStr)
{ // get the limb number
  int limbNo = -1;
  try {
    limbNo = figure.checkLimbNo( Integer.parseInt(limbName) );
  }
  catch(NumberFormatException e)
  {  limbNo = figure.findLimbNo(limbName);  }   // map name to num
  if (limbNo == -1) {
    System.out.println("Illegal Limb name/no: " + limbName);
    return;
  }

  // get the angle change
  double angleChg = 0;
  try {
    angleChg = Double.parseDouble(angleStr);
  }
  catch(NumberFormatException e)
  { System.out.println("Illegal angle change: " + angleStr); }
  if (angleChg == 0) {
    System.out.println("Angle change is 0, so doing nothing");
    return;
  }

  // extract the axis of rotation from the limb operation
  int axis;
  if (opStr.equals("fwd") || opStr.equals("f"))
    axis = X_AXIS;
  else if (opStr.equals("turn") || opStr.equals("t"))
    axis = Y_AXIS;
  else if (opStr.equals("side") || opStr.equals("s"))
    axis = Z_AXIS;
  else {
    System.out.println("Unknown limb operation: " + opStr);
    return;
  }

  // apply the command to the limb
  figure.updateLimb(limbNo, axis, angleChg);

}   // end of limbCommand()
```

The handling of possible parsing errors lengthens the code. The limb number is checked via a call to checkLimbNo() in Figure, which scans the Limbs to determine if the specified number is used by one of them. The mapping of a limb name to number is carried out by Figure's findLimbNo(), which returns –1 if the name is not found amongst the limbs.

Once the correct input has been gathered, it is passed to updateLimb() in the Figure object.

A figure command is processed by figCommand() which uses a multi-way branch to convert the command into a correctly parameterized call to Figure's doMove() or doRotateY() method.

```
private void figCommand(String opStr)
{ if (opStr.equals("fwd") || opStr.equals("f"))
    figure.doMove(FWD);
  else if (opStr.equals("back") || opStr.equals("b"))
    figure.doMove(BACK);
  else if (opStr.equals("left") || opStr.equals("l"))
    figure.doMove(LEFT);
  else if (opStr.equals("right") || opStr.equals("r"))
    figure.doMove(RIGHT);
  else if (opStr.equals("up") || opStr.equals("u"))
    figure.doMove(UP);
  else if (opStr.equals("down") || opStr.equals("d"))
    figure.doMove(DOWN);
  else if (opStr.equals("clock") || opStr.equals("c"))
    figure.doRotateY(CLOCK);
  else if (opStr.equals("cclock") || opStr.equals("cc"))
    figure.doRotateY(CCLOCK);
  else {
    System.out.println("Unknown figure operation: " + opStr);
    return;
  }
} // end of figCommand()
```

6. The Figure Class

The Figure class carries out three main tasks:

1) it builds the figure by connecting Limb objects. The resulting figure is then translated into a Java 3D subgraph;
2) it processes limb-related operations, such as updateLimb() calls;
3) it processes figure movement operations, such as doRotateY().

6.1. Building the Figure

Task (1) is started in the Figure constructor:

```
//global vars
private ArrayList limbs;
  // Arraylist of Limb objects, indexed by limb number
private HashMap limbNames;
  // holds (limb name, limb number) pairs

private TransformGroup figureTG;
  // the top-level TG for the entire figure
    :

public Figure()
{
  yCount = 0;      // the figure is on the floor initially
  t3d = new Transform3D();   // used for repeated calcs
  toMove = new Transform3D();
  toRot = new Transform3D();

  limbs = new ArrayList();
  limbNames = new HashMap();
```

```
    // construct the figure from connected Limb objects
    buildTorso();
    buildHead();

    buildRightArm();
    buildLeftArm();

    buildRightLeg();
    buildLeftLeg();

    printLimbsInfo();

    buildFigureGraph();   // convert figure into a Java 3D subgraph
  }  // end of Figure()
```

The figure's component limb objects are stored in the limbs ArrayList. A limb object is stored in the ArrayList at the index position specified by its limb number.

A limbNames HashMap stores (limb name, limb number) pairs. It is used to determine a limb's number when a limb name is supplied in a limb command.

The various 'build' methods could be combined into a single large function, with a corresponding loss of clarity.

Figure 9 shows the limbs that comprise the figure, labeled with their names and numbers.

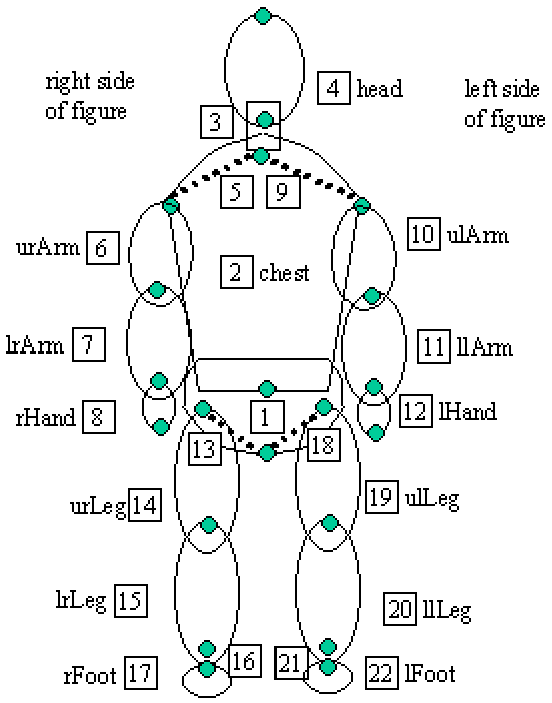


Figure 9. The Figure's Limbs, Named and Numbered.

Only moveable limbs have names, which excludes the neck and bottom. Invisible limbs are also nameless, and are marked as dotted lines in the figure. There are two very short invisible limbs linking the legs to their feet (labeled as 16 and 21 in Figure 9).

The small green (grey) circles in Figure 9 are the joints – the points where limbs connect to each other; they are positioned to make the limbs overlap.

Figure 10 shows the articulated figure again but with emphasis given to the joints. Each arrow shows the positive y-axis in the limb's local coordinate space. A limb's shape extends from the joint, following the arrow's direction.

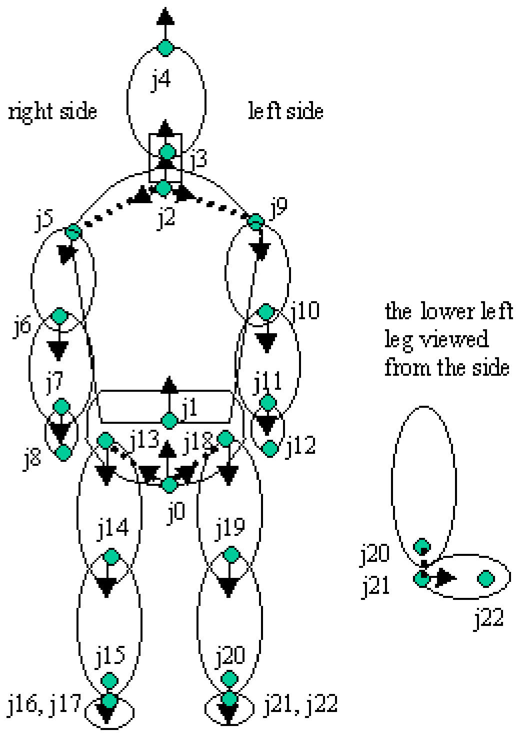


Figure 10. The Figure's Joints.

The first joint in the figure is "j0", which is the starting location of the bottom. The chest limb begins at joint "j1", the neck at "j2", the upper left arm at "j9", the lower left arm at "j10", and so on. The side view of the lower left leg shows the invisible joint which begins at j20 extending downwards. The foot is attached to it via "j21".

The arrows on the joints show that the local y-axis for a limb can be rotated quite significantly when viewed in world coordinates. For example, the 'base' of the upper left arm is at "j9", and the limb's positive y-axis is pointing almost straight down.

It is possible for several limbs to be attached to one joint. For instance, "j0" is the starting point for the 'bottom' limb and two invisible limbs which extend up and to the left and right respectively.

Each limb utilizes two joints. In the joint's local coordinate system, the *start joint* begins at (0,0) on its XZ plane. The limb's shape is placed at the start joint location, and oriented along the positive y-axis. The *end joint* is positioned along the limb's y-axis, 90% of the way towards the end of the limb's shape. For example, the upper left arm's start joint is "j9", its end joint "j10". The lower left arm's start joint is "j10", thereby linking the lower arm to the upper.

A limb's joints are encoded in Java 3D as TransformGroups. The start joint TransformGroup of a child limb is the end joint TransformGroup of its parent, so linking the child to the parent.

Figure 11 shows the articulated figure again, but in terms of the TransformGroups which encode the joints.

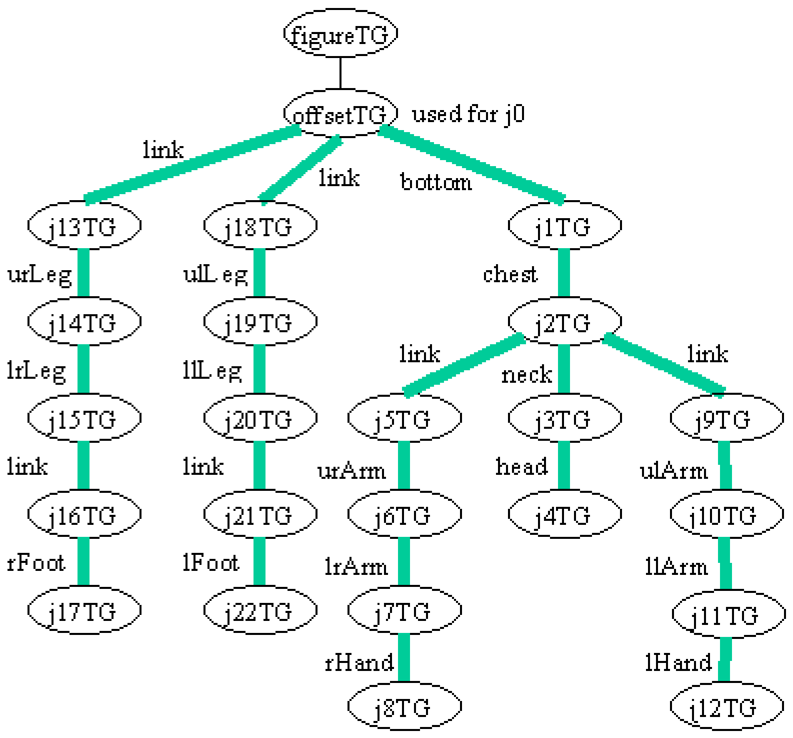


Figure 11. The Figure's TransformGroups.

The thick green (grey) lines denote the limbs, and hide several TransformGroups and other Java 3D nodes which will be discussed below. The visible TransformGroups are for the joints, and are labeled with their joint name and "TG".

For instance, the limb for the upper left arm (ulArm), starts at joint "j9", and its end joint is "j10". The limb for the lower left arm (llArm) is attached to the TransformGroup for "j10" and so becomes its child.

The green lines labeled with "link" are invisible limbs, which have no names.

Figure 11 also shows the top-level TransformGroups for the figure: figureTG and offsetTG. figureTG represents the origin for the entire figure and is located on the floor, initially at (0,0). figureTG is affected by figure commands. offsetTG is a vertical offset, up off the floor, which corresponds to the "j0" start joint.

The details of Limb creation in the Figure object depend on the type of Limb being created. Figure 12 shows the hierarchy for Limb and its subclasses.

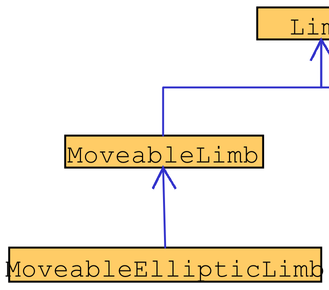


Figure 12. The Limb Class Hierarchy.

Limb defines the appearance of a limb (using a lathe shape), and how it is connected to a parent limb via a joint (TransformGroup). The limb's initial orientation is set. Limb and EllipticLimb cannot be moved, and do not use limb names.

The MoveableLimb and MoveableEllipticLimb classes are moveable. They have limb names, and x-, y-, z- axis rotation ranges. If a range is not specified, then it is assumed to be 0 (i.e. rotation is not possible around that axis).

The lathe shape used in a Limb or MoveableLimb object has a circular cross-section, but is elliptical in EllipticLimb and MoveableEllipticLimb. Lathe shapes were described in chapter 9.5.

buildTorso() from the Figure class shows the use of EllipticLimb and MoveableEllipticLimb to create the bottom and chest for the figure. The bottom is not moveable, the chest is.

```
private void buildTorso()
{
  // the figure's bottom
  double xsIn1[] = {0, -0.1, 0.22, -0.2, 0.001};
  double ysIn1[] = {0, 0.03, 0.08, 0.25, 0.25};
  EllipticLimb limb1 = new EllipticLimb(
        1, "j0", "j1", Z_AXIS, 0, xsIn1, ysIn1, "denim.jpg");
  // no movement, so no name or ranges

  // the figure's chest: moveable so has a name ("chest")
  // and rotation ranges
  double xsIn2[] = {-0.001, -0.2, 0.36, 0.001};
  double ysIn2[] = {0, 0, 0.50, 0.68};
  MoveableEllipticLimb limb2 = new MoveableEllipticLimb("chest",
       2, "j1", "j2", Z_AXIS, 0, xsIn2, ysIn2, "camoflage.jpg");
  limb2.setRanges(0, 120, -60, 60, -40, 40);
       // x range: 0 to 120; y range: -60 to 60; z range: -40 to 40

  limbs.add(limb1);
  limbs.add(limb2);

  limbNames.put("chest", new Integer(2)); // store (name,number)
}  // end of buildTorso()
```

The arrays of coordinates passed to the limb1 and limb2 objects define the lathe curves for the bottom and chest. Figure 13 shows the graph of points making up the curve for the bottom (limb1).

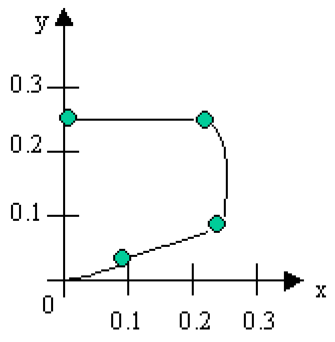


Figure 13. The Lathe Curve for the Figure's Bottom.

A limb requires a limb number, its start and end joint names, an axis of orientation and angle to that axis, and lathe shape coordinates and texture. The lathe shape and texture can be left out, to signal that the limb should be invisible, but a limb length must be supplied instead.

If the limb is moveable (i.e. a MoveableLimb or MoveableEllipticLimb object), then it also requires a name and x-, y-, z- ranges to restrict its movements.

The bottom limb is defined as:

```
EllipticLimb limb1 = new EllipticLimb(
      1, "j0", "j1", Z_AXIS, 0, xsIn1, ysIn1, "denim.jpg");
```

This is a non-moveable limb, so has no name. Its limb number is 1, it starts at joint "j0", and its end joint is called "j1". It is rotated around the z- axis by 0 degrees (i.e. not rotated at all), and has a lathe shape covered in denim. These details can be checked against the information in Figures 9 and 10.

The chest limb is:

```
MoveableEllipticLimb limb2 = new MoveableEllipticLimb("chest",
     2, "j1", "j2", Z_AXIS, 0, xsIn2, ysIn2, "camoflage.jpg");
limb2.setRanges(0, 120, -60, 60, -40, 40);
     // x range: 0 to 120; y range: -60 to 60; z range: -40 to 40
```

This moveable limb is called "chest", limb number 2. Its start joint is "j1", and so will become a child of the bottom limb. Its end joint is called "j2". It is rotated around the z- axis by 0 degrees (i.e. not rotated at all), and has a lathe shape covered in a camouflage pattern. The permitted ranges for rotation around the x-, y-, and z- axes are set with a call to setRanges().

The end of buildTorso() shows the two limbs being added to the limbs ArrayList. A limb numbered as X can be found in the list by looking up entry X-1.

The names of named limbs (in this case only "chest") are added to a limbNames HashMap together with their limb numbers. This data structure is used when a limb is referred to by its name, and its corresponding number must be found.

Orientating Limbs

The construction of the left arm illustrates how the initial orientation of a limb can be adjusted. Figure 14 shows the construction of the left arm, including the angles between the limbs.

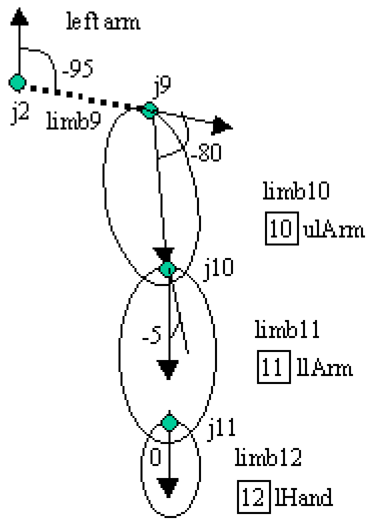


Figure 14. The Left Arm in Detail.

The relevant code is in buildLeftArm():

```
private void buildLeftArm()
{
  // invisible limb connecting the neck and upper left arm
  Limb limb9 = new Limb(9, "j2", "j9", Z_AXIS, -95, 0.35);

  // upper left arm
  double xsIn10[] = {0, 0.1, 0.08, 0};
  double ysIn10[] = {0, 0.08, 0.45, 0.55};
  MoveableLimb limb10 = new MoveableLimb("ulArm",
      10, "j9", "j10", Z_AXIS, -80, xsIn10, ysIn10, "leftarm.jpg");
  limb10.setRanges(-60, 180, -90, 90, -30, 90);

  // lower left arm
  double xsIn11[] = {0, 0.08, 0.055, 0};
  double ysIn11[] = {0, 0.08, 0.38, 0.43};
  MoveableLimb limb11 = new MoveableLimb("llArm",
        11, "j10", "j11", Z_AXIS, -5, xsIn11, ysIn11, "skin.jpg");
  limb11.setRanges(0, 150, -90, 90, -90, 90);

  // left hand
  double xsIn12[] = {0, 0.06, 0.04, 0};
  double ysIn12[] = {0, 0.07, 0.16, 0.2};
  MoveableEllipticLimb limb12 = new MoveableEllipticLimb("lHand",
       12, "j11", "j12", Z_AXIS, 0, xsIn12, ysIn12, "skin.jpg");
```

```
    limb12.setRanges(-50, 50, -90, 40, -40, 40);

    limbs.add(limb9);
    limbs.add(limb10);
    limbs.add(limb11);
    limbs.add(limb12);

    limbNames.put("ulArm", new Integer(10));
    limbNames.put("llArm", new Integer(11));
    limbNames.put("lHand", new Integer(12));
  }  // end of buildLeftArm()
```

The invisible limb, limb9, is made 0.35 units long., and rotated around the z-axis by 95 degrees:

```
     Limb limb9 = new Limb(9, "j2", "j9", Z_AXIS, -95, 0.35);
```

The rotation turns the y-axis of limb9 clockwise by 95 degrees. However, the actual orientation of the limb in world coordinate space depends on the overall orientation of the y-axis specified by its ancestors. In this case, none of its ancestors (the bottom and chest) have been rotated, and so its world orientation is the same as its local value.

The limb for the upper left arm is defined as:

```
    MoveableLimb limb10 = new MoveableLimb("ulArm",
        10, "j9", "j10", Z_AXIS, -80, xsIn10, ysIn10, "leftarm.jpg");
```

This rotates the y-axis of limb10 clockwise by 80 degrees, which when added to the ancestor rotations (bottom, chest, limb9) means that the shape is almost pointing downwards, with a total rotation of 175 degrees.

The lower arm (limb11; llArm) is rotated another 5 degrees to point straight down. The left hand (limb12; lHand) has no rotation of its own, and so also points downwards.

Creating the Scene Graph

The 'build' methods (e.g. buildTorso(), buildLeftArm()) create the limb objects, and specify how they are linked in terms of joint names. The creation of the scene graph outlined in Figure 11 is initiated by buildFigureGraph() after all the limbs have been initialized.

```
  private void buildFigureGraph()
  {
    HashMap joints = new HashMap();
    /* joints will contain (jointName, TG) pairs. Each TG is the
       position of the joint in the scene.
       A limb connected to a joint is placed in the scene by
       using the TG associated with that joint.
    */
    figureTG = new TransformGroup();
    figureTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
    figureTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);
    TransformGroup offsetTG = new TransformGroup();
    Transform3D trans = new Transform3D();
    trans.setTranslation( new Vector3d(0, 1.24, 0));
          // an offset from the ground to the first joint
    offsetTG.setTransform( trans );

    joints.put("j0", offsetTG);   // store starting joint j0

    /* Grow the subgraph for each limb object, attaching it
       to the figure's subgraph below offsetTG. */
    Limb li;
    for (int i = 0; i < limbs.size(); i++) {
      li = (Limb)limbs.get(i);
      li.growLimb(joints);
    }

    figureTG.addChild(offsetTG);
  }  // end of buildFigureGraph()
```

buildFigureGraph() initializes figureTG and offsetTG. offsetTG will be the TransformGroup for the first joint, “j0”, and its name/TransformGroup pair is stored in a HashMap called joints.

A for-loop iterates through the Limb objects stored in the limbs ArrayList, and calls each limb's growLimb() method, passing in the HashMap. growLimb() creates a Java 3D subbranch for the limb and attaches it to the TransformGroup corresponding to the limb’s start joint. This joint/TransformGroup correspondence is found by searching the joints HashMap.

A subtle assumption of this coding is that a ‘child’ limb is never attached to a joint before the joint has been converted into a TransformGroup. Another way of understanding this is that a parent limb must be converted to a Java 3D subbranch before any of its children.

6.2. Processing Limb-related Operations

The Figure class uses the limbNames HashMap, which contains limb name / limb number pairs to check if a user-supplied limb number is actually used by the figure, and also to convert limb names into numbers. These operations are done by checkLimbNo() and findLimbNo().

updateLimb() is called with a legal limb number, an axis of rotation, and a rotation angle, and passes the request on to the limb in question:

```
  public void updateLimb(int limbNo, int axis, double angle)
  { Limb li = (Limb) limbs.get(limbNo-1);
    li.updateLimb(axis, angle);  // pass on axis and angle
  }
```

reset() is called by CommandsPanel when the user presses the reset button. The reset request is sent on to every limb.

```
  public void reset()
  // restore each limb to its original position in space
  { Limb li;
    for (int i = 0; i < limbs.size(); i++) {
      li = (Limb)limbs.get(i);
      li.reset();
    }
  }
```

6.3. Figure Movement

Figure commands, such as "forward" and "clock", are converted into transforms applied to the figureTG TransformGroup at the root of the figure's subgraph.

doMove() converts a move request into a translation vector, which is applied in doMove1():

```
  private void doMove1(Vector3d theMove)
  // Move the figure by the amount in theMove
  {
    figureTG.getTransform( t3d );
    toMove.setTranslation(theMove);    // overwrite previous trans
    t3d.mul(toMove);
    figureTG.setTransform(t3d);
  }
```

toMove and t3d are global Transform3D variables which are reused by doMove1() in order to avoid the overhead of object creation and garbage collection.

doRotateY() converts a rotation request into a rotation around the y-axis, which is carried out by doRotateY1():

```
  private void doRotateY1(double radians)
  // Rotate the figure by radians amount around its y-axis
  {
    figureTG.getTransform( t3d );
    toRot.rotY(radians);   // overwrite previous rotation
    t3d.mul(toRot);
    figureTG.setTransform(t3d);
  }
```

toRot is a global Transorm3D variable.

A drawback of this implementation is the lack of x- and z- axis rotations which makes it impossible to position the figure in certain ways. For instance, we cannot make the figure stand on its hands, which would involve a rotation around the x-axis.

Adding this functionality would not be difficult. We would add two extra TransformGroups below figureTG so that the three rotation axes could be cleanly separated, and be easily 'reset'. This coding strategy is used for limb rotation, as seen below.

7. The Limb Class

The main job of the Limb class is to convert limb information into a Java 3D subgraph, like the one in Figure 15.

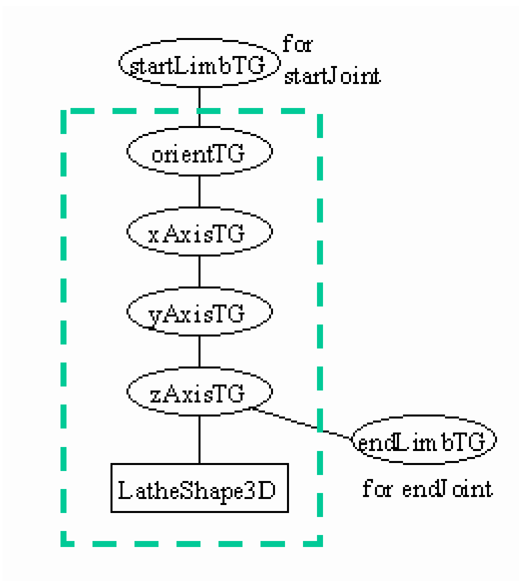


Figure 15. The Subgraph Representing a Limb.

The start and end joints are represented by TransformGroups. startLimbTG is not created by the limb, but obtained from the parent limb. It is the parent's endLimbTG, and in this way are children attached to parents.

The limb creates endLimbTG, which is positioned along the y-axis, 90% of the way along the length of the limb's shape. Child limbs can be attached to endLimbTG, meaning that they will slightly overlap the parent limb shape. This enhances the effect that the limbs are connected, especially when a limb is rotated.

In between the joint TransformGroups are four more TransformGroups and a LatheShape3D node representing the limb shape and its position. These are the details hidden by the thick green (grey) lines between the TransformGroups in Figure 11. Each of those lines should be expanded into the five nodes surrounded by the green (grey) dotted box in Figure 15.

orientTG is used to orientate the shape initially. The other TransformGroups are located below it as its children, so they view the new orientation as pointing along the positive y-axis. The xAxisTG, yAxisTG, and zAxisTG TransformGroups are employed to rotate the limb around the x-, y-, and z- axes at run time. The separation of these rotations into three parts makes it much easier to undo them if the limb is reset.

The LatheShape3D object is the lathe shape, whose base is positioned where the startLimbTG is located, and points up the local y-axis.

Although the Limb class creates the Figure 15 subgraph, it does not allow the xAxisTG, yAxisTG, or zAxisTGs to be affected. The MoveableLimb class offers implementations of the methods which adjust these TransformGroups.

Limb contains a variety of limb data, supplied by its constructor:

```
private int limbNo;
private String startJoint, endJoint;
private int orientAxis;       // limb's axis of initial orientation
private double orientAngle = 0;      // angle to orientation axis

private double limbLen;
private boolean visibleLimb = false;

protected double xsIn[], ysIn[];     // coordinates of lathe shape
protected String texPath;            // shape's texture filename
```

The Limb class does not have a limb name; only moveable limbs utilize names.

The length of a limb is usually obtained from the lathe shape coordinates. We assume that the final value in the lathe shape's y-coordinates is the maximum y-value for the entire shape (i.e. its height). If the limb is to be invisible, then the constructor includes a limb length, which is directly assigned to limbLen.

The visibleLimb boolean is used to distinguish between visible and invisible limbs.

The lathe shape coordinates and texture are set to be protected, since they need to be accessible by Limb subclasses which override the lathe shape creation method, makeShape().

growLimb() starts the process of subgraph creation for the limb:

```
public void growLimb(HashMap joints)
{
  TransformGroup startLimbTG =
        (TransformGroup) joints.get(startJoint);
  if (startLimbTG == null)
    System.out.println("No transform group for " + startJoint);
  else {
    setOrientation(startLimbTG);
    makeLimb(joints);
  }
}
```

The start joint name is used to find the startLimbTG TransformGroup in the joints HashMap. This should have already been created by the parent of this limb.

setOrientation() creates the four rotational TransformGroups (orientTG, xAxisTg, yAxisTG, and zAxisTG) below startLimbTG.

```
private void setOrientation(TransformGroup tg)
{
  TransformGroup orientTG = new TransformGroup();
  if (orientAngle != 0) {
    Transform3D trans = new Transform3D();
    if (orientAxis == X_AXIS)
```

```
        trans.rotX( Math.toRadians(orientAngle));
      else if (orientAxis == Y_AXIS)
         trans.rotY( Math.toRadians(orientAngle));
      else    // must be z-axis
        trans.rotZ( Math.toRadians(orientAngle));
      orientTG.setTransform(trans);
    }

    xAxisTG = new TransformGroup();
    xAxisTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
    xAxisTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

    yAxisTG = new TransformGroup();
    yAxisTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
    yAxisTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

    zAxisTG = new TransformGroup();
    zAxisTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
    zAxisTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

    // scene graph's sequence of TG's
    tg.addChild(orientTG);
    orientTG.addChild(xAxisTG);
    xAxisTG.addChild(yAxisTG);
    yAxisTG.addChild(zAxisTG);
  } // end of setOrientation()
```

The capability bits are set to allow the axis TransformGroups to change during execution, but orientTG remains fixed after being positioned at build time.

makeLimb() creates the endLimbTG TransformGroup and may create a lathe shape if the limb is set to be visible.

```
  private void makeLimb(HashMap joints)
  {
    if (visibleLimb)
      makeShape();  // create the lathe shape

    TransformGroup endLimbTG = new TransformGroup();
    Transform3D trans = new Transform3D();
    trans.setTranslation(
         new Vector3d(0.0, limbLen*(1.0-OVERLAP), 0.0) );
    /* The end position is just short of the actual length of the
       limb so that any child limbs will be placed so they overlap
       with this one. */
    endLimbTG.setTransform(trans);
    zAxisTG.addChild(endLimbTG);

    joints.put(endJoint, endLimbTG);    // store (jointName, TG) pair
  }
```

The endLimbTG TransformGroup is stored in the joints HashMap at the end of the method, so is available for use by this limb's children.

makeShape() creates a LatheShape3D object and attaches it to the zAxisTG node.

```
protected void makeShape()
{
  LatheShape3D ls;
  if (texPath != null) {
    TextureLoader texLd =
            new TextureLoader("textures/"+texPath, null);
    Texture tex = texLd.getTexture();
    ls = new LatheShape3D(xsIn, ysIn, tex);
  }
  else
    ls = new LatheShape3D(xsIn, ysIn, null);

  zAxisTG.addChild(ls);  // add the shape to the limb's graph
}  // end of makeShape()
```

makeShape() is a protected method, since it may be overridden by Limb's subclasses. For example, EllipticLimb replaces the call to LatheShape3D by EllipseShape3D. This causes the limb to have an elliptical cross-section.

Limb() contains empty updateLimb() and reset() methods:

```
public void updateLimb(int axis, double angleStep) {}

public void reset() {}
```

updateLimb() and reset() affect the position of the limb, and so are not used in Limb. They are overridden by the MoveableLimb subclass.

7.1. The MoveableLimb Class

MoveableLimb allows a limb to be moved around the x-, y-, and z-axes. This is achieved by affecting the xAxisTG, yAxisTG, and zAxisTG TransformGroups in the limb's subgraph.

MoveableLimb maintains range information for the three axes, and ignores rotations which would move the limb outside of those ranges. If a range is not specified, then it is assumed to be 0 (i.e. rotation is not possible around that axis). The programmer calls setRanges() or setRange() to initialize the range details for different axes:

```
// global vars: the axis ranges
private double xMin, xMax, yMin, yMax, zMin, zMax;
      :

public void setRanges(double x1, double x2, double y1, double y2,
                                double z1, double z2)
{ setRange(X_AXIS, x1, x2);
  setRange(Y_AXIS, y1, y2);
  setRange(Z_AXIS, z1, z2);
}
```

```
public void setRange(int axis, double angle1, double angle2)
// set the range for axis only
{
  if (angle1 > angle2) {
    System.out.println(limbName + ": wrong order... swapping");
    double temp = angle1;
    angle1 = angle2;
    angle2 = temp;
  }
  if (axis == X_AXIS) {
    xMin = angle1;  xMax = angle2;
  }
  else if (axis == Y_AXIS) {
    yMin = angle1;  yMax = angle2;
  }
  else {  // Z_AXIS
    zMin = angle1;  zMax = angle2;
  }
}  // end of setRange()
```

The methods initialize the xMin, xMax, yMin, yMax, zMin, and zMax globals, and ensure that the ranges are given in the right order.

Rotations are processed by updateLimb() which is called from the Figure object with axis and angle arguments.

```
public void updateLimb(int axis, double angleStep)
// Attempt to rotate this limb by angleStep around axis
{
  if (axis == X_AXIS)
    applyAngleStep(angleStep, xCurrAng, axis, xMax, xMin);
  else if (axis == Y_AXIS)
    applyAngleStep(angleStep, yCurrAng, axis, yMax, yMin);
  else    // Z_AXIS
    applyAngleStep(angleStep, zCurrAng, axis, zMax, zMin);
}



private void applyAngleStep(double angleStep, double currAngle,
                                int axis, double max, double min)
/* Before any rotation, check that the angle step moves the
   limb within the ranges for this axis.
   If not then rotate to the range limit, and no further. */
{
  if ((currAngle >= max) && (angleStep > 0)) {  // will exceed max
    System.out.println(limbName + ": no rot; already at max");
    return;
  }
  if (currAngle <= min && (angleStep < 0)) { // will drop below min
    System.out.println(limbName + ": no rot; already at min");
    return;
  }

  double newAngle = currAngle + angleStep;
  if (newAngle > max) {
    System.out.println(limbName + ": reached max angle");
    angleStep = max - currAngle;   // rotate to max angle only
  }
  else if (newAngle < min) {
    System.out.println(limbName + ": reached min angle");
    angleStep = min - currAngle;   // rotate to min angle only
  }

  makeUpdate(axis, angleStep);    // do the rotation
}  // end of applyAngleStep()
```

updateLimb() uses the supplied axis value to pass the correct axis range to applyAngleStep(). Its purpose is to check that the requested rotation stays within the range. This may mean ignoring the rotation (if the max or min has already been reached), or reducing the rotation so that the limb stops at the max or min value. Once the actual rotation angle has been calculated (and stored in angleStep), then makeUpdate() is called.

```
  // global vars: the current angle in 3 axes
  private double xCurrAng, yCurrAng, zCurrAng;
    :

  private void makeUpdate(int axis, double angleStep)
  // rotate the limb by angleStep around the given axis
  {
    if (axis == X_AXIS) {
      rotTrans.rotX( Math.toRadians(angleStep));
      xAxisTG.getTransform(currTrans);
      currTrans.mul(rotTrans);
      xAxisTG.setTransform(currTrans);
      xCurrAng += angleStep;
    }
    else if (axis == Y_AXIS) {
      rotTrans.rotY( Math.toRadians(angleStep));
      yAxisTG.getTransform(currTrans);
      currTrans.mul(rotTrans);
      yAxisTG.setTransform(currTrans);
      yCurrAng += angleStep;
    }
    else {  // z-axis
      rotTrans.rotZ( Math.toRadians(angleStep));
      zAxisTG.getTransform(currTrans);
      currTrans.mul(rotTrans);
      zAxisTG.setTransform(currTrans);
      zCurrAng += angleStep;
    }
  }
```

makeUpdate() applies a rotation to one of xAxisTG, yAxisTG, or zAxisTG depending on the axis value supplied by the user. The rotational transform is multiplied to the current value held in the relevant TransformGroup, which is equivalent to 'adding' the rotation to the current angle.

rotTrans and currTrans are global Transform3D variables to save on the cost of object creation and deletion.

The new limb angle is stored in xCurrAng, yCurrAng, or zCurrAng.

The limb can be reset to its initial orientation, via a call to reset() by the Figure object.

```
public void reset()
{
  rotTrans.rotX( Math.toRadians(-xCurrAng));   // reset x angle
  xAxisTG.getTransform(currTrans);
  currTrans.mul(rotTrans);
  xAxisTG.setTransform(currTrans);
  xCurrAng = 0;

  rotTrans.rotY( Math.toRadians(-yCurrAng));    // reset y angle
  yAxisTG.getTransform(currTrans);
  currTrans.mul(rotTrans);
  yAxisTG.setTransform(currTrans);
  yCurrAng = 0;

  rotTrans.rotZ( Math.toRadians(-zCurrAng));    // reset z angle
  zAxisTG.getTransform(currTrans);
  currTrans.mul(rotTrans);
  zAxisTG.setTransform(currTrans);
  zCurrAng = 0;
}  // end of reset()
```

The rotations maintained by xAxisTG, yAxisTG, and zAxisTG are undone by rotating each one by the negative of their current angle, as stored in xCurrAng, yCurrAng, and zCurrAng.

The simplicity of this operation is due to the separation of the three degrees of freedom into three TransformGroups.

7.2. The MoveableEllipticLimb Class

MoveableEllipticLimb shows how little code is required in order to adjust the limb's shape. Only makeShape() must be overridden to use EllipseShape3D instead of the version in the Limb class which utilizes LatheShape3D.

```
protected void makeShape()
{
  EllipseShape3D es;
  if (texPath != null) {
    TextureLoader texLd =
        new TextureLoader("textures/"+texPath, null);
    Texture tex = texLd.getTexture();
    es = new EllipseShape3D(xsIn, ysIn, tex);
  }
  else
    es = new EllipseShape3D(xsIn, ysIn, null);

  zAxisTG.addChild(es);  // add the shape to the limb's graph
}
```

8. Other Articulated Figure Approaches

The H-Anim (Humanoid Animation) Working Group (http://www.h-anim.org/) has developed a VRML97 specification for representing figures.

There is a Joint node for defining limb position, orientation, and other attributes such as skin properties. Joints are linked together to form a hierarchy, and so Joint is somewhat similar to the Limb class developed here.

The Segment node is concerned with the shape of the body part, including its mass, and allows the shape's geometry to be adjusted. Segment could be equated with the LatheShape3D class, but has greater functionality.

Site nodes are used to attach items to Segments, such as clothing and jewelry. Site nodes may also be employed to fix a camera so it stays in a certain position relative to the figure.

The Displacer node permits groups of vertices in a Segment to be associated with a higher-level feature of the figure. For example, the location of the nose, eyes, and mouth on a face can be identified with Displacer nodes.

The H-anim specification focuses on humanoids, not on arbitrary articulated figures. Our Limb class may be used to create a variety of figures. Also, the specification says little about animation scripting.

Yuan Cheng has written a robot control simulation environment using Java 3D (http://icmit.mit.edu/robot/simulation.html). The robots are built using a hierarchy of TransformGroups, and there is a rich GUI for controlling them. The code is available for download.

Seungwoo Oh has released a VRML loader which can handle motion data (rotations and translations), and supports geometry skinning (http://vr.kaist.ac.kr/~redmong/research.htm). He has utilized this for clothing human figures, with very convincing results. His site includes Java 3D loaders, and articles explaining the concepts behind his software.

j3d.org has a small demo by Leyland Needham showing how a cylinder can flex like a human arm as it bends in half (http://www.j3d.org/utilities/bones.html). The code gradually updates the cylinder mesh to achieve its effect. This approach can be employed to distort limb shapes as the figure moves, creating the effect of rippling biceps, etc.

Alessandro Borges has created human models using Poser, stored them in the VRML97 format, and loaded them into Java 3D. His demo showing simple movements of the figure is currently unavailable, but was at http://geocities.yahoo.com.br/alessandroborges/ana. Screen shots of the application can still be found there.

There are no Java 3D example using inverse kinematics (as far as I know). The FAQ at j3d.org contains a few links to discussions of how to implement inverse kinematics in procedural languages (http://www.j3d.org/faq/techniques.html#ik).

9. Future Work

As mentioned at the start of this chapter, this work is based on Thana Konglikhit's final year student project. He still has several months left to go, and we are planning the next stage. Currently we hope to concentrate on developing a simple scripting language, that will allow commands such as "walk fast" and "jump high" to be built from the simpler commands described here.

If you have any suggestions about how to develop this work, we would be happy to hear from you. Please send e-mail to `dandrew@ratree.psu.ac.th`. The final outcome of this project will become another chapter sometime in March or April 2004.