3. Transformations: Translations, Rotations, and Scaling

For people new to computer graphics, understanding how 3D (affine) transformations operate can be daunting because the standard approach is to explain things in terms of 4 x 4 matrices applied to homogenous coordinates. Simple3D hides this complexity behind three groups of operations:

1. those that create a **new** TransformGroup: the "set" methods;
2. those that **overwrite** an existing TransformGroup: "To" methods;
3. those that **incrementally change** an existing node: "By" methods;

In particular, the Transform3D object that holds the matrix, nestled inside the TransformGroup, is hidden. Also, rotations are limited to operating around the x-, y-, and z- axes, and there's definitely no mention of axis angles or quaternions.

This may seem limiting, especially when multiple transformations need to be combined, but that difficulty is addressed by utilizing Java 3D's scene graph. As we'll see, it supports an intuitive way to compose simple transformations to form a hierarchy of TransformGroup nodes. This approach will be illustrated with several small examples, and two larger ones – an orbiting 'rocket' and a 'robot arm'. They're both built from primitive shapes (i.e. Cone, Cylinder), but the manipulation of Wavefront OBJ models will be the subject of the next chapter.

1. Primitive Shapes

Although the creation of primitive shapes is straightforward, they have a few fiddly aspects that I'll discuss using StartShapes.java. It contains commented-out code for creating a colored box, sphere, cone, cylinder, and a 3D "R".

```
public class StartShapes
{
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("StartShapes", false, true);

    Box shp = new Box(0.5f, 0.5f, 1.0f, Simple3D.color("red"));
    // Sphere shp = new Sphere(0.5f, Simple3D.color("green"));
    // Cone shp = new Cone(0.5f, 1f, Simple3D.color("blue"));
    // Cylinder shp = new Cylinder(0.5f,1f,Simple3D.color("yellow"));
    // BranchGroup shp = Simple3D.text3D("R", "purple");

    // add to the universe
    j3d.rootBG.addChild( Simple3D.createBG(shp));
  }  // end of main()

}  // end of StartShapes class
```

The different outputs are shown in Figure 1.

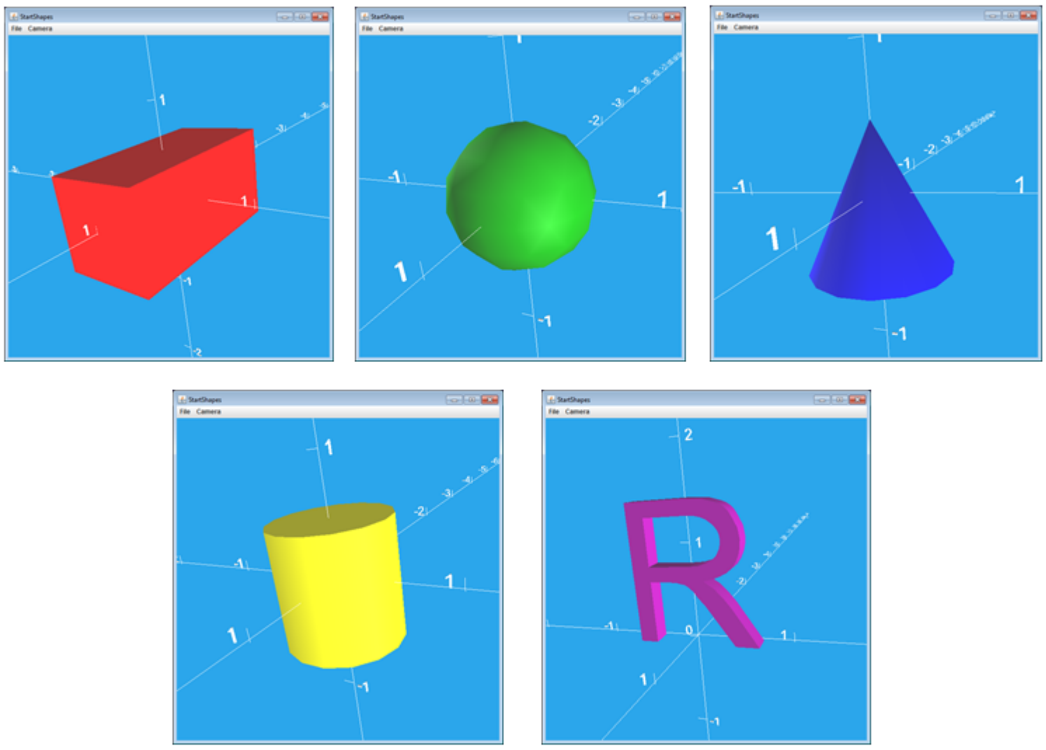


Figure 1. Primitive Shapes and an "R" generated by StartShapes.java.

One tricky thing is that the x-, y-, and z- lengths supplied for the box are half-lengths. In other words, the box's dimensions are actually 1 x 1 x 2.

Another is that the shapes are centered at the origin, which would mean they'd be partially hidden if Simple3D's floor grid was rendered (it extends across the XZ plane at y == 0). This invariably means that the first transformation we need to apply is to move a shape up the y-axis so it's resting on the floor.

A third issue is that these shapes all display varying levels of symmetry, which makes it difficult to be sure how various rotations affect them. For that reason, the examples in the next two sections will use a 3D "R" which conveniently has no symmetries around its x-, y- and z- axes. In addition, most 3D letters are positioned so the back edge of their base is centered on the x-axis at the origin; as a consequence, the entire shape is visible even when the floor grid is drawn.

2. Composing Transforms

RGraph.java renders an "R" after moving it up the y-axis by 0.5 units, rotating it counter-clockwise around the x-axis, and then moving it up another 0.5 units. This could be coded as a single matrix operation, but I'd like to persuade you that it's simpler to define it in small steps, leading to a chain of TransformGroups. Each of these steps is illustrated below.

The first version of the program draws an "R" at its default position:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("RGraph", true, true);

    BranchGroup rShape = Simple3D.text3D("R", "purple");
                    // "R" located at origin
    // add to the universe
    j3d.rootBG.addChild(rShape);
  }  // end of main()
```

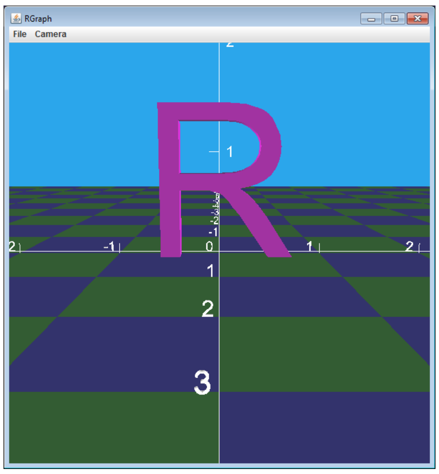


Figure 2. "R" at the origin.

Now a TransformGroup node is added above the rShape BranchGroup to move it up the y-axis:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("RGraph", true, true);

    BranchGroup rShape = Simple3D.text3D("R", "purple");
              // "R" located at origin
    TransformGroup pos1TG = Simple3D.setTranslation(0,0.5,0);
              // up the y axis
    pos1TG.addChild(rShape);   // pos1TG --> "R"

    // add to the universe
    j3d.rootBG.addChild(Simple3D.createBG(pos1TG));
  }  // end of main()
```

Figure 3 shows the new position of "R":

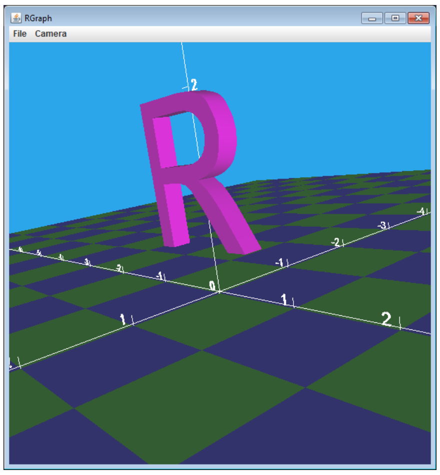


Figure 3. "R" located at (0, 0.5, 0).

Another TransformGroup is added above the first translation node to rotate the shape around the x-axis. Crucially, the rotation is relative to the world's origin, not the letter's 'origin' point (i.e. the center of its base, which is currently at (0, 0.5, 0)).

```
public static void main(String[] args)
{
  Simple3D j3d = new Simple3D("RGraph", true, true);

  BranchGroup rShape = Simple3D.text3D("R", "purple");
               // "R" located at origin
  TransformGroup pos1TG = Simple3D.setTranslation(0,0.5,0);
               // up the y axis
  pos1TG.addChild(rShape);    // pos1TG --> "R"

  TransformGroup rot1TG = Simple3D.rotX(-90);  // ccw around x-axis
  rot1TG.addChild(pos1TG);    // rot1TG --> pos1TG --> "R"

  // add to the universe
  j3d.rootBG.addChild(Simple3D.createBG(rot1TG));
}  // end of main()
```

Figure 4 shows that the letter is now resting on its back on the XZ plane.

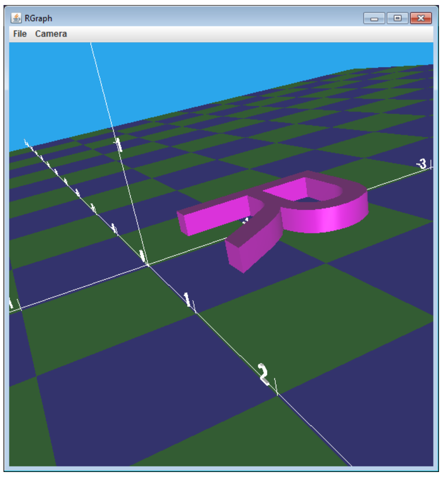


Figure 4. "R" on its back.

A third TransformGroup node is now added, above the other two in the scene graph, to move the shape up the y-axis again. As with the rotation, the translation is relative to the world's origin, not the letter's origin (which is now at (0, 0, -0.5))

```
public static void main(String[] args)
{
  Simple3D j3d = new Simple3D("RGraph", true, true);

  BranchGroup rShape = Simple3D.text3D("R", "purple");
              // "R" located at origin
  TransformGroup pos1TG = Simple3D.setTranslation(0,0.5,0);
              // up the y axis
  pos1TG.addChild(rShape);   // pos1TG --> "R"

  TransformGroup rot1TG = Simple3D.rotX(-90);  // ccw around x-axis
  rot1TG.addChild(pos1TG);   // rot1TG --> pos1TG --> "R"

  TransformGroup pos2TG = Simple3D.setTranslation(0,0.5,0);
              // up the y axis
  pos2TG.addChild(rot1TG);  // pos2TG --> rot1TG --> pos1TG --> "R"

  // add to the universe
  j3d.rootBG.addChild(Simple3D.createBG(pos2TG));
}  // end of main()
```

Figure 5 shows that the letter has risen up the y-axis.

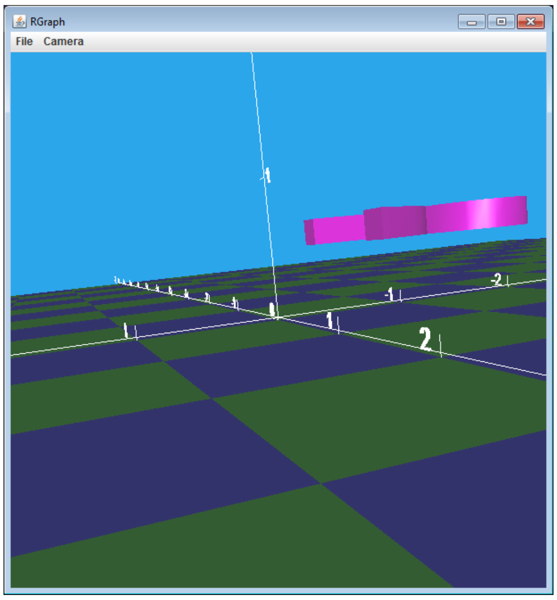


Figure 5. The floating "R".

The resulting chain of TransformGroups is nicely summarized by the comment in the code:

```
// pos2TG --> rot1TG ---> pos1TG --> "R"
```

The key to understanding this chain is to read it from right-to-left (or equivalently up from the leaf node towards the root of the scene graph). Each transformation is applied in terms of world coordinates, which means relative to (0,0,0). Each TransformGroup is limited to one transformation, and the necessary complexity is built up by the addition of nodes.

This gradual application of transformations (as shown in Figures 2-5) is a good way to write Java 3D code. It forces the programmer to divide the transformation into parts which can be implemented and tested incrementally.

3. Composition Rules

This chaining together of TransformGroups has a few 'rules' that should be borne in mind when a chain is being created:

1. A node should perform only one transformation (e.g. a translation, rotation, or scaling).
2. The ordering of the nodes matters: e.g. a rotation and then a translation is not the same as the translation then the rotation.
3. Rotations around different axes should be considered to be different transformations, and so be split across nodes.
4. If a transformation is going to be modified at run time, then you should probably use one of Simple3D's "By" methods to do it. (This rule is the focus of section 5.)

I'll illustrate these rules by looking at various functions in RTrans.java:

```
private static BranchGroup yellowR, redR, blueR;

public static void main(String[] args)
{
  Simple3D j3d = new Simple3D("Text 3D", true, true);

  yellowR = Simple3D.text3D("R", "yellow");
  redR = Simple3D.text3D("R", "red");
  blueR = Simple3D.text3D("R", "blue");

  BranchGroup sceneBG = posRot();
  // BranchGroup sceneBG = posScale();
  // BranchGroup sceneBG = rotScale();

  // BranchGroup sceneBG = twoSameAxisRots();
  // BranchGroup sceneBG = twoDiffAxesRots();

  // BranchGroup sceneBG = twoPosTG();
  // BranchGroup sceneBG = twoRotTG();

  // BranchGroup sceneBG = posRot1TG();

  // BranchGroup sceneBG = axisAngles();

  // add everything to the universe
  j3d.rootBG.addChild(sceneBG);
}  // end of main()
```

The program begins by creating three "R"s, each colored differently and assigned to global variables. The functions apply various transformations, building a scene graph beneath sceneBG which is rendered at the end.

3.1. Ordering Matters

The posRot() function applies the same translation and rotation to the red "R" and the blue "R", but in opposite order. The result, shown in Figure 6, shows that the ordering of these operations matters.

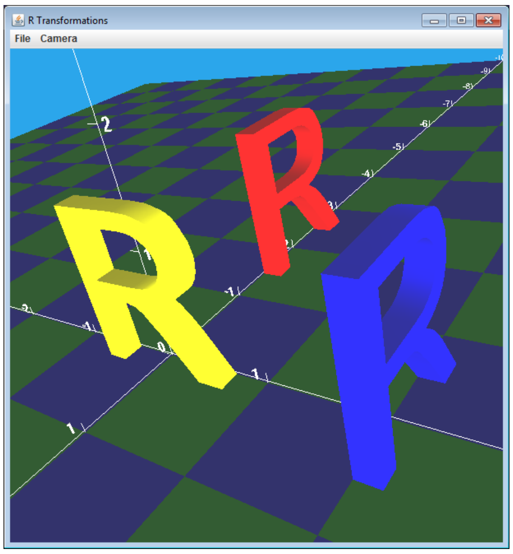


Figure 6. Translation/Rotation in Different Orders.

posRot() is defined as:

```
private static BranchGroup posRot()
// translations and rotations cannot be swapped usually
{
  // pos then rot: rot1 --> pos1 --> red R
  TransformGroup pos1 = Simple3D.setTranslation(2, 0, 0);
  pos1.addChild(redR);
  TransformGroup rot1 = Simple3D.rotY(90);
  rot1.addChild(pos1);

  // rot then pos: pos2 --> rot2 --> blue R
  TransformGroup rot2 = Simple3D.rotY(90);
  rot2.addChild(blueR);
  TransformGroup pos2 = Simple3D.setTranslation(2, 0, 0);
  pos2.addChild(rot2);

  // build top-level scene
  BranchGroup sceneBG = new BranchGroup();
  sceneBG.addChild(yellowR);
  sceneBG.addChild(rot1);   // red
  sceneBG.addChild(pos2);   // blue

  return sceneBG;
}  // end of posRot()
```

The yellow "R" is left unchanged; the red "R" is translated to (2,0,0) and then rotated 90 degrees counter-clockwise around the y-axis, and the blue "R" is rotated first, and then translated.

The secret to understanding the differences between the red and blue "R"s is to recall that the TransformGroups are performed right-to-left as per the code comments, and that the operations are always relative to the world coordinates. In particular, a y-axis rotation is relative to the line y == 0 running through the origin.

In a classroom situation, it helps to have a cardboard letter "R" which can be used to animate the operations.

RTrans.java also includes functions called posScale() and rotScale() which illustrate a similar point about ordering a translation and scaling, and a rotation and scaling.

3.2. Rotations around Different Axes are Different

The importance of ordering translation, rotation, and scaling operations extends to different kinds of axis rotations. For example, a rotation about the x-axis followed by a rotation around the z-axis is not generally the same in the opposite order. This is illustrated by twoDiffAxesRots() in RTrans.java:

```
  private static BranchGroup twoDiffAxesRots()
  // rotations around different axes cannot usually be swapped
  {
    // rotations: rotZ --> rotX --> red R
    TransformGroup rot1 = Simple3D.rotX(90);
    rot1.addChild(redR);
    TransformGroup rot2 = Simple3D.rotZ(90);
    rot2.addChild(rot1);
/*
  // or
    TransformGroup rot2 = Simple3D.setRotations(90, 0, 90);
           // x-, y-, then z- rotation in world coords;
           // best to avoid this, at least when starting out
    rot2.addChild(redR);
*/

    // opp. rotations: rotX --> rotZ --> blue R
    TransformGroup rot3 = Simple3D.rotZ(90);
    rot3.addChild(blueR);
    TransformGroup rot4 = Simple3D.rotX(90);
    rot4.addChild(rot3);

    // build top-level scene
    BranchGroup sceneBG = new BranchGroup();
    sceneBG.addChild(yellowR);
    sceneBG.addChild(rot2);   // red rotations
    sceneBG.addChild(rot4);   // blue rotations

    return sceneBG;
  }  // end of twoDiffAxesRots()
```

The difference between the orders is apparent in Figure 7. Note, that I've switched off the floor grid so that the blue "R" can be seen.

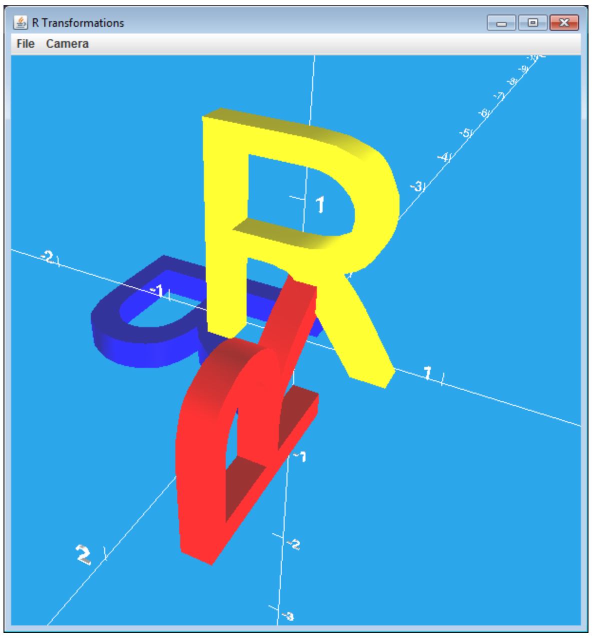


Figure 7. Rotations in Different Orders.

To understand why the blue and red "R"s are where they are, it helps to keep the right hand rule for rotations in mind, which is summarized by Figure 8.

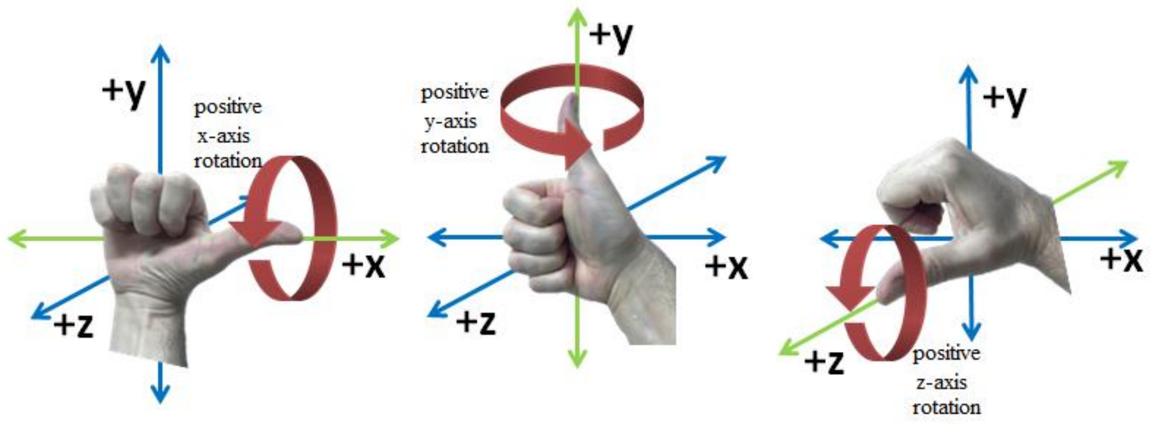


Figure 8. The Right Hand Rule for Rotations.

A positive rotation follows the direction of the fingers in Figure 8.

The blue "R" is rotated around the z-axis, and then the x-axis, and so ends up face down just below the XZ plane (recall that a letter starts with the back of its base resting on the x-axis).

The red "R" is rotated around the x-axis, and then the z-axis, and so ends up facing to the right, with half of the letter below the XZ plane.

twoDiffAxesRots() contains a few lines of commented-out code which illustrate the use of Simple3D.setRotation**s**() (note the "s"), which can apply an x-, y-, and then an z- axis rotation to a single TransformGroup. This operation should be avoided in most cases since the ordering of the rotations is easy to forget, and the resulting TransformGroup is harder to modify. In twoDiffAxesRots(), setRotations() achieves the same effect as the two rotations applied to the red "R".

3.3. Use "By" methods to Incrementally Change a TransformGroup

Especially when an object is being animated, there's a need to incrementally change a TransformGroup value. If the 'rules' listed at the start of this section are followed, then each TransformGroup will manage a single translation, rotation, or scaling, and so can be modified using Simple3D's "By" methods:

```
public static void moveBy(TransformGroup tg,
                               double dx, double dy, double dz);
private static void moveBy(TransformGroup tg, Vector3d offset);

public static void rotXBy(TransformGroup tg, double degrees);
public static void rotYBy(TransformGroup tg, double degrees);
public static void rotZBy(TransformGroup tg, double degrees);
```

There's no scaleBy() since it seems easiest just to assign a new scale factor to a node by calling Simple3D.scale().

The use of moveBy() is illustrated in twoPosTG() in RTrans.java, while rotYBy() is utilized in twoRotTG().

twoPosTG() translates the red "R" and blue "R" to almost the same position by applying Simple3D.setTranslation() and Simple3D.moveBy() operations to the pos1 and pos2 TransformGroups. The end positions are slightly different so that the two shapes don't occupy the same space.

```
private static BranchGroup twoPosTG()
// two translations applied to one TG; uses moveBy()
{
  // pos1 (1,0,3) + (1,0,0) --> red R
  TransformGroup pos1 = Simple3D.setTranslation(1,0,3);
  pos1.addChild(redR);
  Simple3D.moveBy(pos1, 1,0,0);

  // pos2 (1,0.5,3) + (1,0,3) --> blue R
  TransformGroup pos2 = Simple3D.setTranslation(1,0,0);
  pos2.addChild(blueR);
  Simple3D.moveBy(pos2, 1,0.5,3);   // up a bit, so visible

  // build top-level scene
  BranchGroup sceneBG = new BranchGroup();
  sceneBG.addChild(yellowR);   // yellow
  sceneBG.addChild(pos1);   // red
  sceneBG.addChild(pos2);   // blue
```

```
    return sceneBG;
  }  // end of twoPosTG()
```

The rendering of the "R"s is shown in Figure 9.

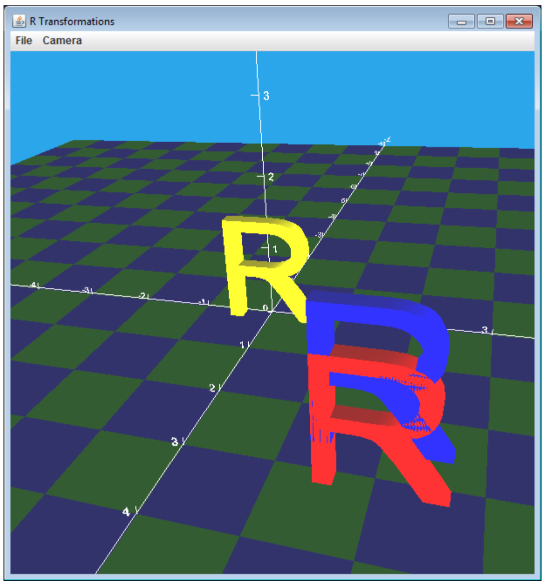


Figure 9. Using Simple3D.moveBy().

The important thing to remember is that moveBy() should only be applied to a translation-based TransformGroup, and will then have the effect of 'adding' the moveBy() coordinate to the existing coordinate in the node. If it's applied to a node containing a rotation or a scaling it will most likely 'work', but the user will need to think in terms of matrix multiplication to confirm the result. Also, if that node is later modified by a rotation "By" method, the translation component will also be affected, which is probably not the intention.

twoRotTG() illustrates incremental rotation, in this case around the y-axis. The red and blue "R"s are rotated by Simple3D.rotY() and Simple3D.rotYBy() applied to the rot1 and rot2 TransformGroups. They end up in with almost the same orientation, but the angles are deliberately a little different so the two shapes don't end up in the same space.

```
  private static BranchGroup twoRotTG()
  // two rotations (around the same axis) applied to one TG
  {
    // rot1 40 + 50 --> red R
    TransformGroup rot1 = Simple3D.rotY(40);
    rot1.addChild(redR);
    Simple3D.rotYBy(rot1, 50);

    // rot2 50 + 50 --> blue R
    TransformGroup rot2 = Simple3D.rotY(50);
    rot2.addChild(blueR);
    Simple3D.rotYBy(rot2, 50);    // rotate a bit more, so visible
    // build top-level scene
    BranchGroup sceneBG = new BranchGroup();
    sceneBG.addChild(yellowR);    // yellow
    sceneBG.addChild(rot1);    // red
    sceneBG.addChild(rot2);    // blue

    return sceneBG;
  }  // end of twoRotTG()
```

Figure 10 shows the output.

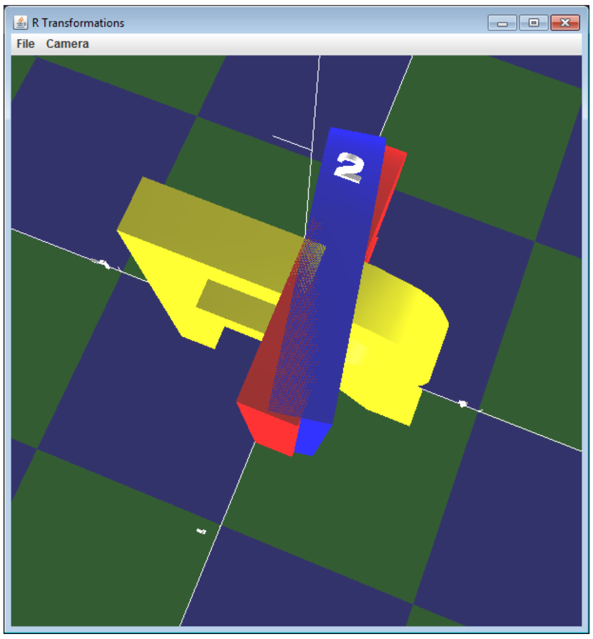


Figure 10. Using Simple3D.rotYBy().

As with moveBy(), this operation may 'work' when applied to TransformGroups containing translations, scalings, or rotations around other axes, but will be much harder to understand.

3.4. Breaking the Rules: Rotating and Translating one TransformGroup

There may come a time when the TransformGroup rules that I've outlined are too limiting, although I think this unfortunate situation won't arise in the simple examples that I'll be developing in later chapters.

posRot1TG() implements a situation where a rotation and then a translation increment is applied to the tg1 TransformGroup for the red "R". Meanwhile, the blue "R" is transformed by the tg2 node with the same operations in the opposite order:

```
  private static BranchGroup posRot1TG()
  /* translation and rotation applied to one TG;
     Avoid this approach since 'and' is really
     4x4 matrix multiplication. Use a scene graph
     hierarchy instead.
```

```
   */
   {
     // tg1 45-rotY '+' (0,0,3) --> red R
     TransformGroup tg1 = Simple3D.rotY(45);
     Simple3D.moveBy(tg1, 0, 0, 3);
     tg1.addChild(redR);
     System.out.println("tg1 45-rotY '+' (0,0,3) --> red R");
     Simple3D.printMatrix(tg1);

     // tg2 (0,0,3) '+' 45-rotY --> blue R
     TransformGroup tg2 = Simple3D.setTranslation(0,0,3);
     Simple3D.rotsBy(tg2, 0, 45, 0);
/*
     // or use setTransforms() to combine into one op.
     TransformGroup tg2 = Simple3D.setTransforms(
                                 0,0,3,      // (x,y,z) translation
                                 0,45,0,     // rotations
                                 1);         // scale
*/
     tg2.addChild(blueR);
     System.out.println("tg2 (0,0,3) '+' 45-rotY --> blue R");
     Simple3D.printMatrix(tg2);

     // build top-level scene
     BranchGroup sceneBG = new BranchGroup();
     sceneBG.addChild(yellowR);    // yellow
     sceneBG.addChild(tg1);    // red
     sceneBG.addChild(tg2);    // blue

     return sceneBG;
   }  // end of posRot1TG()
```

Figure 11 shows that the two "R"s end up in different positions.

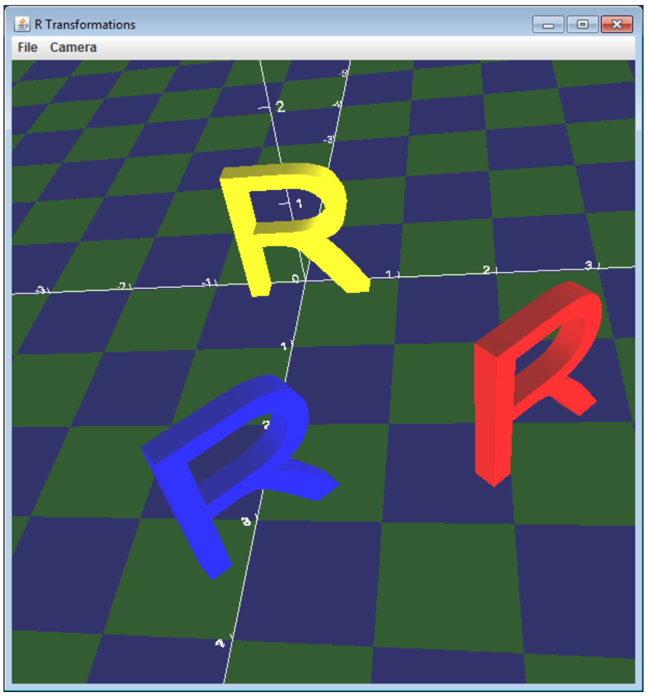


Figure 11. Combining Transformations in one TransformGroup.

Although the red and blue "R"s are rotated by what appears to be a similar amount, they are positioned differently on the XZ plane. I would argue that the only way to understand why this is so is to peer inside the TransformGroup node, into its Transform3D object, and examine how its 4x4 matrix is modified.

This nasty task is made somewhat easier by calling Simple3D.printMatrix(), which reports:

```
tg1 45-rotY '+' (0,0,3) --> red R
|  0.71  0.00  0.71  2.12  |
|  0.00  1.00  0.00  0.00  |
| -0.71  0.00  0.71  2.12  |
|  0.00  0.00  0.00  1.00  |

tg2 (0,0,3) '+' 45-rotY --> blue R
|  0.71  0.00  0.71  0.00  |
|  0.00  1.00  0.00  0.00  |
| -0.71  0.00  0.71  3.00  |
|  0.00  0.00  0.00  1.00  |
```

This will mean nothing to a novice 3D programmer unless they understand how rotations, translations, and scalings are represented inside the matrix. The gory details are summarized by Figure 12.

$$\text{trans(x,y,z)} = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{rotX}(\Theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{scale(x,y,z)} = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{rotY}(\Theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{rotZ}(\Theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 12. Transforms in a 4x4 Matrix.

The unfortunate programmer will also have to know how these transformations are combined via matrix multiplication, including the ordering of the matrices (since matrix multiplication isn't commutative).

For the red "R", the tg1 TransformGroup is first assigned a 45 degree y-axis rotation (a rotY() matrix in Figure 12) and then a translation matrix is multiplied to it as its *right-hand* argument. This is represented by Figure 13.

$$\overset{\text{t3d}}{\begin{bmatrix} r & 0 & r & 0 \\ 0 & 1 & 0 & 0 \\ -r & 0 & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \text{X} \overset{\text{moveT3d}}{\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}} = \begin{bmatrix} r & 0 & r & 3r \\ 0 & 1 & 0 & 0 \\ -r & 0 & r & 3r \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad r = \sqrt{2}/_2$$

Figure 13. The red "R": A 45 degree rotation which is then translated.

The r represents the cosine and sine values for 45 degrees, and the red numbers are the resulting position of the "R". 3r is approximately 2.12, and the right hand side of Figure 13 does indeed correspond to the first matrix printed by Simple3D.printMatrix().

The t3d and moveT3d labels in Figure 13 refer to the code for Simple3D.moveBy():

```
private static void moveBy(TransformGroup tg, Vector3d offset)
// Move the TransformGroup by the offset amount
{
  Transform3D t3d = new Transform3D();
  tg.getTransform(t3d);

  Transform3D moveT3d = new Transform3D();
  moveT3d.setTranslation(offset); // only translate
  t3d.mul(moveT3d);    // 'add' translation to existing one
  tg.setTransform(t3d);
}  // end of moveBy()
```

t3d is the Transform3D object inside tg1 which is multiplied by the translation encoded in moveT3d.

For the blue "R", the tg2 TransformGroup is first translated to (0,0,3) and then a 45 degree y-axis rotation is multiplied to it on the right. The operation is represented by Figure 14.

$$\overset{\text{t3d}}{\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \text{X} \overset{\text{rotT3d}}{\begin{bmatrix} r & 0 & r & 0 \\ 0 & 1 & 0 & 0 \\ -r & 0 & r & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} = \begin{bmatrix} r & 0 & r & 0 \\ 0 & 1 & 0 & 0 \\ -r & 0 & r & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 14. The blue "R": A translation is then rotated by 45 degrees around the y-axis.

The t3d and rotT3d labels in Figure 14 refer to the code for Simple3D.rotYBy():

```
public static void rotYBy(TransformGroup tg, double degrees)
// rotate the existing TransformGroup by degrees around the y-axis
{
  Transform3D t3d = new Transform3D();
  tg.getTransform(t3d);

  Transform3D rotT3d = new Transform3D();
  rotT3d.rotY( Math.toRadians(degrees) );
  t3d.mul(rotT3d);

  tg.setTransform(t3d);
}  // end of rotYBy()
```

The purpose of this section is to illustrate that combining transformations in a single TransformGroup requires a much lower-level understanding of the way that 3D transformations are implemented.

Nevertheless, sometimes it is necessary to combine transformations in this way, and that's supported by Simple3D.setTransforms(), an example of which is commented out in posRot1TG():

```
/*
    // or use setTransforms() to combine into one op.
    TransformGroup tg2 = Simple3D.setTransforms(
                                0,0,3,     // (x,y,z) translation
                                0,45,0,    // rotations
                                1);        // scale
*/
```

The programmer utilizing setTransforms() should remember that the translation is applied first, followed by the rotations. If there are several rotations then they're performed in the order x-, y-, then z-. Also, the scaling factor is applied only to the rotation component of the matrix, which means that the shape changes size but its position is unaffected.

This behavior matches the actions applied to the blue "R", and so setTransforms() could be substituted for its translation 'plus' rotation.

4. An Orbiting Rocket

A drawback of the preceding examples is that the transformations have only been applied to a single shape (the letter "R"). This section shows how to position the components of a more complex shape (in this case, a rocket), and collect them together as a single model by using a BranchGroup. This node is packaged inside a Rocket class, from which Rocket objects can be instantiated.

An object's top-level BranchGroup node can be transformed in the same way as any simple shape, and this technique is used to rotate and position the rocket so that Simple3D.orbit() can be employed to have it circle the y-axis. Figure 15 shows the rocket in orbit.

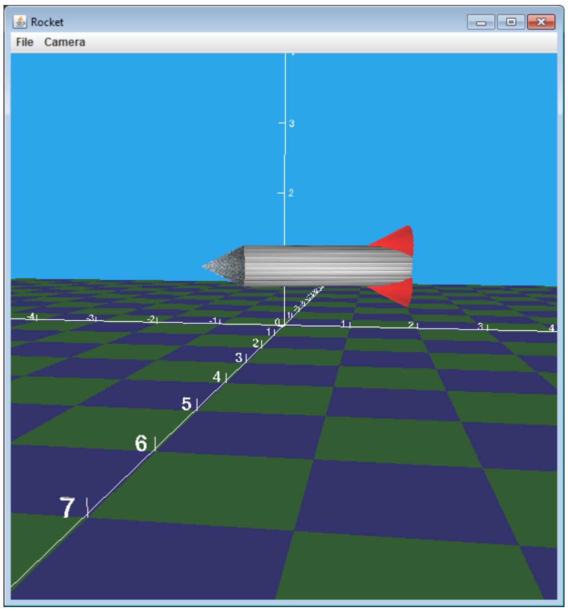


Figure 15. The Orbiting Rocket.

4.1. Sketching the Rocket

The first stage in rocket design is to draw a sketch which breaks it down into simpler, built-in primitive shapes, namely boxes, cones, cylinders, and spheres. In this instance, we'll utilize three cones and a cylinder, as shown in Figure 16.

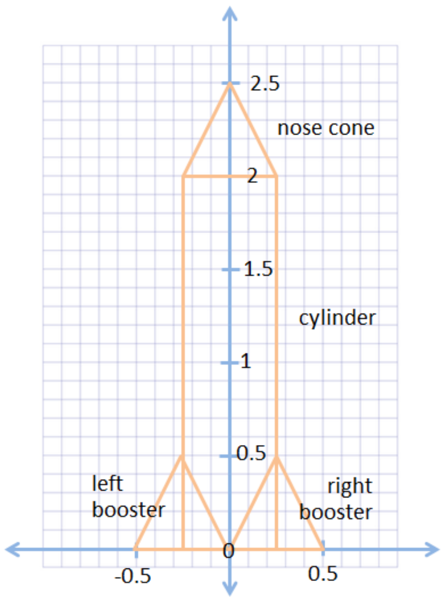


Figure 16. The Rocket at the Origin.

Aside from Figure 16 listing the various dimensions of the shapes, it also specifies that the rocket is initially positioned with its base centered at the origin.

4.2. Defining the Rocket's Scene Graph

The scene graph for the rocket must create the primitive shapes and move/rotate them into the positions specified by the sketch in Figure 16.

Neither the cones or cylinder need to be rotated, but they must be repositioned. A scene graph drawing of what's required:

```
rocketBG --> pos1TG --> cylinder
    |
    ----> pos2TG --> nose cone
    |
    ----> pos3TG ---> left booster
    |
    ----> pos4TG --> right booster
```

"TG" is short for TransformGroup, and "BG" stands for BranchGroup.

As mentioned above, the BranchGroup will be the node visible to programs that use rockets. Any TransformGroups to the *left* of rocketBG (i.e. higher in the scene graph) will affect all the shapes managed by that group.

Based on the dimensions in Figure 16 and the scene graph drawing, the constructor for the Rocket class can be encoded. The completed scene graph is assigned to a rocketBG global variable:

```
public class Rocket
{
  private BranchGroup rocketBG = null;


  public Rocket()
  {
    // build the shapes
    Cone noseCone = Simple3D.texCone(0.25f, 0.5f, "images/grid.jpg");

    Cylinder cyl = Simple3D.texCylinder(0.25f, 2f,
                                          "images/steel1.jpg");
    Cone leftBooster = new Cone(0.25f, 0.5f,
                            Simple3D.color("red"));
    Cone rightBooster = new Cone(0.25f, 0.5f,
                            Simple3D.color("red"));

    // move the cylinder up the y-axis to rest on floor
    TransformGroup pos1TG = Simple3D.setTranslation(0, 1, 0);
    pos1TG.addChild(cyl);

    // position the top cone up y-axis to rest on cylinder top
    TransformGroup pos2TG = Simple3D.setTranslation(0, 2.25, 0);
    pos2TG.addChild(noseCone);
    // move the left booster left along x-axis and up
    TransformGroup pos3TG = Simple3D.setTranslation(-0.25, 0.24, 0);
    pos3TG.addChild(leftBooster);

    // move the right booster right along x-axis and up
    TransformGroup pos4TG = Simple3D.setTranslation(0.25, 0.24, 0);
    pos4TG.addChild(rightBooster);

    // group the shapes under a BranchGroup
    rocketBG = Simple3D.createBG();
    rocketBG.addChild(pos1TG);  // cylinder
    rocketBG.addChild(pos2TG);  // top cone
    rocketBG.addChild(pos3TG);  // left booster
    rocketBG.addChild(pos4TG);  // right booster
  }  // end of Rocket()


  public BranchGroup getBG()
  {  return rocketBG;  }

} // end of Rocket class
```

The simplicity of the rocket code means that it could have been implemented as a single function, returning the rocketBG BranchGroup, but it's generally more useful to utilize a class so that additional state (e.g. the speed and current position of the rocket) can be stored along with the visualization.

A rocket can be displayed like so:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("Rocket", 0, 1, 10, true, true);

    Rocket r = new Rocket();
    j3d.rootBG.addChild(r.getBG());
  }  // end of main()
```

4.3. Positioning the Rocket

The rocketBG BranchGroup lets the rocket be manipulated in the same way as primitive shapes (e.g. as in section 2). A key point to recall is which part of the rocket is located at the origin, since any translations, rotations, or scalings will be relative to that point. As the sketch in Figure 16 indicates, the rocket's base is centered at the origin.

The positioning of the rocket is specified in terms of another scene graph:

```
posTG --> rotTG --> rocketBG
```

Note that rocketBG hides the complexity of the underlying shape. The rocket is rotated clockwise around the z-axis to point right by rotTG, and then moved up and backwards into the scene via posTG:

```
    Simple3D j3d = new Simple3D("Rocket", 0, 1, 10, true, true);

    Rocket r = new Rocket();

    // posTG --> rotTG --> rocket
    TransformGroup rotTG = Simple3D.rotZ(-90);  // cw around z-axis
    rotTG.addChild(r.getBG());

    // translate so centered at (0,1,-2); up and into the scene
    TransformGroup posTG = Simple3D.setTranslation(-1.25,1,-2);
    posTG.addChild(rotTG);

    j3d.rootBG.addChild( Simple3D.createBG(posTG));
```

The last step is to employ Simple3D.orbit() to make the rocket orbit clockwise around the y-axis. This creates another TransformGroup which attaches a rotator interpolator to the posTG node:

```
 flyingRocket --> orbitTG --> posTG --> rotTG --> rocketBG
                     |         ||
                     ----> rotator
```

This extends the code:

```
    Simple3D j3d = new Simple3D("Rocket", 0, 1, 10, true, true);

    Rocket r = new Rocket();
    TransformGroup rotTG = Simple3D.rotZ(-90);  // cw around z-axis
    rotTG.addChild(r.getBG());

    // translate so centered at (0,1,-2); up and into the scene
    TransformGroup posTG = Simple3D.setTranslation(-1.25,1,-2);
    posTG.addChild(rotTG);

    // make the rocket orbit clockwise around y-axis
    TransformGroup orbitTG = Simple3D.orbit(posTG, 4000,
                                           -Simple3D.Y_AXIS);
    BranchGroup flyingRocket = Simple3D.createBG(orbitTG);

    j3d.rootBG.addChild(flyingRocket);
```

Perhaps the most important thing to take away from this example is the stepwise development of the code. First the rocket was created at the origin, then moved to its orbital starting position, and finally made to rotate.

5. A Controllable Jointed Arm

The jointed arm developed in this section is shown in Figure 17. The user can rotate its joints located in the red base and the ends of the green and blue cylinders by typing instructions at the command line.

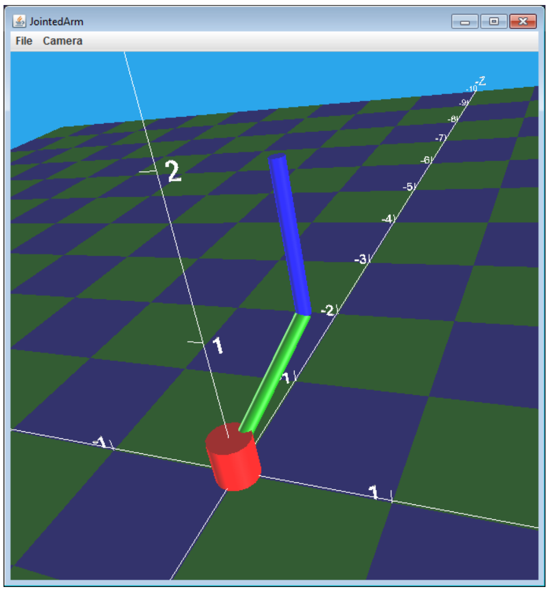


Figure 17. The Jointed Arm.

The arm is implemented as a class called JointedArm, and a separate program, UseArm.java, contains the code for reading and processing user commands. The position of the arm in Figure 17 was achieved by the user typing:

```
>> l 45
>> u -45
>> b 30
>> b 30
```

"l' refers to the thin blue cylinder (the 'lower' part of the arm), "u" is the thin green cylinder (the 'upper' part of the arm), and "b" is the red base. The lower and upper arms can rotate around the z-axis, while the base turns around the y-axis. The rotation values are supplied in degrees.

5.1. Sketching the Jointed Arm

The three cylinders are created with their centers located at the origin, as depicted in Figure 18. Of course, the shapes will overlap, but I've drawn them offset to make things a little clearer.

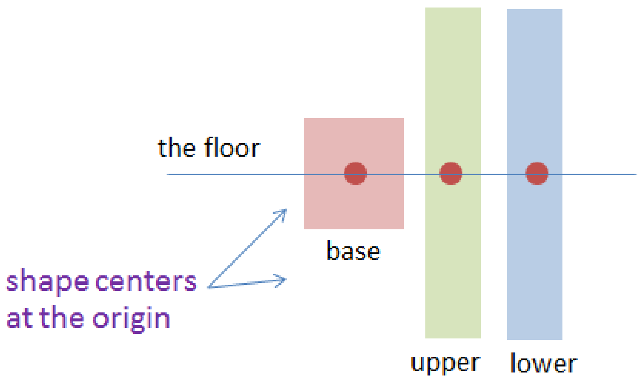


Figure 18. The Three Cylinders at the Origin.

These shapes need to be repositioned so they can rotate around their joints – drawn as circles in Figure 19, and called rotBase, rotUpper, and rotLower. The rotBase and rotUpper nodes are actually at the same location, but rotBase handles y-axis rotations, while rotUpper rotates only around the z-axis.

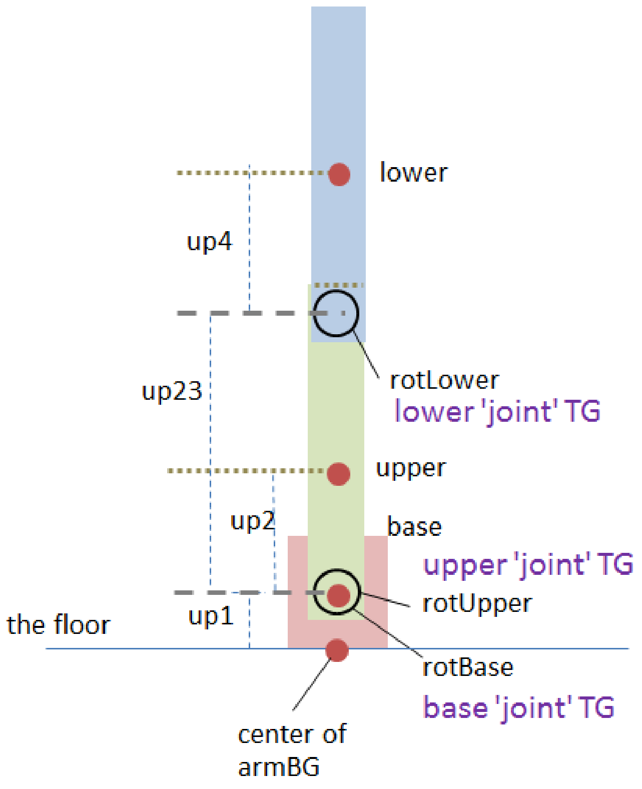


Figure 19. The Three Cylinders Positioned to Form the Arm.

It's best to design the scene graph for such a complicated model in stages. The first thing to remember is that the shapes (base, upper, and lower) must be leaf nodes, while the top-level node is usually a BranchGroup (I'll call it armBG).

The three joints (rotBase, rotUpper, and rotLower) will be implemented as TransformGroups to handle the rotations. Crucially, rotUpper must affect both the upper and lower cylinders, which implies that they should be grouped in a BranchGroup below rotUpper. A similar argument applies to rotBase – it should affect all three cylinders: base, upper and lower, and so they must also be grouped.

These insights allow a preliminary scene graph to be designed, which lacks positional nodes, as shown in Figure 20.

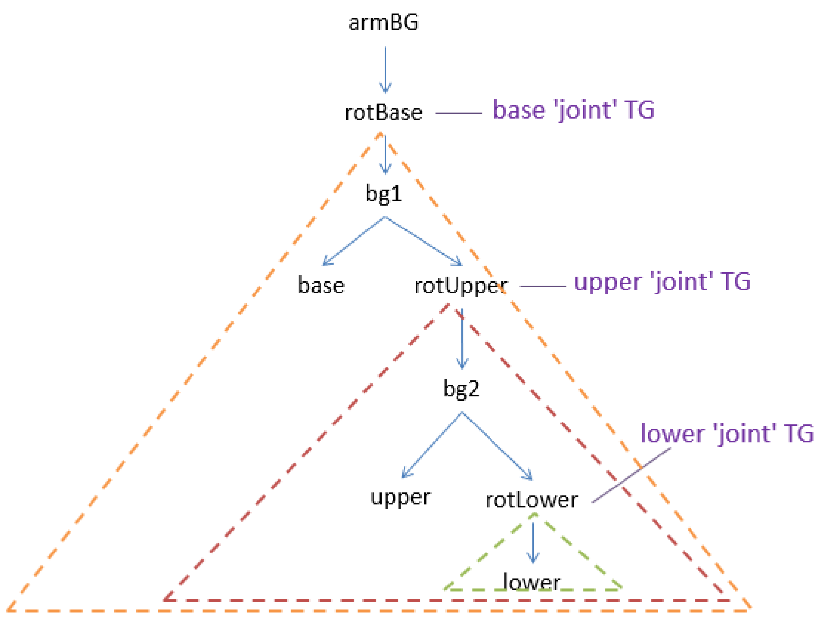


Figure 20. A Partial Scene Graph for the Jointed Arm.

The dashed triangles in Figure 20 illustrate the influence of rotating a joint on the nodes further down in the graph.

We can now turn to the distances shown in Figure 19, and decide *where* they should be added to the graph in Figure 20.

up2 and up4 are vertical offsets of the center of the upper and lower cylinders from their joints (rotUpper and rotLower), and so can be inserted just above the upper and lower nodes in the graph.

up23 represents the distance between the rotUpper and rotLower joints, and so should go somewhere in the graph between those two nodes.

up1 is the most complicated node to insert since it's used to lift the base up the y-axis *and* also the rotBase joint. This suggests that it should be inserted above both of those nodes in the graph.

The end result is shown in Figure 21.

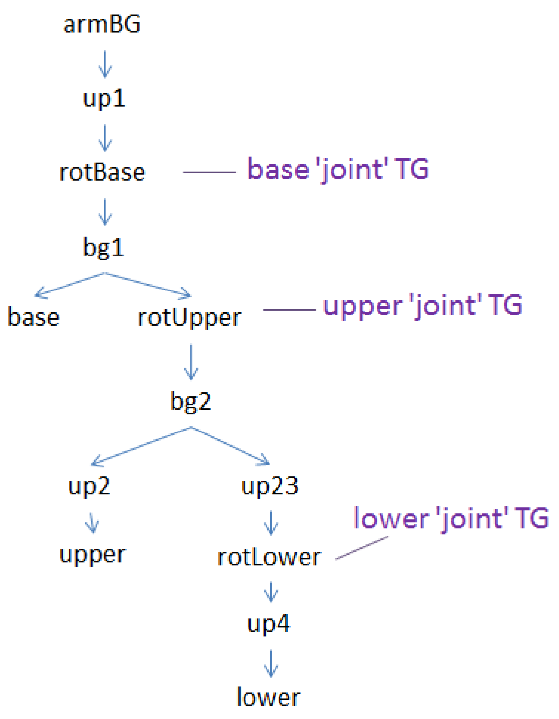


Figure 21. The Completed Scene Graph.

5.2. Defining the Arm's Scene Graph

Devising the scene graph in Figure 20 is no easy matter, but translating it into Java 3D code is relatively automatic, using the same techniques as employed for the rocket in the previous section.

The graph is built in the constructor of a JointedArm class which ends with the graph's assignment to a global variable called armBG which is accessible via a public getBG() method.

```
public class JointedArm
{
  private BranchGroup armBG;
  private TransformGroup rotLower, rotUpper, rotBase;
              // the three arm 'joints'


  public JointedArm()
  {
    // create the shapes
    Cylinder base = new Cylinder(0.15f, 0.3f, Simple3D.color("red"));
    Cylinder upper = new Cylinder(0.05f,1f, Simple3D.color("green"));
    Cylinder lower = new Cylinder(0.05f,1f, Simple3D.color("blue"));

    /*
         bg2 --> up23 --> rotLower --> up4 --> lower cyl
          |
          -----> up2 --> upper cyl
    */
    TransformGroup up4 = Simple3D.setTranslation(0,0.5,0);
    up4.addChild(lower);

    rotLower = Simple3D.createTG();
    rotLower.addChild(up4);
    new TGViewer("rotLower", rotLower, true);  // show world coords

    TransformGroup up23 = Simple3D.setTranslation(0,1,0);
    up23.addChild(rotLower);

    TransformGroup up2 = Simple3D.setTranslation(0,0.5,0);
    up2.addChild(upper);

    BranchGroup bg2 = new BranchGroup();
    bg2.addChild(up23);
    bg2.addChild(up2);

    /*
        up1 --> rotBase --> bg1 --> rotUpper --> bg2
                             |
                             -----> base cyl
    */
    rotUpper = Simple3D.createTG();
    rotUpper.addChild(bg2);
    new TGViewer("rotUpper", rotUpper);

    BranchGroup bg1 = new BranchGroup();
    bg1.addChild(base);
    bg1.addChild(rotUpper);

    rotBase = Simple3D.createTG();
    new TGViewer("rotBase", rotBase);
    rotBase.addChild(bg1);

    TransformGroup up1 = Simple3D.setTranslation(0,0.15,0);
    up1.addChild(rotBase);

    /* complete the arm:
        armBG --> up1
    */
    armBG = Simple3D.createBG(up1);

    // a hand model (not currently used)
    BranchGroup modelBG =
              Simple3D.loadModel("models/hand/hand.obj");
    TransformGroup handTG = Simple3D.setTranslation(2,0.1,0);
    handTG.addChild(modelBG);
    // armBG.addChild(handTG);
  }  // end of JointedArm()


  public BranchGroup getBG()
  {  return armBG;  }
  // more methods, explained below...

}  // end of JointedArm class
```

I didn't code JointedArm() in one attempt, instead breaking it into several stages: initially restricting myself to positioning the base and upper shapes, and only later adding the lower cylinder and its joint. The test code I utilized during this was something like:

```
  public static void main(String[] args)
  {
    Simple3D j3d = new Simple3D("JointedArm", 0, 1, 7, true, true);

    JointedArm arm = new JointedArm();
    j3d.rootBG.addChild(arm.getBG());
  }  // end of main()
```

JointedArm also has three global TransformGroup variables – rotBase, rotUpper, and rotLower, which are manipulated by three public methods:

```
 // methods in JointedArm

  public void rotateBase(double degrees)
  // add degrees to the base's rotation around the y-axis
  {  Simple3D.rotYBy(rotBase, degrees);  }


  public void rotateUpper(double degrees)
  /* add degrees to the upper's rotation around its z-axis
     at the point where it intersects the base
  */
  {  Simple3D.rotZBy(rotUpper, degrees);  }


  public void rotateLower(double degrees)
  /* add degrees to the lower's rotation around its z-axis
     at the point where it intersects the upper cylinder
  */
  {  Simple3D.rotZBy(rotLower, degrees); }
```

These use Simple3D "By" methods to incrementally rotate the arm's joints.

5.3. Controlling the Arm

The main() method from above is extended in the UseArm.java program to process 'l', 'u', and 'b' commands entered by the user. A typical session with the arm would be:

```
> run UseArm
Executing UseArm with Simple3D.jar
Loading JOGL, Java 3D, Simple3D...
Named objects in model:
  default: Shape3D; no. geometries: 1
Possible commands:  b | u | l  <angle>
  b == base (the red cylinder), which rotates around the y-axis
  u == upper (the thin green cylinder), which rotates around the z-
axis
  l == lower (the thin blue cylinder), which rotates around the z-
axis
Enter command (e.g. quit)
>> l 45
>> u -45
>> b 90
>> b 90
>> q
Simple3D closing down...
Finished.
>
```

The user's input is highlighted in bold. Figure 22 shows a sequence of screenshots as the four joint commands are processed.

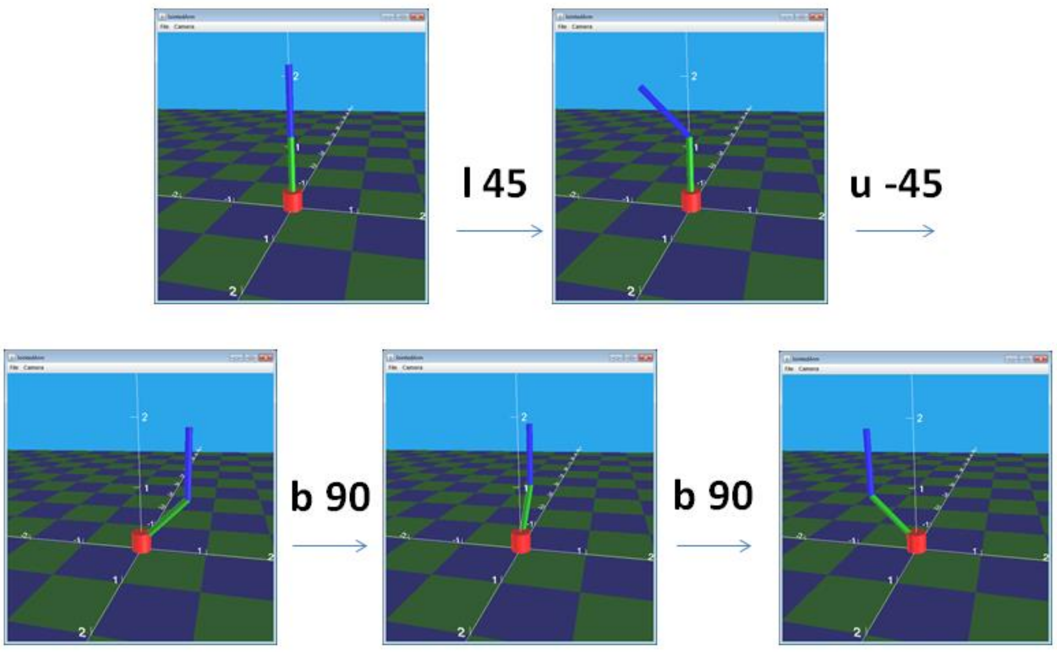


Figure 22. The Moving Arm.

The details of how the user's input is read aren't that relevant, but eventually the following code is called to process a command:

```
  private static void processCmd(String[] toks, JointedArm arm)
  {
    if (toks.length != 2) {
      System.out.println("Wrong no. of args for \"" +toks[0] +"\"");
      return;
    }

    if (toks[0].equals("b"))
      arm.rotateBase(Simple3D.getDouble(toks[1]));
    else if (toks[0].equals("u"))
      arm.rotateUpper(Simple3D.getDouble(toks[1]));
    else if (toks[0].equals("l"))
      arm.rotateLower(Simple3D.getDouble(toks[1]));
    else
      System.out.println("Unrecognized command: " + toks[0]);
  }  // end of processCmd()
```

5.4. Observing the TransformGroups

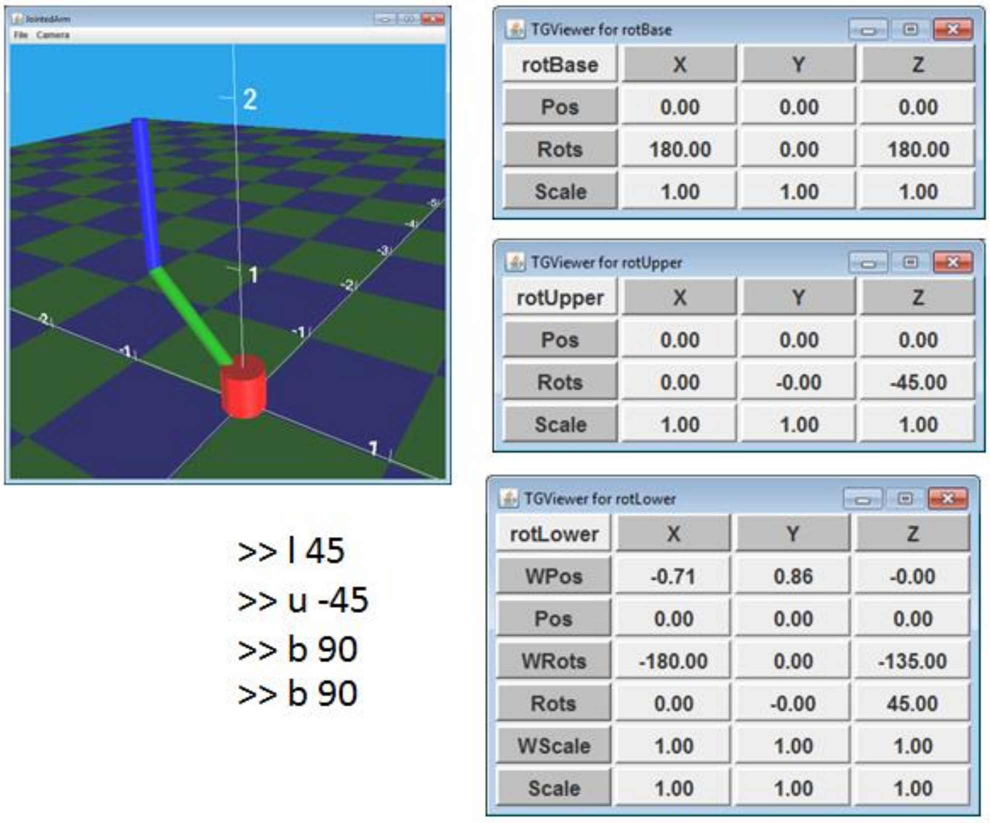
Although the effects of a command can be observed by looking at how the cylinders move inside the Java 3D window, it's sometimes useful to get more accurate information about the current position, rotation and scaling stored in a TransformGroup. This is possible by creating a TGViewer window, which displays a TransformGroup's current values.

Three TGViewers are created in JointedArm() to display the data in rotBase, rotUpper, and rotLower:

```
// code from JointedArm()
   :
new TGViewer("rotLower", rotLower, true);   // show world coords
  :
new TGViewer("rotUpper", rotUpper);
  :
new TGViewer("rotBase", rotBase);
```

The TGViewer for rotLower includes a boolean argument which means that it also displays the TransformGroup values mapped to the world coordinate system.

Figure 23 shows the three TGViewer windows after the arm has been moved to the last position in Figure 22.



```
>> l 45
>> u -45
>> b 90
>> b 90
```

rotBase	X	Y	Z
Pos	0.00	0.00	0.00
Rots	180.00	0.00	180.00
Scale	1.00	1.00	1.00

rotUpper	X	Y	Z
Pos	0.00	0.00	0.00
Rots	0.00	-0.00	-45.00
Scale	1.00	1.00	1.00

rotLower	X	Y	Z
WPos	-0.71	0.86	-0.00
Pos	0.00	0.00	0.00
WRots	-180.00	0.00	-135.00
Rots	0.00	-0.00	45.00
WScale	1.00	1.00	1.00
Scale	1.00	1.00	1.00

Figure 23. TGViewers for the Jointed Arm.

The WPos, WRots, and WScale rows in the rotLower TGViewer give the position, rotation, and scaling of the TransformGroup in world coordinates. For example, the WPos value (-0.71, 0.86, 0) corresponds to the joint's current position in the world, which is at the point where the green and blue cylinders meet.

The rotation information supplied by TGViewer is generally less useful since it can be difficult to reconcile the reported values with the operations that were carried out. This is readily apparent in the Rots values reported for rotBase, which states that the base has rotated 180 degrees around the x-axis and 180 degrees around the z-axis. However, the two operations applied to rotBase amount to 180 degrees rotation around the *y-axis*.

In fact, both values are equivalent, reflecting how there's no unique way to define a rotation in 3D space based on the x-, y-, and z- axes. The math utilized by Simple3D.getRotations() to convert the 4x4 matrix in the TransformGroup's Transform3D object into three Euler angles is described in https://mino-git.github.io/rtcw-wet-blender-model-tools/publications/EulerToMatrix.pdf.