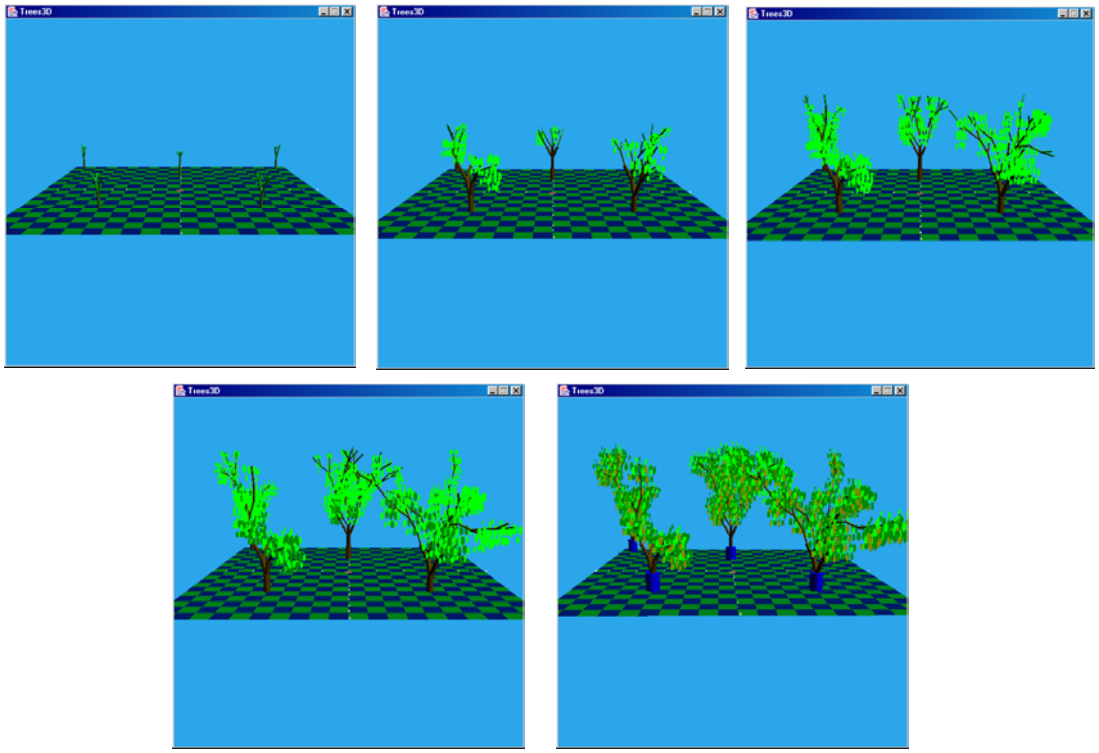
Chapter 17.8. Trees that Grow

We've seen a few ways that trees can be added to a scene. In chapter 10, a red palm trees was loaded as a 3D model, while trees were represented as images pasted onto transparent QuadArray surfaces in chapter 17.5. The drawback of both approaches is that the trees never change, never grow.

Figures 1-5 show a sequence of screen shots of the Trees3D application. Five trees grow from saplings, young green shoots turn brown, leaves sprout, all taking place over a period of a few seconds.



Figures 1-5. Growing Trees.

A tree is a collection of tree limbs, each one encoded as a TreeLimb object. TreeLimb collects together the techniques for making a branch grow longer, thicker, and change colour. Instead of using a GeometryUpdater interface (as in the particle systems demos of chapter 12), the underlying shape (a cylinder) is manipulated by non-uniform scaling, a much simpler approach.

The leaves at the end of a branch are represented by two ImagesCsSeries 'screens' which show pictures of leaves that are always oriented towards the viewer. The image on a screen can be replaced by another one, perhaps an image containing more leaves, creating the effect that the leaves are 'growing' and changing.

The modifications carried out on a tree limb in each time frame are specified by 'rules' in a GrowthBehavior object, which is triggered every 100ms. The rules can be easily adjusted to make limbs change in a variety of ways.

Each time Trees3D is executed, the trees will look different due to random elements in the rules.

An approach *not* used here are rewrite rules in the style of a Lindenmayer system (L-system). We briefly compare our rule mechanism with L-systems at the end of the chapter.

UML Diagrams for Trees3D

Figure 6 shows the UML diagrams for all the classes in the Trees3D application. The class names and public methods are shown.

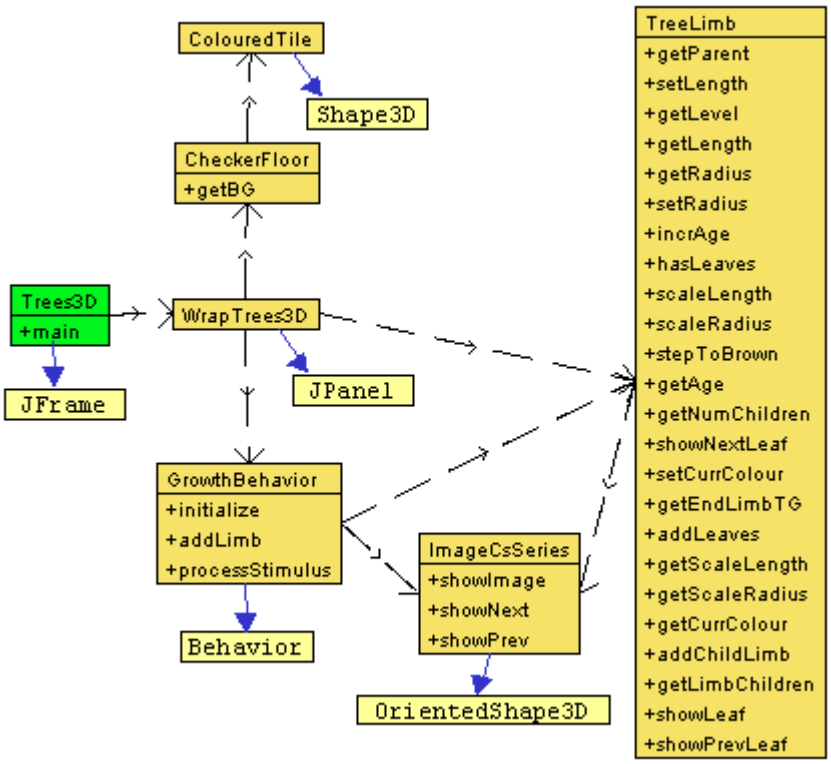


Figure 6. UML Class Diagrams for Trees3D.

Trees3D is the top-level JFrame for the application, while WrapTrees3D sets up the scene. It creates five TreeLimb objects which grow into five trees like those in Figures 1-5. The growth, colour change, creation of new limbs, and the appearance of leaves drawn onto ImageCsSeries screens, is governed by the GrowthBehavior object. The rules that guide the trees' development are located in that class. They utilize the rich set of public methods exposed by TreeLimb.

The WrapTrees3D Class

Most of the tree-related activity is started in growTrees(), which creates five TransformGroups and five TreeLimb objects. For example:

```
// starting position for the first tree: (0,0,-5)
Transform3D t3d = new Transform3D();
t3d.set( new Vector3f(0,0,-5));
TransformGroup tg0 = new TransformGroup(t3d);
sceneBG.addChild(tg0);

// create the tree
TreeLimb t0 = new TreeLimb(Z_AXIS, 0, 0.05f, 0.5f, tg0, null);
```

The TransformGroup reference (tg0) is passed to the TreeLimb constructor, where it is used to position the limb.

Loading Leaves

growTrees() also loads a sequence of leaf images, storing them in an ImageComponent2D[] array. This is a matter of efficiency: each TreeLimb will utilize this array if it requires leaves rather than loading its own copy of the images.

```
// load the leaf images used by all the trees
ImageComponent2D[] leafIms = loadImages("images/leaf", 6);
```

The loadImages() method:

```
private ImageComponent2D[] loadImages(String fNms, int numIms)
/* Load the leaf images: they all start with fNms, and there are
   numIms of them.  */
{
  String filename;
  TextureLoader loader;
  ImageComponent2D[] ims = new ImageComponent2D[numIms];
  System.out.println("Loading " + numIms +" textures from "+ fNms);
  for (int i=0; i < numIms; i++) {
    filename = new String(fNms + i + ".gif");
    loader = new TextureLoader(filename, null);
    ims[i] = loader.getImage();
    if(ims[i] == null)
      System.out.println("Load failed for: " + filename);
    ims[i].setCapability(ImageComponent2D.ALLOW_SIZE_READ);
  }
  return ims;
}
```


The leaf images are in the files images/leaf0-5.gif, as shown in Figure 7.

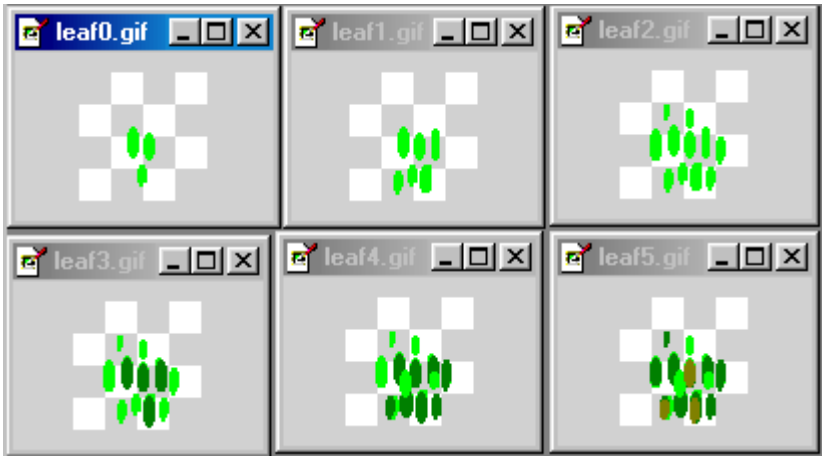


Figure 7. The Leaf Images in leaf0-5.gif.

The idea is to gradually switch between the images, going from leaf0.gif to leaf5.gif, updating the ImageCsSeries screens with the next image in the sequence. The effect will appear like leaves growing, and turning darker.

Clearly, there's some room for artistic improvement in the drawing style :). However, the effect is still surprisingly good, especially when viewed at some distance, with many leaves overlapping. Later images were created by modifying earlier ones, so the run time transition from one image to the next is fairly smooth.

The grey and white squares indicate that the GIFs' backgrounds are transparent.

Since the scene will contain numerous semi-transparent textures wrapped over quads, it is necessary to turn on Java 3D's depth-sorting of transparent objects on a per-geometry basis:

```
View view = su.getViewer().getView();
view.setTransparencySortingPolicy(View.TRANSPARENCY_SORT_GEOMETRY);
```

This is carried out in the WrapTrees3D() constructor. If depth-sorting is not switched on then the relative ordering of the leaf images will frequently appear wrong as the viewer moves around the trees.

Getting Ready for Growth

WrapTrees3D's growTrees() also initializes the GrowthBehavior object:

```
// the behaviour that manages the growing of the trees
GrowthBehavior grower = new GrowthBehavior(leafIms);
grower.setSchedulingBounds( bounds );

// add the trees to GrowthBehavior
grower.addLimb(t0);
grower.addLimb(t1);
```



```
    grower.addLimb(t2);
    grower.addLimb(t3);
    grower.addLimb(t4);

    sceneBG.addChild( grower );
```

The TreeLimb Class

Each TreeLimb object builds a subgraph like the one in Figure 8.

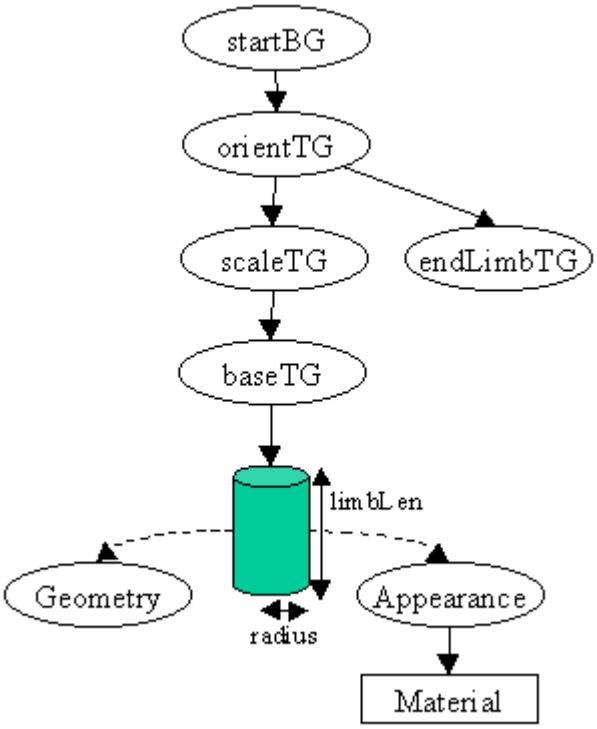


Figure 8. The TreeLimb Subgraph.

The startBG BranchGroup is linked to a 'parent' limb via the parent's endLimbTG node. Since the link is made at execution time, Java 3D requires the use of a BranchGroup.

orientTG holds the orientation of the limb (its initial axis of rotation, and angle to that axis). Once set, this cannot change during execution.

scaleTG manages the scaling of the x-, y-, and z- dimensions of the cylinder. The x- and z- values are kept the same (they represent the radius), while the y-axis is for the length.

The central point for Java 3D's Cylinder is its middle, but we want the origin to be its base, at the place where the cylinder connects to the parent limb. baseTG moves the cylinder up the y-axis by length/2 so this is the case.

The capabilities for the Cylinder's Material node component are set so that its colour can be adjusted at run time.

Child limbs or leaves are attached to this limb through endLimbTG. The transform inside endLimbTG is an offset of *almost* the cylinder's scaled length. It is a little less than the length, so that child limbs will overlap the parent. This partly hides any gaps between the limbs when a child is orientated at an extreme angle.

The endLimbTG node is not attached to Cylinder since that would make it prone to scaling, which would also affect any child limbs attached to endLimbTG. Scaling is restricted to the limb's cylinder.

Aside from the subgraph data structure, TreeLimb maintains a variety of other information related to a tree limb, including a reference to its parent, its current colour, its current age, its level in the overall tree (the first branch is at level 1), and whether it is showing leaves.

The age is simply a counter, set to 0 when the limb is created, and incremented in each time interval by the GrowthBehaviour object. Each limb will have a different age depending on when it was added to the tree.

The large number of public methods in TreeLimb can be roughly classified into five groups:

- scaling of the cylinder's radius or length;
- colour adjustment;
- parent and children methods;
- leaves-related;
- various others (e.g. accessing the limb's current age).

The TreeLimb constructor takes various arguments:

```
public TreeLimb(int axis, double angle, float rad, float len,
                      TransformGroup startLimbTG, TreeLimb par)
```

The axis and angle are used by orientTG, while rad and len become the radius and length of the new cylinder. startLimbTG will be assigned the TransformGroup of the parent limb where this limb will be attached. par is a reference to the parent as a TreeLimb object.

Subgraph Creation

The subgraph in Figure 8 is constructed by buildSubgraph():

```
private void buildSubgraph(TransformGroup startLimbTG)
/* Create the scene graph.
   startLimbTG is the parent's endLimbTG. */
{
```



```
    BranchGroup startBG = new BranchGroup();

    // set the limb's orientation
    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);
    }

    // scaling node
    scaleTG = new TransformGroup();
    scaleTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ);
    scaleTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE);

    // limb subgraph's sequence of TGs
    startBG.addChild(orientTG);
    orientTG.addChild(scaleTG);
    scaleTG.addChild( makeLimb() );

    TransformGroup endLimbTG = locateEndLimb();
    orientTG.addChild(endLimbTG);

    startBG.compile();

    startLimbTG.addChild(startBG);   //connect to parent's endLimbTG
  } // end of buildSubgraph()
```

The cylinder and its components are built inside makeLimb():

```
  private TransformGroup makeLimb()
  // a green cylinder whose base is at (0,0,0)
  {
    // fix limb's start position
    TransformGroup baseTG = new TransformGroup();
    Transform3D trans1 = new Transform3D();
    trans1.setTranslation( new Vector3d(0, limbLen/2, 0) );
                                        // move up length/2
    baseTG.setTransform(trans1);

    Appearance app = new Appearance();
    limbMaterial = new Material(black, black, green, brown, 50.f);
    limbMaterial.setCapability( Material.ALLOW_COMPONENT_READ);
    limbMaterial.setCapability( Material.ALLOW_COMPONENT_WRITE);
       // can change colours; only the diffuse colour will be altered

    limbMaterial.setLightingEnable(true);

    app.setMaterial( limbMaterial );
    Cylinder cyl = new Cylinder( radius, limbLen, app);

    baseTG.addChild( cyl );
    return baseTG;
```



```
  }  // end of makeLimb()
```

The Material's capabilities must be set since colour change will be carried out at run time. Initially the limb is green.

The radius and length of the cylinder are stored in the radius and limbLen globals. They will never change (the cylinder's geometry is not updated). Instead, scaling is applied to the cylinder through the scaleTG node.

The endLimbTG node is connected to the subgraph in locateEndLimb():

```
private TransformGroup locateEndLimb()
{
  // fix limb's end position, and store in endLimbTG
  endLimbTG = new TransformGroup();
  endLimbTG.setCapability( TransformGroup.ALLOW_CHILDREN_EXTEND );
  endLimbTG.setCapability( TransformGroup.ALLOW_TRANSFORM_READ );
  endLimbTG.setCapability( TransformGroup.ALLOW_TRANSFORM_WRITE );

  Transform3D trans2 = new Transform3D();
  trans2.setTranslation(new Vector3d(0, limbLen*(1.0-OVERLAP), 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(trans2);

  return endLimbTG;
}  // end of locateEndLimb()
```

It is important to set the necessary capabilities in locateEndLimb(). The `ALLOW_CHILDREN_EXTEND` bit will permit BranchGroup nodes to be attached to this node.

Scaling

There are several public scaling methods: they either employ a scale factor, or scale so that the radius (or length) becomes a desired value.

We consider setLength(), which scales the cylinder's length until it is the specified amount:

```
// global for storing scaling info
private Vector3d scaleLimb;
     :

public void setLength(float newLimbLen)
// change the cylinder's length to newLimbLen
// (by changing the scaling)
{ double scaledLimbLen = ((double) limbLen) * scaleLimb.y;
  double lenChange = ((double) newLimbLen) / scaledLimbLen;
  scaleLength( lenChange );
}
```


To simplify the calculations, the current scaling factors (in the x-, y-, and z-directions) are maintained in scaleLimb. The current scaled limb length is stored in scaledLimbLen. The desired scaling is the factor to take the length from scaledLimbLen to the required newLimbLen value.

The scaling is done by scaleLength():

```
public void scaleLength(double yChange)
{ scaleLimb.y *= yChange;
  applyScale();
}

private void applyScale()
{
  moveEndLimbTG( scaleLimb.y);

  scaleTG.getTransform(currTrans);
  currTrans.setScale(scaleLimb);
  scaleTG.setTransform(currTrans);
}
```

applyScale() applies the new scaling to scaleTG, which changes the perceived length of the cylinder. However, this change will not automatically affect the endLimbTG node (the node that represents the 'end' of the limb), since it is not attached to the graph below scaleTG. The call to moveEndLimbTG() adjusts the node's position so it stays located at the end of the cylinder.

```
private void moveEndLimbTG( double yScale)
/* yScale is the amount that the Cylinder is about to
   be scaled. Apply it to the y- value in endLimbTG  */
{
  endLimbTG.getTransform( currTrans );
  currTrans.get( endPos );         // current posn of endLimbTG
  double currLimbLen = endPos.y;
    // current y-posn, the cylinder length including scaling

  double changedLen =
        ((double) limbLen*(1.0-OVERLAP) * yScale) - currLimbLen;
    // change in the y- value after scaling has been applied

  endPos.set(0, changedLen, 0);       // store the length change
  toMove.setTranslation( endPos );    // overwrite previous trans
  currTrans.mul(toMove);
  endLimbTG.setTransform(currTrans);  // move endLimbTG
}  // end of moveEndLimbTG()
```

endLimbTG's (x,y,z) position is extracted to the endPos vector. This position corresponds to the end of the *scaled* cylinder.

The necessary position change is calculated by multiplying the cylinder's physical length by the new scale factor in yScale, and subtracting the endPos y-value. We also factor in an overlap.

Changing the Limb's Colour

The capabilities for changing the limb's Material were set up in makeLimb(), described earlier. Also, a global, limbMaterial, stores a reference to the node to make it simple to change:

```
public void setCurrColour(Color3f c)
// Change the limb's colour to c.
{ currColour.x = c.x;
  currColour.y = c.y;
  currColour.z = c.z;
  limbMaterial.setDiffuseColor( currColour );
}
```

We want to change the limb's colour from green to brown, spread over several time frames, to give the effect of "aging oak". This is achieved by precalculating red, green, and blue transitions that will change the RGB values for green to brown over the course of `MAX_COLOUR_STEP` steps:

```
// globals
private final static int MAX_COLOUR_STEP = 15;

private final static Color3f green = new Color3f(0.0f, 1.0f, 0.1f);
private final static Color3f brown = new Color3f(0.35f, 0.29f, 0.0f);

// incremental change in terms of RGB to go from green to brown
private float redShift = (brown.x - green.x)/
                          ((float) MAX_COLOUR_STEP);
private float greenShift = (brown.y - green.y)/
                           ((float) MAX_COLOUR_STEP);
private float blueShift = (brown.z - green.z)/
                          ((float) MAX_COLOUR_STEP);
```

The `redShift`, `greenShift`, and `blueShift` values are utilized in stepToBrown():

```
public void stepToBrown()
// Incrementally change the limb's colour from green to brown
{
  if (colourStep <= MAX_COLOUR_STEP) {
    currColour.x += redShift;
    currColour.y += greenShift;
    currColour.z += blueShift;
    limbMaterial.setDiffuseColor( currColour );
    colourStep++;
  }
}
```

stepToBrown() will be repeatedly called until the limb has turned brown.

Leaves on Trees

Two ImageCsSeries objects display leaves at the end of a branch. This makes the 'mass' of the leaves seem greater, especially since the images are offset from each

other. The two objects are connected to the limb via BranchGroup nodes since the links are formed at run time.

```
public void addLeaves(ImageCsSeries fls, ImageCsSeries bls)
// Leaves are represented by two ImageCsSeries 'screens'
{
  if (!hasLeaves) {
    frontLeafShape = fls;
    backLeafShape = bls;

    // add the screens to endLimbTG, via BranchGroups
    BranchGroup leafBG1 = new BranchGroup();
    leafBG1.addChild(frontLeafShape);
    endLimbTG.addChild(leafBG1);

    BranchGroup leafBG2 = new BranchGroup();
    leafBG2.addChild(backLeafShape);
    endLimbTG.addChild(leafBG2);

    hasLeaves = true;
  }
}
```

The positioning of the ImageCsSeries objects is done when they are created, inside GrowthBehavior.

The other leaf-related methods in TreeLimbs pass requests to the ImageCsSeries objects; the requests change the image currently being displayed. For instance:

```
public void showNextLeaf()
// show the next leaf image
{ if (hasLeaves) {
    frontLeafShape.showNext();
    backLeafShape.showNext();
  }
}
```

The GrowthBehavior Class

The GrowthBehaviour object created by WrapTrees3D maintains an ArrayList of TreeLimb objects, and an ImageComponent2D[] array of leaf pictures. The pictures are passed to it at construction time, while the first five TreeLimb objects are stored via calls to addLimb():

```
// globals
private final static int TIME_DELAY = 100;  //ms

private WakeupCondition timeOut;
private ArrayList treeLimbs;           // of TreeLimb objects
private ImageComponent2D[] leafIms;    // a sequence of leaf images


public GrowthBehavior(ImageComponent2D[] lfIms)
{ timeOut = new WakeupOnElapsedTime(TIME_DELAY);
```



```
    treeLimbs = new ArrayList();
    leafIms = lfIms;
  }


  public void addLimb(TreeLimb limb)
  {  treeLimbs.add(limb);  }
```

processStimulus() is called every `TIME_DELAY` milliseconds, which calls applyRulesToLimbs(). applyRulesToLimbs() cycles through the TreeLimb objects in the ArrayList, calling applyRules() for each one.

```
  private void applyRulesToLimbs()
  {
    TreeLimb limb;
    for(int i=0; i < treeLimbs.size(); i++) {
      limb = (TreeLimb) treeLimbs.get(i);
      applyRules(limb);
      limb.incrAge();   // a limb gets older after each iteration
    }
  }
```

Rules controlling the modification of a tree limb are placed in applyRules(). Currently it holds seven rules.

```
  private void applyRules(TreeLimb limb)
  // Apply rules to the tree limb.
  {
    // get longer
    if ((limb.getLength() < 1.0f) && !limb.hasLeaves())
      limb.scaleLength(1.1f);

    // get thicker
    if ((limb.getRadius() <= (-0.05f*limb.getLevel()+0.25f))
              && !limb.hasLeaves())
      limb.scaleRadius(1.05f);

    // get more brown
    limb.stepToBrown();

    // spawn some child limbs
    int axis;
    if ((limb.getAge() == 5) && (treeLimbs.size() <= 256)
          && !limb.hasLeaves() &&  (limb.getLevel() < 10)) {
      axis = (Math.random() < 0.5) ? Z_AXIS : X_AXIS;
      if (Math.random() < 0.85)
        makeChild(axis, randomRange(10,30), 0.05f, 0.5f, limb);

      axis = (Math.random() < 0.5) ? Z_AXIS : X_AXIS;
      if (Math.random() < 0.85)
        makeChild(axis, randomRange(-30,-10), 0.05f, 0.5f, limb);
    }

    // start some leaves
    if ( (limb.getLevel() > 3) && (Math.random() < 0.08) &&
         (limb.getNumChildren() == 0) && !limb.hasLeaves() )
```



```
      makeLeaves(limb);

    // grow the leaves
    if (limb.getAge()%10 == 0)
      limb.showNextLeaf();

    // turn the base limb into a 'blue bucket'
    if ((limb.getAge() == 100) && (limb.getLevel() == 1)) {
      limb.setRadius( 2.0f*limb.getRadius());
      // limb.setLength( 2.0f*limb.getLength());
      limb.setCurrColour( new Color3f(0.0f, 0.0f, 1.0f));
    }

  }  // end of applyRules()
```

Most rules have an if-then form, where the action is only carried out if the conditions evaluate true for the current limb. Due to the repeating nature of GrowthBehavior, each rule will be applied to each tree limb in each time interval. This means that change is expressed in incremental terms in the rules.

The 'thickness' rule:

```
if ((limb.getRadius() <= (-0.05f*limb.getLevel()+0.25f))
                && !limb.hasLeaves())
      limb.scaleRadius(1.05f);
```

The equation `-0.05*limb.getLevel()+0.25` relates the maximum radius to the limb's level. For example, a limb touching the ground (level == 1) can have a larger maximum radius than a branch higher up the tree. This means that branches will get less thick the higher up the tree they appear, as in nature. The hasLeaves() part of the condition stops branches from growing thicker once they have leaves.

The child limbs rule:

```
  if ((limb.getAge() == 5) && (treeLimbs.size() <= 256)
           && !limb.hasLeaves() &&  (limb.getLevel() < 10)) {
      axis = (Math.random() < 0.5) ? Z_AXIS : X_AXIS;
      if (Math.random() < 0.85)
        makeChild(axis, randomRange(10,30), 0.05f, 0.5f, limb);

      axis = (Math.random() < 0.5) ? Z_AXIS : X_AXIS;
      if (Math.random() < 0.85)
        makeChild(axis, randomRange(-30,-10), 0.05f, 0.5f, limb);
  }
```

The four conditions only permit a child limb to appear if the parent is at least five time intervals old, the total number of limbs in the scene is less or equal to 256, the parent has no leaves, and the branch isn't too far up the tree.

Math.random() is used to randomize the orientation axis, and to make it less certain that two children will be spawned. randomRange() returns a random number (in this case, an angle) in the specified range.

makeChild()'s definition:

```
private void makeChild(int axis, double angle, float rad,
                                    float len, TreeLimb par)
{ TransformGroup startLimbTG = par.getEndLimbTG();
  TreeLimb child = new TreeLimb(axis, angle, rad, len,
                                       startLimbTG, par);
  treeLimbs.add(child);   // add new limb to the ArrayList
} // end of makeChild()
```

Leaves are spawned with the rule:

```
  if ( (limb.getLevel() > 3) && (Math.random() < 0.08) &&
       (limb.getNumChildren() == 0) && !limb.hasLeaves() )
    makeLeaves(limb);
```

If a limb is far enough up the tree, has no children or existing leaves, then it is has a small chance of bursting into leaf. makeLeaves() packages up the creation of two ImageCsSeries objects, which are passed to the TreeLimb object:

```
private void makeLeaves(TreeLimb limb)
{
  ImageCsSeries frontLeafShape =
      new ImageCsSeries(0.5f, 2.0f, leafIms);
  ImageCsSeries backLeafShape =
      new ImageCsSeries(-0.5f, 2.0f, leafIms);

  limb.addLeaves(frontLeafShape, backLeafShape);
}
```

The ImageCsSeries Class

An ImagesCsSeries object is a 'screen', centered at (0,0,0), that can display an image from a series passed to it at creation time as an ImageComponent2D array. The screen is a subclass of OrientedShape3D, configured to rotate around the point (0,0, zCoord) relative to its origin.

By default, the first image in the array is shown, and methods must be explicitly called to change the displayed picture. There is no default animation. A version of ImageCsSeries with animation can be found in FPShooter3D in chapter 15.

The constructor sets up the rotation point:

```
public ImageCsSeries(float zCoord, float screenSize,
                                  ImageComponent2D[] ims)
{ this.ims = ims;
  imIndex = 0;
  numImages = ims.length;

  // set the orientation mode
  setAlignmentMode(OrientedShape3D.ROTATE_ABOUT_POINT);
  setRotationPoint(0.0f, 0.0f, zCoord);
```



```
    createGeometry(screenSize);
    createAppearance();
  }
```

A look back at makeLeaves() used by the leaf creation rule shows that two ImageCsSeries screens are created with different rotation points: one 0.5 units in front of its location, the other 0.5 units behind. This means that the two screens will rotate differently as the user's viewpoint moves around a tree. This creates a much larger 'mass' of leaves which always seems to surround the tree limb.

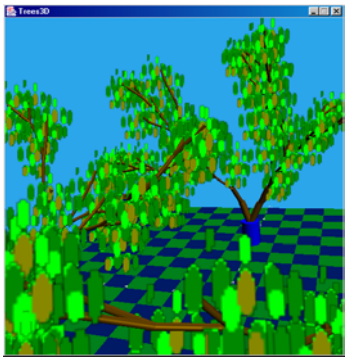


Figure 9. A Mass of Leaves Around Each Limb.

The createGeometry() and createAppearance() methods are unchanged from the version of the class in FPShooter3D. The geometry is a mix of coordinates and texture information. The appearance employs a blended form of transparency so that transparent parts of the image will not render onto the shape.

The ImageCsSeries class in FPShooter3D uses a time-delayed loop triggered by a call to showSeries(). The Trees3D version has several methods for replacing the currently displayed image with another one. For instance:

```
public void showNext()
// show the next image in the sequence
{ if (imIndex < numImages-1) {
    imIndex++;
    texture.setImage(0, ims[imIndex]);
  }
}
```

Comparison with Lindenmayer Systems

Lindenmayer systems (L-systems) consist of rewrite rules, and have been widely used for plant modeling and simulation. Perhaps surprisingly, there is a direct mapping between the string expansions of the rule system and a visual representation of the plant. An example, using a bracketed L-system, will give an idea of how this works.

The L-system contains a single start string "F", and rewrite rule:

F --> F [-F] F [+F] F

The visual characterization is obtained by thinking of each 'F' as a 'limb' of the plant. The bracketed notation is viewed as a branch-creation operator; the '-' as a rotation to the right for the branch, '+' a left rotation.

Consequently, the rewrite of 'F' to "F[-F]F[+F]F" can be seen as the plant expansion in Figure 10.

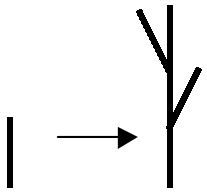


Figure 10. First Rewrite of "F".

Since each limb is an "F", rewriting can continue, creating longer strings, and more complex plant-like shapes, as shown in Figure 11.

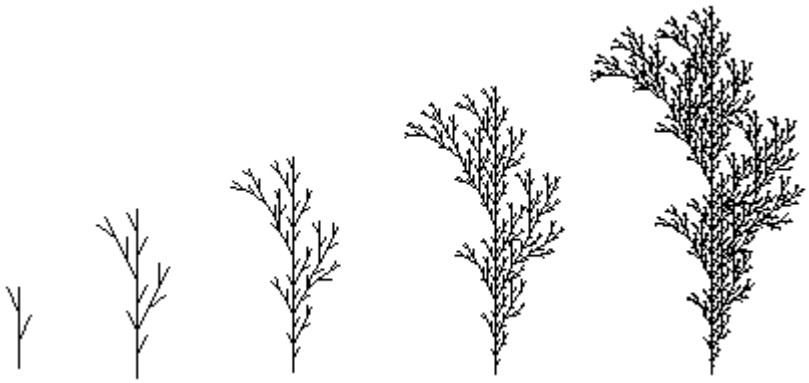


Figure 11. A Sequence of Rewrites.

There are a great variety of L-system languages, perhaps the richest being cpfg, available from http://www.cpsc.ucalgary.ca/Research/bmv/. It includes a full range of programming constructs, data structures, library functions, and various extensions such as parameterized L-system symbols, and sub L-systems.

Two good introductions to L-Systems, available online:

> *An Introduction to Lindenmayer Systems*
> Gabriela Ochoa, February 1998
> http://www.cogs.susx.ac.uk/lab/nlp/gazdar/teach/atc/1998/web/ochoa/
>
> *Simulating Plant Growth*
> Marco Grubert
> ACM Crossroads Student Magazine, Nov. 2001
> http://www.acm.org/crossroads/xrds8-2/plantsim.html

Another good starting point is Google's directory listings for Lindenmayer systems:

http://directory.google.com/Top/Computers/Artificial_Life/Lindenmayer_Systems/

Java 3D was used by René Gressly to implement L-systems for growing bushes, trees and flowers in a landscape. The user chooses where to place plants, and then walks around the scene as the plants grow (http://www.hta-bi.bfh.ch/~swc/DemoJ3D/VirtualPlants/).

Chris Buckalew implemented a Java 3D L-Systems engine that uses recursion to parse the start string and replace string elements. It is part of a lab exercise in his CSC 474 Computer Graphics course (http://www.csc.calpoly.edu/~buckalew/474Lab7-W02.html).

Scott Teresi has written code that reads a two-dimensional L-System, and renders it in 3D (http://teresi.us/html/main/programming.html)

So Why Not Use L-Systems?

The truth is that originally I did use L-Systems. A student and I developed code along the lines of Chris Buckalew's example (see above). Unfortunately, the heavy use of recursion meant that only one or two trees of perhaps 10 or so levels can be generated before Java requires a major memory extension.

The real problem is with the L-system formalism itself – it is very difficult to represent *incremental* change using L-system rewrite rules. As figure 10 indicates, each expansion creates a more complex tree, but it is hard to see how the fancier tree has 'grown' out of the simpler one. What part of the current tree is new wood, which is old wood that has grown a bit?

An L-System sees growth as a new tree completely replacing the old one. That doesn't matter when the tree is a mathematical abstraction, but has important consequences when implementing growth in Java 3D. The natural approach, and the most disastrous from an efficiency point of view, is to discard the current tree at the start of a rewrite and generate a new one matching the new string expansion.

The rules notation used here, as typified by the rules in applyRules(), are phrased in terms of incremental change to existing limbs. New limbs can be added, but only by explicitly spawning children. This has the practical benefit that thousands of limbs can be created before the application needs additional heap allocation.

Another drawback of the Lindenmayer notation is its lack of tree nomenclature. For instance, it is not possible to talk about the *parent* of a node, its *children*, its *level* in the tree, and so on. To be fair, some of these capabilities can be programmed by using parameterized L-system rules.

Basic L-Systems have no notion of global time, or the age of individual limbs. Again, this can be remedied with additional parameters in rules.

L-System rules tend to be embedded in procedural languages, and so it's difficult to create new plant node types (or classes) with their own data, operators, and behavior. This presents no problem to Java of course; we could start by subclassing TreeLimb.

