Chapter 26.2. A Multitextured Landscape

The FractalLand3D example in chapter 26 uses quad-level texturing based on the average height of each quad, making the ground look a bit like a patchwork quilt. This chapter's focus is on using *multitexturing* to make the landscape appear more natural, with less severe boundaries between the various textures.

Figure 1 shows a view of the TexLand3D application.

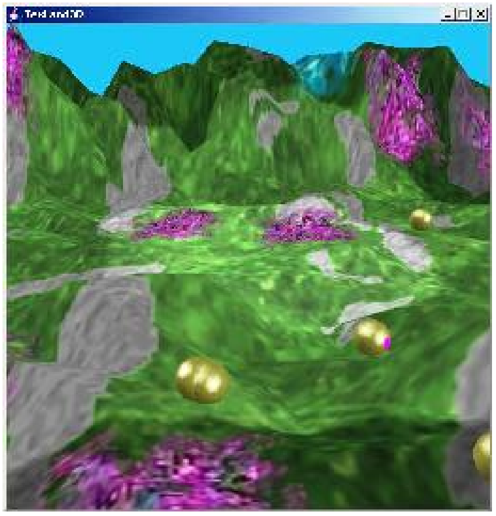

Figure 1. The TexLand3D Application.

Multitexturing is the application of several textures to the same shape, a surprisingly versatile mechanism since it's possible to combine textures is numerous ways, especially when they have alpha channels.

The floor mesh (composed from quads) is covered with a repeating grass texture, completely replacing the geometry's underlying colour. A partially transparent stone texture is applied on top of the grass. The grass shows through in the areas where the stone texture is transparent.

A *light map* texture is modulated with the grass and stones textures to add shadows and light without the underlying shape requiring normals or a light-enabled material. TexLand3D can load a light map from an image file, or draw one at execution time.

The texture coordinates for the grass, stone, and light map textures are generated at runtime, and the code can be easily tweaked to adjust the repetition frequency of each texture. For example, in Figure 1 the grass texture is repeated 16 times, the stone texture 4 times, and the light map appears once.

Multitexturing is used to 'splash' purple flowers and pools of water onto random areas of the floor. A SplashShape object is made from quads copied from the floor mesh, and covered with a combination of a standard texture image and an *alpha map* texture. The map specifies what areas of the standard texture will be opaque, translucent, and transparent. Each SplashShape generates its own semi-random alpha map at runtime, which produces unique borders for each shape's flowerbed or pool.

My SplashShape class owes a great deal to David Yazel's SplatShape class, and Justin Couch's AlphaDemo class.

TexLand3D illustrates the utility of a *heights map* for constructing a landscape. Heights for the floor's vertices are generated using a hill-raising algorithm, and stored in a 2D array of floats. This array is employed by the MultiFloor object to create the multitextured floor, by each SplashShape object to copy floor quads, and by roaming balls to navigate over the landscape (four of them can be seen in Figure 1).

1. TexLand3D Overview

Figure 2 shows the class diagrams for the application; only public methods are listed.

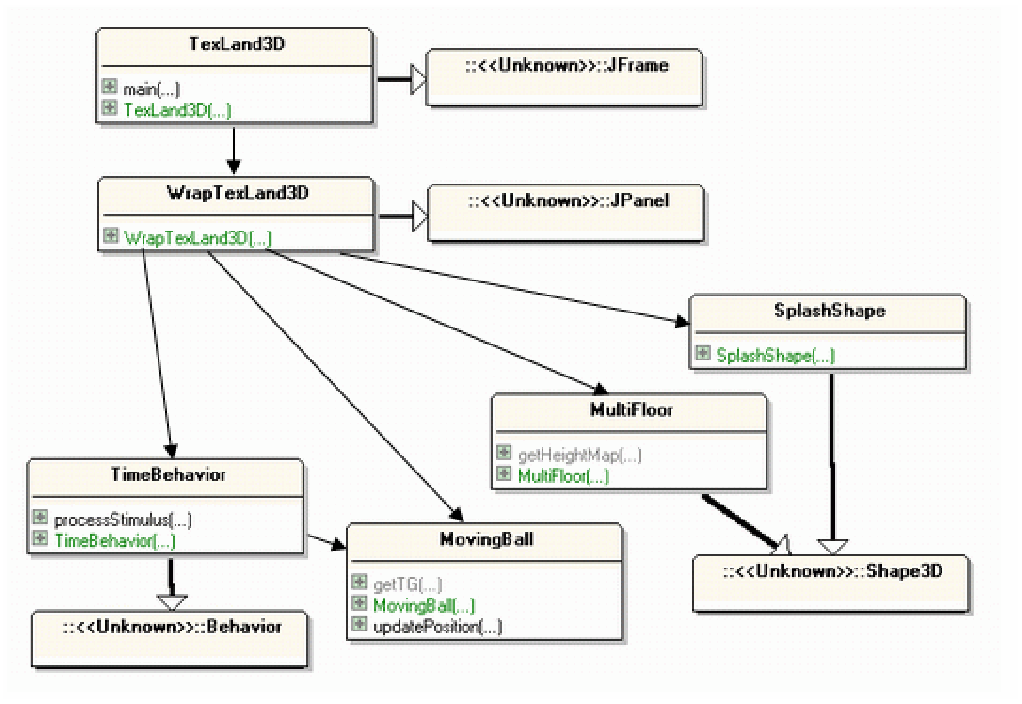

Figure 2. Class Diagrams for TexLand3D.

TexLand3D sets up the top-level JFrame, while WrapTexLand3D renders the 3D scene inside a JPanel. MultiFloor generates the heights map, and textures the floor with grass, stones, and the light map. The splashes of flowers and water are added with SplashShape objects. The balls are represented by MovingBall objects, which are animated via periodic calls to their updatePosition() method by TimeBehavior.

2. Building the Scene

WrapTexLand3D constructs the scene graph shown in Figure 3.

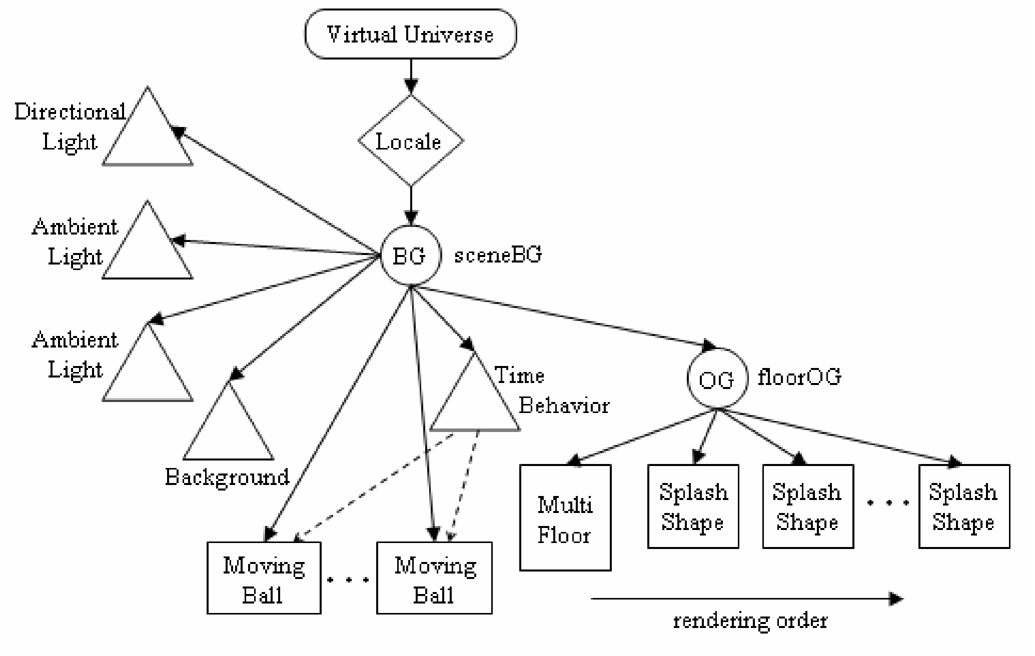

Figure 3. The TexLand3D Scene Graph.

To make things simpler, I've hidden the details of the scene graph branches for the moving balls, the multitextured floor (MultiFloor), and the splash shapes; I'll explain them later in this chapter.

The nodes at the top of the graph, the lights, and the background node are built in the same way as in previous examples, through calls from createSceneGraph().

```
// globals
private static final int BOUNDSIZE = 100;  // larger than world
private BranchGroup sceneBG;
private BoundingSphere bounds;


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

  lightScene();         // the lights
  addBackground();      // the sky
  addFloor();           // the multi-textured floor (and splashes)
  movingBalls();        // the moving balls

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

addFloor() constructs the subtree on the right of Figure 3, starting at the OG (OrderedGroup) node. movingBalls() creates the MovingBalls branches, and the TimeBehavior instance.

2.1. Creating the Floor

addFloor() builds the multitextured floor and the splash shapes that lie on top of it.

```
// global
private float[][] heights;  // heights map for the floor


private void addFloor()
{
  MultiFloor floor = new MultiFloor("grass.gif", 4,
                                        "stoneBits.gif", 2);
  heights = floor.getHeightMap();

  /* Use an ordered group of to avoid rendering conflicts
     between the floor and the splash shapes. */
  OrderedGroup floorOG = new OrderedGroup();
  floorOG.addChild(floor);

  // load the textures for the splashes
  Texture2D flowersTex = loadTexture("images/flowers.jpg");
  Texture2D waterTex = loadTexture("images/water.jpg");

  // add splashes
  for(int i=0; i < 8; i++)     // 8 splashes of flowers
    floorOG.addChild( new SplashShape(flowersTex, heights) );

  for (int i=0; i < 3; i++)    // 3 pools of water
    floorOG.addChild( new SplashShape(waterTex, heights) );

  // add all the meshes to the scene
  sceneBG.addChild( floorOG );
}  // end of addFloor()
```

The two files named in the MultiFloor constructor hold the grass and stone textures. Their contents are shown in Figures 4 and 5.

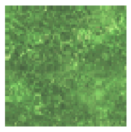
Figure 4. The Grass Texture.

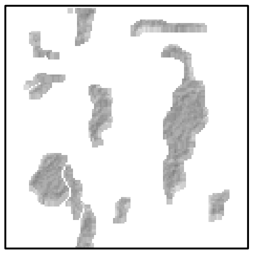
Figure 5. The Stones Texture.

The stones texture has a transparent background, which allows the grass to show through when the textures are combined by MultiFloor.

The constructor's numeric arguments (4 and 2) are the number of times the textures should be repeated along the sides of the floor. There are some restrictions on what values can be used, which I'll explain when I discuss the MultiFloor class.

The flowers and water textures used by the splash shapes are loaded in WrapTexLand3D rather than individually by each SplashShape object, to avoid repeated calls to Java 3D's texture loader. The textures are shown in Figures 6 and 7.

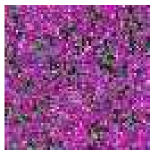
Figure 6. The Flowers Texture.

Figure 7. The Water Texture.

The MultiFloor and SplashShape objects are added to an OrderedGroup node so they'll be rendered in a fixed order. This avoids problems with drawing overlapping transparent geometries.

2.2. Start the Balls Moving

movingBalls() places balls randomly on the floor, and the TimeBehavior object keeps triggering updates to their positions.

```
// global
private static final int NUM_BALLS = 10;    // no. of moving balls
private static final int UPDATE_TIME = 100; // ms, for updating balls

private void movingBalls()
{
  ArrayList<MovingBall> mBalls = new ArrayList<MovingBall>();

  Texture2D ballTex = loadTexture("images/spot.gif");
  MovingBall mb;
  for (int i=0; i < NUM_BALLS; i++) {
    mb = new MovingBall(ballTex, heights);
    sceneBG.addChild( mb.getTG() );
    mBalls.add(mb);
  }

  // pass the list of balls to the behavior
  TimeBehavior ballTimer = new TimeBehavior(UPDATE_TIME, mBalls);
  ballTimer.setSchedulingBounds( bounds );
  sceneBG.addChild( ballTimer );
}  // end of movingBalls()
```

TimeBehavior cycles through the MovingBalls ArrayList (mBalls) every UPDATE_TIME (100) ms, calling each ball's updatePosition() method.

2.3. Moving Around the Scene

The view branch part of the scene graph isn't shown in Figure 3, since it's automatically created by Java 3D's SimpleUniverse class. WrapTexLand3D adds a

Java 3D OrbitBehavior node to it so the user can move/rotate/zoom the camera through the scene. The behavior is set up with orbitControls() in the same way as in the Checkers3D example in chapter 15.

3. The Multitextured Floor

MultiFloor is a subclass of Shape3D which builds the scene graph branch shown in Figure 8. This corresponds to the "MultiFloor" box shown in Figure 3.

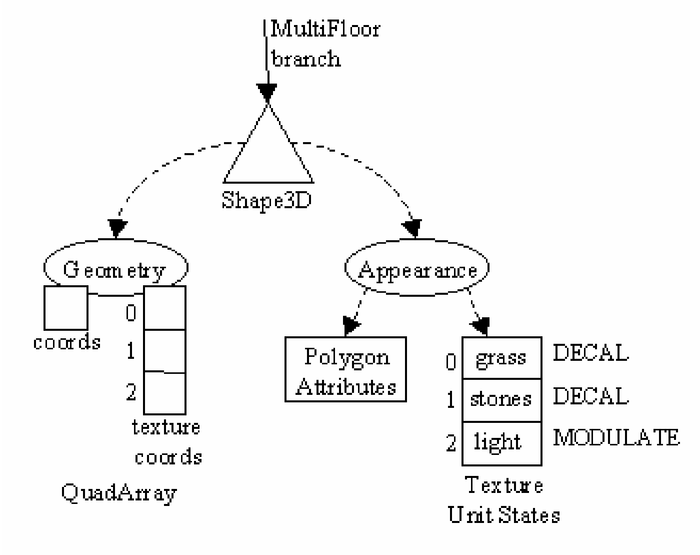

Figure 8. The MultiFloor Branch.

MultiFloor also generates a 2D heights map which it uses to populate the coordinates (coords) used in the geometry node. The heights map holds heights for a FLOOR_LEN+1 by FLOOR_LEN+1 grid of points, with each point one unit apart. FLOOR_LEN must be an even number, for reasons explained below.

The heights map is defined as:

```
private final static int FLOOR_LEN = 20;
private float[][] heights = new float[FLOOR_LEN+1][FLOOR_LEN+1];
```

Figure 9 shows the heights map (FL stands for FLOOR_LEN).

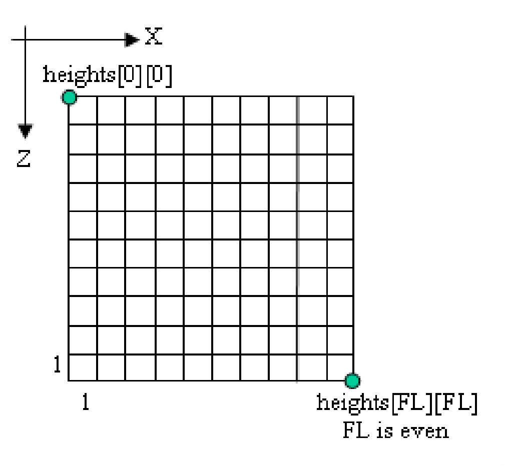

Figure 9. The Floor's Heights Map.

MultiFloor treats heights[z][x] as a height for a (x,z) coordinate. It's then an easy step to convert the heights array into the coordinates mesh shown in Figure 10. The height in heights[z][x] cell becomes the coordinate (x – FLOOR_LEN/2, height, z – FLOOR_LEN/2).

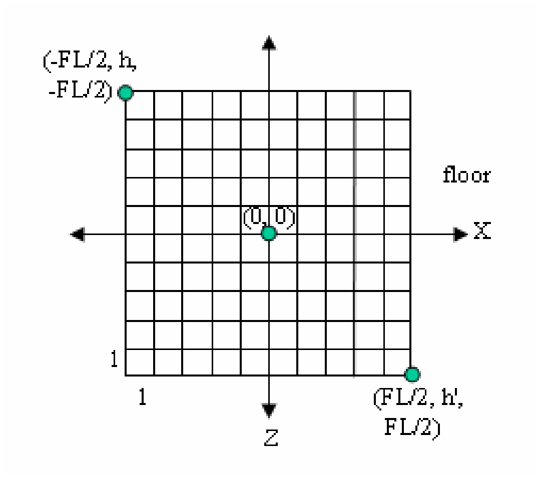

Figure 10. The Coordinates Mesh.

The coordinates mesh is used to build a QuadArray geometry of sides FLOOR_LEN, centered at (0,0) on the XZ plane. The requirement that the mesh is centered means that FLOOR_LEN must be even. The vertices are spaced out at 1 unit intervals along the x- and z- axes.

The geometry node shown in Figure 8 contains three sets of texture coordinates, one each for the grass, stone, and light map textures. The separate coordinates allows the textures to be repeated over the mesh at different frequencies.

The textures are stored in the shape's Appearance node, inside a 3-element Java 3D TextureUnitState array.

The constructor for MultiFloor:

```
// globals
private final static int FLOOR_LEN = 20;

private float[][] heights;   // heights map for the floor


public MultiFloor(String texFnm1, int freq1,
                        String texFnm2, int freq2)
{
  System.out.println("floor len: " + FLOOR_LEN);
  int texLen1 = calcTextureLength(texFnm1, freq1);
  int texLen2 = calcTextureLength(texFnm2, freq2);

  heights = makeHills();
  createGeometry(texLen1, texLen2);
  createAppearance(texFnm1, texFnm2);
}
```

The texture repetition frequencies are passed to calcTextureLength() calls. calcTextureLength() works out the required length of the texture so it will repeat that number of times across the floor. Texture length is defined by:

texture length * frequency == FLOOR_LEN

```
private int calcTextureLength(String texFnm, int freq)
{
  if (freq < 1)   // if 0 or negative
    freq = 1;
  else if (freq > FLOOR_LEN)  // if bigger than the floor length
    freq = FLOOR_LEN;

  return (FLOOR_LEN / freq);   // integer division
}
```

Due to calcTextureLength()'s use of integer division, the freq value should be a factor of the floor length. Since FLOOR_LEN 20, freq should be 1, 2, 4, 5, 10, or 20.

Figures 11 and 12 show the rendering of a "r.gif" texture containing a single letter "R". Figure 11 shows "r.gif' rendered with a frequency of 20, while Figure 12 uses a frequency of 4.

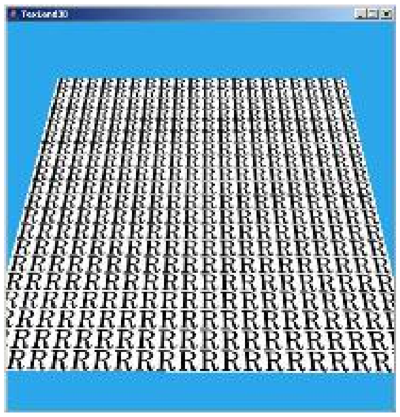
Figure 11. "R" (freq = 20).

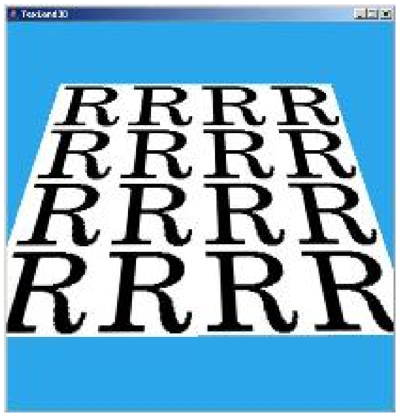
Figure 12. "R" (freq = 4).

3.1. Heights Map Generation

makeHills() utilizes a hill-raising algorithm to produces the heights map. A cell is chosen at random inside the heights[][] array and it's height is increased by a small amount (PEAK_INCR). The heights of its four neighbours on the left, right, back, and front are also increased, but by a smaller number (SIDE_INCR).

```
// hill creation constants for the heights map
private static final int NUM_HILL_INCRS = 2000;
private static final float PEAK_INCR = 0.3f;   // height incrs
private static final float SIDE_INCR = 0.25f;


private float[][] makeHills()
{
  float[][] heights = new float[FLOOR_LEN+1][FLOOR_LEN+1];
       // include heights on the front and right edges of the floor

  Random rand = new Random();

  int x, z;   // index into array (x == column, z == row)
  for (int i=0; i < NUM_HILL_INCRS; i++) {
    x = (int)(rand.nextDouble()*FLOOR_LEN);
    z = (int)(rand.nextDouble()*FLOOR_LEN);
    if (addHill(x, z, rand)) {
      heights[z][x] += PEAK_INCR;
      if (x > 0)
        heights[z][x-1] += SIDE_INCR;   // left
      if (x < (FLOOR_LEN-1))
        heights[z][x+1] += SIDE_INCR;   // right
      if (z > 0)
        heights[z-1][x] += SIDE_INCR;   // back
      if (z < (FLOOR_LEN-1))
        heights[z+1][x] += SIDE_INCR;   // front
    }
  }
  return heights;
}  // end of makeHills()
```

The addHill() test in makeHills() adds a twist, by assigning probabilities to whether the heights at particular locations are increased.

In the TexLand3D landscape, I want more hills along the left side, right side, and back of the floor. This distribution is represented graphical by the *probability map* in Figure 13. The map specifies the likelihood that a given height in the heights map will be increased when selected to become a hill.

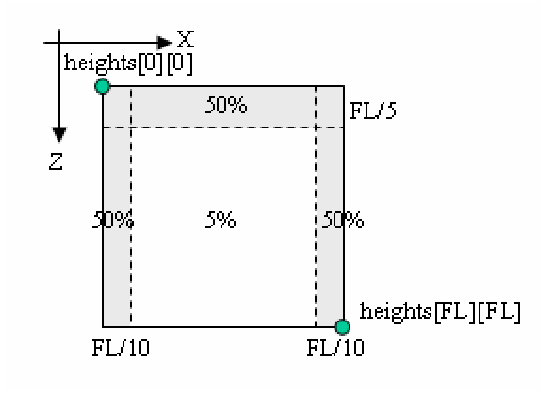

Figure 13. The Probability Map for the Heights Map.

If a height cell is chosen on the left, right, or back of the heights map, then there's a 50% chance that it will trigger hill raising, while the central region only has a 5% probability of doing so.

addHill() implements the map as a series of tests of the selected heights[][] array's x- and z- cell indices, and returns true if a hill should be raised at that position.

```
private boolean addHill(int x, int z, Random rand)
{
  if (z < FLOOR_LEN/5.0f)  //towards back of floor
    return (rand.nextDouble() < 0.5);  // 50% chance of a hill

  if (x < FLOOR_LEN/10.0f)  // on the left side of the floor
    return (rand.nextDouble() < 0.5);   // 50% chance

  if ((FLOOR_LEN - x) < FLOOR_LEN/10.0f) // on the right side
    return (rand.nextDouble() < 0.5);   // 50% chance

  // other areas of the floor
  return (rand.nextDouble() < 0.05);  // 5% chance
}
```

3.2. The Floor's Geometry

As shown in Figure 8, the Geometry node is a quad array with one set of vertices and three sets of texture coordinate (one for each texture).

```
private void createGeometry(int texLen1, int texLen2)
{
  Point3f[] coords = createCoords();   // create vertices

  // create (s,t) coordinates for the three textures
  TexCoord2f[] tCoords1 = createTexCoords(coords, texLen1);
  TexCoord2f[] tCoords2 = createTexCoords(coords, texLen2);
  TexCoord2f[] tCoords3 = createTexCoords(coords, FLOOR_LEN);

  QuadArray plane = new QuadArray(coords.length,
                         GeometryArray.COORDINATES |
                         GeometryArray.TEXTURE_COORDINATE_2,
                          3, new int[]{0,1,2});

  plane.setCoordinates(0, coords);
  plane.setTextureCoordinates(0, 0, tCoords1);  // grass texture
  plane.setTextureCoordinates(1, 0, tCoords2);  // stones texture
  plane.setTextureCoordinates(2, 0, tCoords3);  // light map
  setGeometry(plane);
}  // end of createGeometry()
```

The third call to createTexCoords() is supplied with a texture length equal to FLOOR_LEN, which means that the a single copy of the texture (the light map) will span the mesh.

The QuadArray() constructor doesn't include a GeometryArray.NORMALS flag since the mesh's reflective qualities will be 'faked' with the light map. However, the constructor has two arguments to specify the number of texture units it will utilize (3), and their indices. These indices are used in the calls to QuadArray.setTextureCoordinates(), to assign the texture coordinates to the right texture units.

Comparisons with the FractalLand3D Floor

The FractalLand3D ground in chapter 26 is built using a QuadArray-filled GeometryInfo object rather than with a plain QuadArray. This allows the floor's normals to be generated automatically (with Java 3D's NormalGenerator), and for the geometry to be converted to more efficient triangle strips with Java 3D's Stripifier.

Although I don't need normals in TexLand3D, stripification would be a good optimization for the floor. The revised version of createGeometry() using GeometryInfo is:

```
private void createGeometry(int texLen1, int texLen2)
// GeometryInfo and Stripifier version (not used in TexLand3D)
{
  // create (x,y,z) coordinates
  Point3f[] coords = createCoords();

  // create (s,t) coordinates for the three textures
  TexCoord2f[] tCoords1 = createTexCoords(coords, texLen1);
  TexCoord2f[] tCoords2 = createTexCoords(coords, texLen2);
  TexCoord2f[] tCoords3 = createTexCoords(coords, FLOOR_LEN);

  // create GeometryInfo
  GeometryInfo gi = new GeometryInfo(GeometryInfo.QUAD_ARRAY);
  gi.setCoordinates(coords);
  gi.setTextureCoordinateParams(3, 2); // 3 sets of 2D texels
  gi.setTexCoordSetMap( new int[]{0, 1, 2} );
  gi.setTextureCoordinates(0, tCoords1);   // for first texture
  gi.setTextureCoordinates(1, tCoords2);   // second texture
  gi.setTextureCoordinates(2, tCoords3);   // light map

  // stripifier to use triangle strips
  Stripifier st = new Stripifier();
  st.stripify(gi);

  // extract and use GeometryArray
  setGeometry( gi.getGeometryArray() );
}  // end of createGeometry()
```

This version of createGeometry() is *not* used in TexLand3D, due to the coding of the SplashShape class. Each SplashShape object selects a small set of heights from the floor's heights map, and builds it own copy of the floor in a QuadArray. The splash shapes are drawn at the same places as the original floor elements, but the ordered grouping of the floor and the SplashShapes means that the splash shapes are rendered after the floor.

The main reason for using GeometryInfo is to stripify the floor's QuadArray into triangle strips. This leaves the coordinates unchanged, but the floor mesh's faces will be different. The unfortunate consequence is that it's quite likely that some of the floor's triangles will be positioned above parts of the splash shapes.

The solution is to use a single mesh type for the floor and the splash shapes. I chose QuadArrays since its easier to explain how texturing is applied to a grid of quadrilaterals.

Creating the Floor's Vertices

createCoords() generates the floor's (x,y,z) coordinates by utilizing the heights map. As shown in Figures 9 and 10, the heights[][] array is treated as a (x,z) grid of heights, and translated into a coordinates mesh starting at (-FLOOR_LEN/2, -FLOOR_LEN/2).

```
private Point3f[] createCoords()
{
  Point3f[] coords = new Point3f[FLOOR_LEN*FLOOR_LEN*4];
              // since each quad has 4 coords
  int i = 0;
  for(int z=0; z <= FLOOR_LEN-1; z++) {    // skip z's front row
    for(int x=0; x <= FLOOR_LEN-1; x++) {  // skip x's right column
      createTile(coords, i, x, z);
      i = i + 4;  // since 4 coords created for 1 tile
    }
  }
  return coords;
}
```

createTile() initializes four coordinates that will later form a quad (tile) in the QuadArray mesh.

```
private void createTile(Point3f[] coords, int i, int x, int z)
{
  // (xc, zc) is the (x,z) coordinate in the scene
  float xc = x - FLOOR_LEN/2;
  float zc = z - FLOOR_LEN/2;

  // points created in counter-clockwise order from bottom left
  coords[i] = new Point3f(xc, heights[z+1][x], zc+1.0f);
  coords[i+1] = new Point3f(xc+1.0f, heights[z+1][x+1], zc+1.0f);
  coords[i+2] = new Point3f(xc+1.0f, heights[z][x+1], zc);
  coords[i+3] = new Point3f(xc, heights[z][x], zc);
}  // end of createTile()
```

The x and z indices of the heights[][] array are converted into x- and z- axis values by subtracting FLOOR_LEN/2 from them.

Building the Floor's Texture Coordinates

createTexCoords() generates texture coordinates for the floor's vertices. Since the vertices are grouped in four's for each quad, the texture coordinates are grouped in a similar way.

```
private TexCoord2f[] createTexCoords(Point3f[] coords, int texLen)
{
  int numPoints = coords.length;
  TexCoord2f[] tcoords = new TexCoord2f[numPoints];

  TexCoord2f dummyTC = new TexCoord2f(-1,-1);   // dummy tex coord
  for(int i=0; i < numPoints; i=i+4)
    createTexTile(tcoords, i, texLen, coords, dummyTC);
                       // 4 tex coords for 1 coordinates tile
  return tcoords;
}  // end of createTexCoords()
```

The 'dummy' texture coordinate, dummyTC, is used to deal with some tricky edge cases when generating the texels, which I'll explain in a moment.

In coords[], each group of four vertices for a quad is stored in counter-clockwise order, starting from the bottom left vertex. This same ordering is employed by createTexTile() for storing the texture coordinates in tcoords[].

```
private void createTexTile(TexCoord2f[] tcoords, int i, int texLen,
                              Point3f[] coords, TexCoord2f dummyTC)
{
  // make the bottom-left tex coord, i
  tcoords[i] = makeTexCoord(coords[i], texLen, dummyTC);

  for (int j = 1; j < 4; j++)   // add the other three coords
    tcoords[i+j] = makeTexCoord(coords[i+j], texLen, tcoords[i]);
}  // end of createTexTile()
```

makeTexCoord() converts a (x,y,z) coordinate into a (s,t) texel based on the (x,z) value modulo the texture length, and divided by that length. Unfortunately this causes some problems for the texels at the top and right edges of the texture.

```
private TexCoord2f makeTexCoord(Point3f coord, int texLen,
                                          TexCoord2f firstTC)
{
  float s, t;
  if (texLen > 1) {
    s = ((float)((coord.x + FLOOR_LEN/2) % texLen))/texLen;
    t = ((float)((FLOOR_LEN/2 - coord.z) % texLen))/texLen;
  }
  else {   // don't use modulo when texLen == 1
    s = ((float)(coord.x + FLOOR_LEN/2))/texLen;
    t = ((float)(FLOOR_LEN/2 - coord.z))/texLen;
  }
  if (s < firstTC.x)   // deal with right edge rounding
    s = 1.0f - s;
  if (t < firstTC.y)   // deal with top edge rounding
    t = 1.0f - t;

  return new TexCoord2f(s, t);
}  // end of makeTexCoord
```

The problem is illustrated by Figure 14, which shows how a "R" texture is a mapped to the vertices covering 5*5 floor quads.

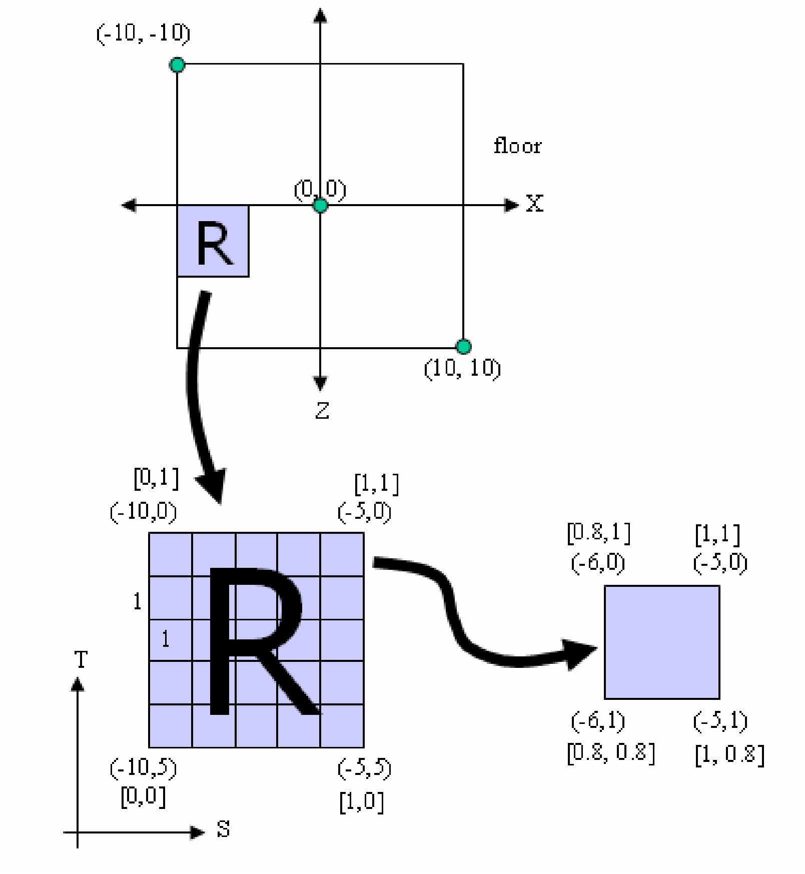

Figure 14. Correctly Texturing the "R".

The modulo-related problems occur for mappings involving quads at the top and right edges of the texture. The single quad on the right of Figure 14 shows the correct texel mapping for the (x,z) coordinates (-6,1), (-5,1), (-5,0), and (-6,0). Unfortunately, makeTexCoord()'s use of modulo incorrectly maps the three coordinates, (-5,1), (-5,0), and (-6,0), to [**0**,0.8], [**0**,**0**], and [0.8,**0**] respectively. When the s or t value should be a 1, the modulo operation produces a 0.

makeTexCoord() handles this by comparing the generated texel's s and t values with the bottom left texel of the quad ([0.8,0.8] in the example). If the generated s or t values are smaller (i.e. less than 0.8), then the modulo problem has occurred. It's fixed by subtracting the offending s or t value from 1, changing the 0 to a 1.

This extra test is why createTexTile() passes the tcoords[i] texel into the makeTexCoords() calls for its three neighbours.

3.3. The Floor's Appearance

createAppearance() builds the Appearance node shown in Figure 8.

```
private void createAppearance(String texFnm1, String texFnm2)
{
  Appearance app = new Appearance();

  // switch off face culling
  PolygonAttributes pa = new PolygonAttributes();
  pa.setCullFace(PolygonAttributes.CULL_NONE);
  app.setPolygonAttributes(pa);

  // texture units
  TextureUnitState tus[] = new TextureUnitState[3];

  // cover the floor with the first texture
  tus[0] = loadTextureUnit(texFnm1, TextureAttributes.DECAL);

  // add second texture (it has transparent parts)
  tus[1] = loadTextureUnit(texFnm2, TextureAttributes.DECAL);

  // modulated light map
  tus[2] = loadTextureUnit("light.gif", TextureAttributes.MODULATE);
  // tus[2] = lightMapTUS();

  app.setTextureUnitState(tus);

  setAppearance(app);
}  // end of createAppearance()
```

createAppearance() loads two ground textures (the grass and stones) and combines them as decals. The second texture (the stones) has transparent parts so both textures can be seen at runtime. The third texture, the light map, is modulated with the others so they all remain visible. The light map can either be loaded from a file, or be drawn at runtime. The drawing method, lightMapTUS() is commented out in the code above.

The light map image in light.gif is shown in Figure 15.

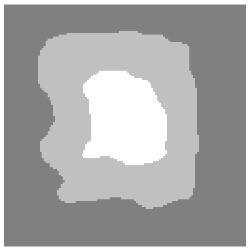

Figure 15. The Light Map in light.gif.

Figure 16 shows the floor after the textures have been combined. The view is from above, with hill-generation switched off so the texturing is easier to see.

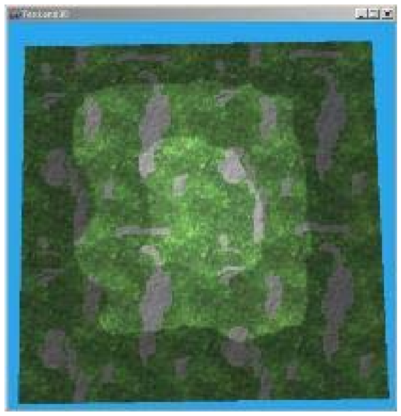
Figure 16. The Floor From Above.

The light map modulation means that the map's darker areas add 'shadows' to the ground.

Although light.gif only uses grayscales, it's possible to include colours in the map, which will tint the floor.

Making a Texture Unit

loadTextureUnit() creates a texture unit by combining a texture and texture attributes.

```
private TextureUnitState loadTextureUnit(String fnm, int texAttr)
{
  TextureLoader loader = new TextureLoader("images/"+fnm,
                             TextureLoader.GENERATE_MIPMAP, null);
  System.out.println("Loaded floor texture: images/" + fnm);

  Texture2D tex = (Texture2D) loader.getTexture();
  tex.setMinFilter(Texture2D.MULTI_LEVEL_LINEAR);

  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode(texAttr);

  TextureUnitState tus =  new TextureUnitState(tex, ta, null);
  return tus;
}  // end of loadTextureUnit()
```

Mipmaps are generated for the texture, so it'll look better at various distances from the camera.

Graphics Hardware Concerns

The processing required to render three texture units is handled by the graphics card. Older cards may not be able to deal with large numbers of texture units, so your code should check that the hardware is sufficient. This is done by calling

Canvas3D.queryProperties(), and checking the returned HashMap of graphics capabilities. The maximum number of supported texture units can be retrieved with the "textureUnitStateMax" key.

reportTextureUnitInfo() shows how queryProperties() is used. This method is located in WrapTexLand3D, where the Canvas3D object is created.

```
private void reportTextureUnitInfo(Canvas3D c3d)
/* Report the number of texture units supported by the machine's
   graphics card. Called in WrapTexLand3D. */
{
  Map c3dMap = c3d.queryProperties();

  if (!c3dMap.containsKey("textureUnitStateMax"))
    System.out.println("Texture unit state maximum not found");
  else {
    int max =((Integer)c3dMap.get("textureUnitStateMax")).intValue();
    System.out.println("Texture unit state maximum: " + max);
  }
}
```

The graphic card's behaviour when the texture unit maximum is exceeded is hardware dependent, but the extra textures will probably be rendered with TextureAttributes.REPLACE. For instance, if the maximum is 2, then the light map will replace the ground textures, so only the map image is visible.

I haven't bothered using the texture unit maximum information in TexLand3D since most modern cards support at least 4 texture units, and TexLand3D only needs a maximum of three for a shape. I wouldn't advise this devil-may-care attitude for real games.

Creating a Light Map at Runtime

A static light map loaded from a file is sufficient for most applications, but sometimes it's preferable to generate lighting effects at run time.

The map's image is drawn using Java 2D, converted to a texture, then added to a TextureUnitState object, along with suitable texture attributes. lightMapTUS() builds the texture unit, and delegates the drawing work to createLightMap().

```
private TextureUnitState lightMapTUS()
{
  Texture2D tex = createLightMap(); // draw light map

  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode(TextureAttributes.MODULATE); // for light

  TextureUnitState tus =  new TextureUnitState(tex, ta, null);
  return tus;
}
```

The light map texture is made from a RGB BufferedImage. Java 2D operations are utilized by manipulating the image through a Graphics2D object.

To keep the example short, createLightMap() draws a series of concentric circles, filled with various shades of grey.

```
// globals
private static final int LIGHT_MAP_SIZE = 128;
   // the light map is a texture, so it's size should be a power of 2


private Texture2D createLightMap()
{
  BufferedImage img = new BufferedImage(LIGHT_MAP_SIZE,
                 LIGHT_MAP_SIZE, BufferedImage.TYPE_INT_RGB);
  Graphics2D g = img.createGraphics();

  g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                     RenderingHints.VALUE_ANTIALIAS_ON);

  g.setColor(Color.gray);
  g.fillRect(0, 0, LIGHT_MAP_SIZE, LIGHT_MAP_SIZE);

  int xCenter = LIGHT_MAP_SIZE/2;
  int yCenter = LIGHT_MAP_SIZE/2;

  int diam = LIGHT_MAP_SIZE*7/10;   // 70% of light map size
  g.setColor(Color.lightGray);
  g.fillOval(xCenter-diam/2, yCenter-diam/2, diam, diam);

  diam = LIGHT_MAP_SIZE*4/10;    // 40%
  g.setColor(Color.white);
  g.fillOval(xCenter-diam/2, yCenter-diam/2, diam, diam);

  diam = LIGHT_MAP_SIZE*2/10;    // 20%
  g.setColor(Color.lightGray);
  g.fillOval(xCenter-diam/2, yCenter-diam/2, diam, diam);

  diam = LIGHT_MAP_SIZE*15/100;  // 15%
  g.setColor(Color.gray);
  g.fillOval(xCenter-diam/2, yCenter-diam/2, diam, diam);

  g.dispose();

  // convert the buffered image into a texture
  ImageComponent2D grayImage =
             new ImageComponent2D(ImageComponent.FORMAT_RGB, img);
  Texture2D lightTex = new Texture2D(Texture.BASE_LEVEL, Texture.RGB,
                              LIGHT_MAP_SIZE, LIGHT_MAP_SIZE);
  lightTex.setImage(0, grayImage);

  return lightTex;
} // end of createLightMap()
```

After the BufferedImage has been filled, it's used to initialize a Texture2D object.

Figure 17 shows the floor with the runtime light map, viewed from above. I've switched off hill-generation to make the map easier to see.

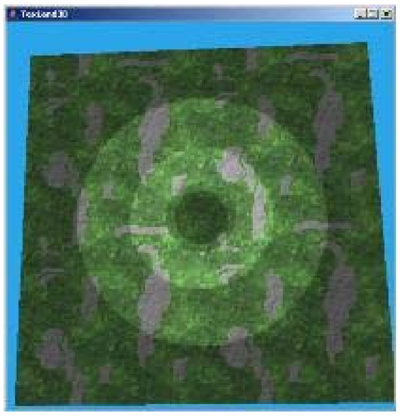
Figure 17. The Runtime Light Map.

4. The Splash Shape

The SplashShape class combines an *alpha mask* with a texture to make it's boundary irregular in a random way.

An alpha mask is a texture that contains only an alpha channel (i.e. only transparency information). It can be implemented as a full RGBA image with its colour channels ignored at runtime, or as a grayscale image with a single channel, which is interpreted as the alpha channel (as I do here).

The splash shape's geometry comes from the floor. A fragment of the floor's heights map is copied to make a mesh that matches that portion of the floor.

The SplashShape constructor divides up the tasks of geometry and appearance building.

```
// global
private int floorLen;  // length of floor's side


public SplashShape(Texture2D tex, float[][] heights)
{
  floorLen = heights.length-1;
  createGeometry(heights);
  makeAppearance(tex);
}
```

The result is the scene graph branch shown in Figure 18.

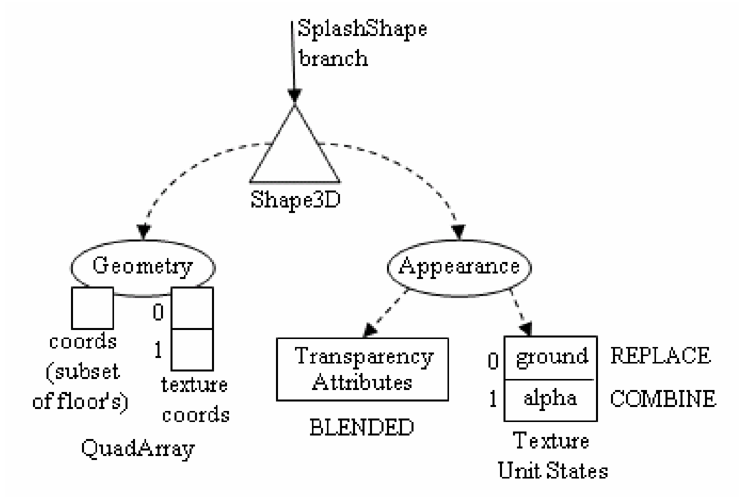

Figure 18. Scene Graph Branch for SplashShape.

Copies of this branch appear in place of the "SplashShape" boxes in Figure 3.

4.1. The SplashShape's Geometry

The splash shape's geometry is a subsection of the floor's geometry, starting at a random coordinate, and extending several vertices along the x- and z- axes.

The geometry uses a QuadArray like the floor, and two sets of texture coordinates, one for the ground texture (flowers, water), the other for the alpha mask. The geometry appears on the left side of Figure 18.

```
private void createGeometry(float[][] heights)
{
  // get a starting point for the splash
  Point3i startPt = getStartPt(heights);

  // create coords from a section of heights map starting at startPt
  Point3f[] coords = createCoords(startPt, heights);

  // create texture coordinates for the splash shape
  TexCoord2f[] tCoords = createTexCoords(startPt, coords);

  // make a mesh with two texture units
  QuadArray plane = new QuadArray(coords.length,
                        GeometryArray.COORDINATES |
                        GeometryArray.TEXTURE_COORDINATE_2,
                        2, new int[]{0,1});

  plane.setCoordinates(0, coords);
  plane.setTextureCoordinates(0, 0, tCoords);   // for detail texture
  plane.setTextureCoordinates(1, 0, tCoords);   // alpha mask texture
  setGeometry(plane);
}  // end of createGeometry()
```

getStartPt() obtains a starting point for the splash by randomly generated row and column indices for the floor's heights map. These are easily converted to x- and z-coordinates when needed.

```
// global
private static final int SPLASH_SIZE = 3;
   /* SplashShape copies SPLASH_SIZE*SPLASH_SIZE quads from
      the floor mesh. SPLASH_SIZE should be less than the
      floor length (which is stored in floorLen). */


private Point3i getStartPt(float[][] heights)
{
  Random rand = new Random();
  int z = (int)(rand.nextDouble()*floorLen);   // z is the row index
  int x = (int)(rand.nextDouble()*floorLen);   // x is column index

  if (z+SPLASH_SIZE > floorLen)  // if splash extends off front edge
    z = floorLen - SPLASH_SIZE;  // move back

  if (x+SPLASH_SIZE > floorLen)  // if splash extends off right edge
    x = floorLen - SPLASH_SIZE;  // move left

  return new Point3i(x, 0, z);   // don't use the y-value
}
```

The selected values are tested to see if the splash shape, which is SPLASH_SIZE units long in the x- and z- directions, extends off the map. If it does, then the start point is moved back and/or to the left.

The x- and z- indices are returned in a Point3i object. This is a convenient way of packaging up the integers; the point's y-value isn't used.

The SplashShape's Vertices

Armed with a start point and the floor's heights map, createCoords() creates a heights map for the splash shape. The map is utilized to generate the geometry's vertices, which are ordered in the same way as the floor's vertices, in groups of four, representing groups of quads.

```
private Point3f[] createCoords(Point3i startPt, float[][] heights)
{
  // copy heights from the floor's heights map
  float[][] splashHeights = getSplashHeights(startPt, heights);

  Point3f[] coords = new Point3f[SPLASH_SIZE*SPLASH_SIZE*4];
                   // since each quad in the mesh has 4 coords
  int i = 0;
  for(int z=0; z <= SPLASH_SIZE-1; z++) {    // skip z's front row
    for(int x=0; x <= SPLASH_SIZE-1; x++) {  // skip x's right column
      createTile(coords, i, x, z, startPt, splashHeights);
      i = i + 4;  // since 4 coords are needed for 1 quad
    }
  }
  return coords;
}  // end of createCoords()
```

The splash heights map is produced by traversing the floor's heights map, starting from the splash's start point, and copying over a square of heights SPLASH_SIZE wide by SPLASH_SIZE long.

```
private float[][] getSplashHeights(Point3i startPt,
                                          float[][] heights)
{
  float[][] splashHeights = new float[SPLASH_SIZE+1][SPLASH_SIZE+1];
             // "+1" to include front and right edges

  for(int z=0; z <= SPLASH_SIZE; z++)
    for(int x=0; x <= SPLASH_SIZE; x++)
      splashHeights[z][x] = heights[startPt.z + z][startPt.x + x];

  return splashHeights;
}
```

createTile() creates coordinates for a single quad, with its top left hand corner at (xc,zc), which is calculated by converting the splashHeights[][] indices into x- and z-coordinates.

```
private void createTile(Point3f[] coords, int i, int x, int z,
                          Point3i startPt, float[][] splashHeights)
{
  // (xc,zc) is the (x,z) coordinate in the floor mesh
  float xc = startPt.x + x - floorLen/2;
  float zc = startPt.z + z - floorLen/2;

  // points created in counter-clockwise order from bottom left
  coords[i] = new Point3f(xc, splashHeights[z+1][x], zc+1.0f);
  coords[i+1] = new Point3f(xc+1.0f,splashHeights[z+1][x+1],zc+1.0f);
  coords[i+2] = new Point3f(xc+1.0f, splashHeights[z][x+1], zc);
  coords[i+3] = new Point3f(xc, splashHeights[z][x], zc);
}
```

The SplashShape's Texture Coordinates

The texture coordinates generation for the splash shape closely mirrors the code used in MultiFloor for generating the texture coordinates for the floor. The code is actually a bit simpler since the splash shape texture always covers the mesh, without repeating.

```
private TexCoord2f[] createTexCoords(Point3i startPt,
                                          Point3f[] coords)
{
  int numPoints = coords.length;
  TexCoord2f[] tcoords = new TexCoord2f[numPoints];

  for(int i=0; i < numPoints; i=i+4)  // 4 tex coords for 1 quad
    for (int j = 0; j < 4; j++)
      tcoords[i+j] = makeTexCoord(startPt, coords[i+j]);
  return tcoords;
}
```

The splash shape's vertices are grouped into four's for each quad in the mesh, and so the texture coordinates are similarly grouped. The points are stored in counter-clockwise order, starting from the bottom left vertex.

The (x,z) parts of a vertex is converted to values between 0 and SPLASH_SIZE, and dividing by SPLASH_SIZE so they fall in the texel 0-1 range. The t value is adjusted so it increases as z decreases.

```
private TexCoord2f makeTexCoord(Point3i startPt, Point3f coord)
{
  float s = ((float)(coord.x - startPt.x + floorLen/2))/SPLASH_SIZE;
  float t = 1.0f -
           ((float)(coord.z - startPt.z + floorLen/2))/SPLASH_SIZE;
  return new TexCoord2f(s, t);
}
```

4.2. The Splash Shape's Appearance

makeAppearance() builds the right half of the scene branch shown in Figure 18.

```
private void makeAppearance(Texture2D tex)
{
  Appearance app = new Appearance();

  // the ground's texture unit
  TextureUnitState groundTUS = makeGround(tex);

  // the alpha texture unit
  TextureUnitState alphaTUS = new TextureUnitState();
  alphaTUS.setTextureAttributes( getAlphaTA() );
  alphaTUS.setTexture( getAlphaTexture() );

  // put the two texture units together
  TextureUnitState[] tus = new TextureUnitState[2];
  tus[0] = groundTUS;
  tus[1] = alphaTUS;
  app.setTextureUnitState(tus);

  // switch on transparency, and use blending
  TransparencyAttributes ta = new TransparencyAttributes();
  ta.setTransparencyMode(TransparencyAttributes.BLENDED);
  app.setTransparencyAttributes(ta);

  setAppearance(app);
}  // end of makeAppearance()
```

Setting the transparency attribute is essential otherwise the transparent parts of the alpha mask will be ignored when the textures are rendered.

Making the Ground

makeGround() loads the ground texture (either flowers or water) and its attributes into a texture unit.

```
private TextureUnitState makeGround(Texture2D tex)
{
  TextureAttributes groundTAs = new TextureAttributes();
  groundTAs.setTextureMode(TextureAttributes.REPLACE);
  groundTAs.setPerspectiveCorrectionMode(TextureAttributes.NICEST);

  TextureUnitState groundTUS = new TextureUnitState();
  groundTUS.setTextureAttributes(groundTAs);
  groundTUS.setTexture(tex);

  return groundTUS;
} // end of makeGround()
```

The TextureAttribute.REPLACE mode means that the texture replaces any underlying material colour, and lighting is ignored.

Texture Attributes for the Alpha Mask

The texture attributes required for the alpha mask go beyond Java 3D's basic modes (e.g. DECAL, REPLACE, MODULATE) since it's necessary to differentiate between the alpha texture's RGBA channels.

Finer-grain channel control is available through the TextureAttributes.COMBINE mode, which permits separate texture attribute equations to be defined for the RGB and alpha channels.

```
private TextureAttributes getAlphaTA()
{
  TextureAttributes alphaTA = new TextureAttributes();

  alphaTA.setPerspectiveCorrectionMode(alphaTA.NICEST);
  alphaTA.setTextureMode(TextureAttributes.COMBINE);
        // COMBINE gives us control over the RGBA channels

  // use COMBINE_REPLACE to replace colour and alpha of the geometry
  alphaTA.setCombineRgbMode(TextureAttributes.COMBINE_REPLACE);
  alphaTA.setCombineAlphaMode(TextureAttributes.COMBINE_REPLACE);

  /* the source RGB == previous texture unit (i.e. the first unit),
     and the source alpha == this texture (i.e. the alpha). */
  alphaTA.setCombineRgbSource(0,
             TextureAttributes.COMBINE_PREVIOUS_TEXTURE_UNIT_STATE);
  alphaTA.setCombineAlphaSource(0,
             TextureAttributes.COMBINE_TEXTURE_COLOR);

  /* The combined texture gets its colour from the source RGB and
     its alpha from the source alpha. */
  alphaTA.setCombineRgbFunction(0,
                    TextureAttributes.COMBINE_SRC_COLOR);
  alphaTA.setCombineAlphaFunction(0,
                    TextureAttributes.COMBINE_SRC_ALPHA);

  return alphaTA;
}  // end of getAlphaTA()
```

The TextureAttribute class' setCombineXXXMode(), setCombineXXXSource() and setCombineXXXFunction() methods are utilized to specify two equations which say how the final colour of the shape is determined in terms of the ground and alpha textures. These equations can be written as:

$\text{Colour}_{\text{final/RGB}} = \text{Colour}_{\text{tus(0)/RGB}}$

$\text{Colour}_{\text{final/A}} = \text{Colour}_{\text{alpha/RGB}}$

The final geometry colour, $\text{Colour}_{\text{final}}$, is divided into equations for its RGB and alpha parts ($\text{Colour}_{\text{final/RGB}}$ and $\text{Colour}_{\text{final/A}}$). The RGB component comes from the first texture unit (tus(0)), while the alpha comes from the colour channels of the alpha texture.

The numerous combine methods in Java 3D's TextureAttributes class make it possible to define much more complex colour equations than the two shown here. For instance, an equation can utilize RGB and alpha channels from several sources, and combine or blend them in several ways. The TextureAttribute class documentation gives some examples, using a notation similar to mine.

A good way of gaining a better insight about colour equations in Java 3D is to study the similar mechanism in OpenGL (which Java 3D borrowed).

Drawing the Alpha Mask

The alpha mask texture is generated at runtime as a BufferedImage, in much the same way as the light mask, but the image *must* be a grayscale. Why?

A look back at the colour equations in the last section shows that the *alpha* component of the geometry's colour ($\text{Colour}_{\text{final/A}}$) comes from the *colour* part of the alpha mask ($\text{Colour}_{\text{alpha/RGB}}$). For this to work, the alpha mask's colour component must be a single channel, which means that the image must be a grayscale. Black will correspond to fully transparent, white to opaque, with degrees of translucency between.

It's also necessary to set the alpha texture's format to be Texture.ALPHA.

```
// global
private static final int ALPHA_SIZE = 64;


private Texture2D getAlphaTexture()
{
  /* the image component is defined to be a single 8-bit channel,
     which will hold a grayscale image created by alphaSplash() */
  ImageComponent2D alphaIC =
         new ImageComponent2D(ImageComponent2D.FORMAT_CHANNEL8,
                              ALPHA_SIZE, ALPHA_SIZE, true, false);

  alphaIC.set( alphaSplash() );  // generate a buffered image

  // convert the image into an alpha texture
  Texture2D tex = new Texture2D(Texture2D.BASE_LEVEL, Texture.ALPHA,
                                         ALPHA_SIZE, ALPHA_SIZE);
  tex.setMagFilter(Texture.BASE_LEVEL_LINEAR);
  tex.setMinFilter(Texture.BASE_LEVEL_LINEAR);
  tex.setImage(0, alphaIC);
  return tex;
} // end of getAlphaTexture()
```

alphaSplash() must create a grayscale BufferedImage of size ALPHA_SIZE by ALPHA_SIZE due to the formats specified for the ImageComponent2D and Texture2D objects in getAlphaTexture(). However, it can use any Java 2D drawing operations it likes.

My aim is to break up the sharp rectangular border of the ground texture by utilizing the alpha mask, and to break it up in a different way for each splash shape. This is achieved by alphaSplash() drawing a series of semi-randomly positioned circles with various levels of transparency. The more opaque circles tend to be in the center of the alpha mask.

```
private BufferedImage alphaSplash()
{
  Random rand = new Random();

  // create a grayscale buffered image
  BufferedImage img = new BufferedImage(ALPHA_SIZE, ALPHA_SIZE,
                                  BufferedImage.TYPE_BYTE_GRAY);
  Graphics2D g = img.createGraphics();

  // draw into it
  g.setColor(Color.black);   // fully transparent
  g.fillRect(0, 0, ALPHA_SIZE, ALPHA_SIZE);

  int radius = 3;  // circle radius
  int offset = 8;  // offset of boxed circle from top-left of graphic
  g.setColor(new Color(0.3f, 0.3f, 0.3f)); //near transparent circles
  boxedCircles(offset, offset, ALPHA_SIZE-(offset*2), radius,
                                                 100, g, rand);
  offset = 12;
  g.setColor(new Color(0.6f, 0.6f, 0.6f)); // mid-level translucent
  boxedCircles(offset, offset, ALPHA_SIZE-(offset*2), radius,
                                                 80, g, rand);
  offset = 16;
  g.setColor(Color.white);   // fully opaque circles
  boxedCircles(offset, offset, ALPHA_SIZE-(offset*2), radius,
                                                 50, g, rand);
  g.dispose();
  return img;
}  // end of alphaSplash()
```

boxedCircles() draws circles whose centers are somewhere within the square defined by its input arguments.

```
private void boxedCircles(int x, int y, int len, int radius,
                        int numCircles, Graphics2D g, Random rand)
/* Generate numCircles circles whose centers are within the square
   whose top-left is (x,y), with sides of len. A circle has a radius
   equal to the radius value.
*/
{
  int xc, yc;
  for (int i=0; i < numCircles; i++) {
    xc = x + (int)(rand.nextDouble()*len) - radius;
    yc = y + (int)(rand.nextDouble()*len) - radius;
    g.fillOval(xc, yc, radius*2, radius*2);
  }
}
```

Figure 19 shows some several splash shapes drawn against a plain white floor, viewed from above. Each shape has a different irregular border.

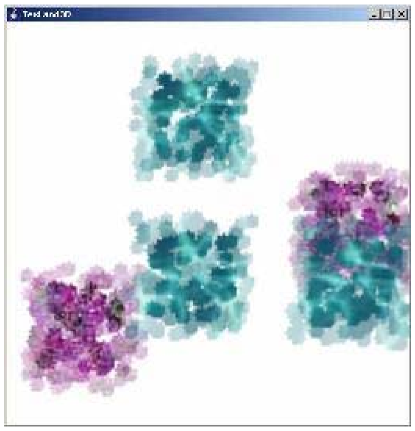

Figure 19. SplashShapes on a White Surface.

There's no absolute requirement that the alpha mask has to be generated at run time. It's quite possible to load a grayscale image from a file. For example:

```
// in getAlphaTexture()
alphaIC.set( loadAlpha("alpha.gif") );
```

loadAlpha() is:

```
private BufferedImage loadAlpha(String fnm)
{ try {
    return ImageIO.read( new File("images/" + fnm));
  }
  catch (IOException e) {
    System.out.println("Could not load alpha mask: images/" + fnm);
    return null;
  }
}
```

The file must use 8-bit grayscales, and be ALPHA_SIZE*ALPHA_SIZE large.

5. Moving Balls

Several textured, illuminated balls are randomly placed on the floor, and then meander about aimlessly. They rotate slightly as they move, and can't travel beyond the edges of the floor. When a ball encounters an edge, it turns away from it. Figure 20 shows a close-up of one of the balls.

Figure 20. A Moving Ball.

The MovingBall constructor calls makeBall() to build the scene graph branch shown in Figure 21. Copies of it correspond to the "MovingBall" boxes in Figure 3.

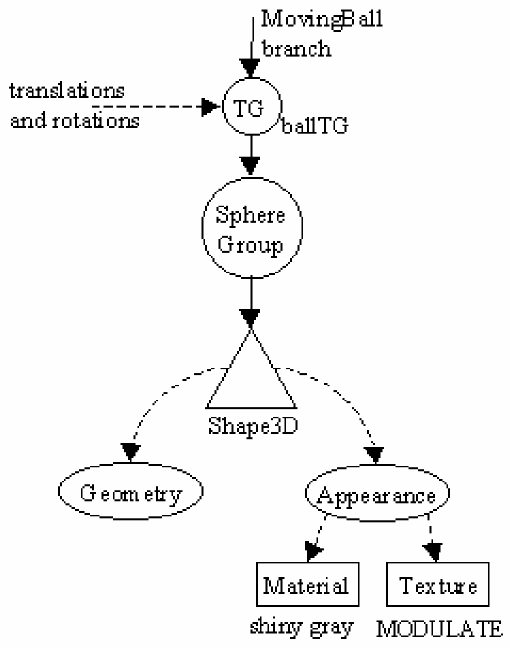

Figure 21. The Scene Graph Branch for a Moving Ball.

The translations and rotations applied to the ballTG TransformGroup are triggered by calls to MovingBall's updatePosition() method by TimeBehavior.

The makeBall() method:

```
// globals
private static final float RADIUS = 0.25f;   // of sphere

// material colours
private static final Color3f BLACK = new Color3f(0.0f, 0.0f, 0.0f);
private static final Color3f GRAY = new Color3f(0.6f, 0.6f, 0.6f);
private static final Color3f WHITE = new Color3f(0.9f, 0.9f, 0.9f);

private TransformGroup ballTG;    // TG which the ball hangs off

// reusable object for calculations
private Transform3D t3d;  // for manipulating ballTG's transform


private void makeBall(Texture2D ballTex)
{
  Appearance app = new Appearance();

  // combine texture with material and lighting of underlying surface
  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode( TextureAttributes.MODULATE );
  app.setTextureAttributes( ta );

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

  // apply texture to shape
  if (ballTex != null)
    app.setTexture(ballTex);

  // randomly position the ball on the floor
  t3d.set( genStartPosn());

  // ball's transform group can be changed (so it can move/rotate)
  ballTG = new TransformGroup(t3d);
  ballTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
  ballTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

  // the ball has normals for lighting, and texture support
  Sphere ball = new Sphere(RADIUS,
                        Sphere.GENERATE_NORMALS |
                        Sphere.GENERATE_TEXTURE_COORDS,
                        15, app);
  ballTG.addChild(ball);
 }  // end of makeBall()
```

5.1. Positioning the Ball

A random (x,z) starting position for the ball is calculated between -floorLen/2 and floorLen/2. floorLen is obtained from the floor's heights map, which is passed to MovingBall via its constructor.

```
// global
private int floorLen;        // length of floor sides
```

```
// reusable object for calculations
private Vector3f posnVec;     // for manipulating the ball's position
private Random rand;


private Vector3f genStartPosn()
{
  float x = (float)((rand.nextDouble() * floorLen)-floorLen/2);
  float z = (float)((rand.nextDouble() * floorLen)-floorLen/2);
  float y = getHeight(x, z) + RADIUS;  // ball rests on the floor
  posnVec.set(x,y,z);
  return posnVec;
}  // end of genStartPosn()
```

getHeight() uses the floor's heights map to calculate the height at the supplied (x, z) position. The ball will usually be located somewhere between the heights map vertices, so the height is calculated as a weighted average of the (x,z) coordinate's distance from the four nearest vertices.

```
// globals
private float[][] heights;   // the floor's heights map


private float getHeight(float x, float z)
// calculate a height for the ball's (x,z) location
{
  float xLoc = x + floorLen/2;  // (xLoc,zLoc) is on the heights map
  float zLoc = z + floorLen/2;

  // split (xLoc,zLoc) coordinate into integer and fractional parts
  int xH = (int) Math.floor(xLoc);
  float xFrac = xLoc - xH;

  int zH = (int) Math.floor(zLoc);
  float zFrac = zLoc - zH;

  /* the average height is based on the (xLoc,zLoc) coord's
     distance from the four surrounding heights in the
     heights[][] array. */
  float height = ((heights[zH][xH] * (1.0f-zFrac) * (1.0f-xFrac)) +
                  (heights[zH][xH+1] * (1.0f-zFrac) * xFrac) +
                  (heights[zH+1][xH] * zFrac * (1.0f-xFrac) ) +
                  (heights[zH+1][xH+1] * zFrac * xFrac) );
  return height;
}  // end of getHeight()
```

5.2. Moving About

At periodic intervals, TimeBehavior calls MovingBall's updatePosition() method:

```
public void updatePosition()
{
  doRotateY();
  while (!moveFwd())
    doRotateY();
}
```

The ball rotates a small random amount, then moves forward. If the move isn't possible (due to the ball being at the floor's edge), then it keeps turning by small random amounts until it can move.

Rotating the Ball

doRotateY() rotates the ball around its y-axis by a small random angle. The rotation is applied to the ballTG TransformGroup.

```
// globals
private final static double ROTATE_AMT = Math.PI / 18.0;
                                          // 10 degrees
// reusable objects for calculations
private Transform3D t3d, toRot;  // for ballTG's transform


private void doRotateY()
{
  ballTG.getTransform( t3d );
  toRot.rotY( rotateSlightly() );
  t3d.mul(toRot);
  ballTG.setTransform(t3d);
}

private double rotateSlightly()
// rotate between -ROTATE_AMT and ROTATE_AMT radians
{  return rand.nextDouble()*ROTATE_AMT*2 - ROTATE_AMT;  }
```

Translating the Ball

Moving the ball forward consists of three stages: first the translation is calculated, then tested to see if it's allowed. If it's okay, then the move is applied to ballTG.

```
// globals
private final static float STEP = 0.1f;     // step size for moving

// reusable objects for calculations
private Vector3f posnVec, moveVec;       // for the ball's position


private boolean moveFwd()
{
  moveVec.set(0,0,STEP);
  tryMove(moveVec);  // test the move, store new position in posnVec
  if (offEdge(posnVec))
    return false;
  else {  // carry out the move, with a height change
    float heightChg = getHeight(posnVec.x, posnVec.z) +
                                       RADIUS - posnVec.y;
    moveVec.set(0, heightChg, STEP);
    doMove(moveVec);
    return true;
  }
}  // end of moveFwd()
```

The attempted move (which is stored in the posnVec global) contains the ball's new x- and z- positions, which are employed to find the floor's height at that spot. When the move is really executed the height change between the ball's current location and the new spot is included in the move, so the ball moves up or down, following the floor.

Trying out the move involves calculating the effect of the translation, but not updating the ball's TransformGroup, ballTG.

```
// global reusable object for various calculations
private Transform3D toMove;  // for manipulating ballTG


private void tryMove(Vector3f theMove)
{
  ballTG.getTransform(t3d);
  toMove.setTranslation(theMove);
  t3d.mul(toMove);
  t3d.get(posnVec);
}
```

offEdge() checks if the ball's intended move will take it over the floor's edge.

```
// global constant
private final static float OBS_FACTOR = 0.5f;


private boolean offEdge(Vector3f loc)
// is the ball off the edge of the floor?
{
  float r = RADIUS*OBS_FACTOR;
  return ((loc.x - r < -floorLen/2) ||    // off left edge?
          (loc.x + r > floorLen/2) ||     // off right?
          (loc.z - r < -floorLen/2) ||    // off back?
          (loc.z + r > floorLen/2));      // off front?
}
```

OBS_FACTOR is used to reduce the radius of the ball during the tests, which permits the ball to overlap the floor's edge slightly before the move is rejected.

tryMove() applies the ball translation to ballTG.

```
private void doMove(Vector3f theMove)
// translate the ball by the amount in theMove
{
  ballTG.getTransform(t3d);
  toMove.setTranslation(theMove);
  t3d.mul(toMove);
  ballTG.setTransform(t3d);
}  // end of doMove()
```

6. Driving the Balls

The TimeBehavior class is the Java 3D version of a timer: it wakes up every timeDelay milliseconds, and calls the updatePosition() method in all its MovingBall objects.

```
public class TimeBehavior extends Behavior
{
  private WakeupCondition timeOut;
  private ArrayList<MovingBall> mBalls;  // the moving balls

  public TimeBehavior(int timeDelay, ArrayList<MovingBall> mbs)
  { mBalls = mbs;
    timeOut = new WakeupOnElapsedTime(timeDelay);
  }

  public void initialize()
  { wakeupOn( timeOut );  }

  public void processStimulus(Enumeration criteria)
  { // ignore criteria
    for (MovingBall mBall : mBalls)    // move all the balls
      mBall.updatePosition();
    wakeupOn( timeOut );
  }

}  // end of TimeBehavior class
```

7. More Multitexturing

The official Java 3D demos contains a MultiTextureTest.java example in the TextureTest/ directory (texture/ in the Java 3D 1.4 demos) which shows how to combine multiple textures, including the use of a light map generated at runtime.

The alpha mask technique used in my SplashShape class is borrowed from David Yazel's SplatShape class, and Justin Couch's AlphaDemo class. They can be found in the Java3D-interest mailing list archive at http://archives.java.sun.com/java3d-interest.html for May 2003 and July 2002 respectively.

Bump mapping adds lighting detail to a smooth shape, giving it a rough (or bumpy) appearance, without the shape requiring extra polygons. The bump mapping technique based on multitexturing utilizes a texture acting as a *normal map*. A normal map contains texel values for variations of the shape's surface normals. The map creates patterns of shade and light when applied to the shape.

Bump mapping in Java 3D is possible with the COMBINE_DOT3 texture mode, but it's also necessary that the graphics card supports DOT3 texturing. The demos for Java 3D 1.4 include a bump mapping example in the dot3/ directory. Alessandro Borges also has an example at http://planeta.terra.com.br/educacao/alessandroborges/java3d.html. The Yaarq game, by Wolfgang Kienreich, utilizes several texturing effects, including bump mapping (http://www.sbox.tugraz.at/home/w/wkien/#demos).

Java 3D 1.4 supports two shading languages: OpenGL's GLSL (the default choice in Java 3D) and NVIDIA's Cg. Shader programs can carry out processing at the vertex level (involving their position, normals, color, texture coordinates, per vertex lighting, and others), and at the pixel level (affecting texture and pixel colour). They can replicate all the multitexturing effects, and can implement many other forms of texturing.

It's relatively simple to port existing GLSL and Cg shaders over to Java 3D, although your graphics card must support shader functionality. GLSL only became a core part of OpenGL with version 2.0 in 2004, so isn't available in older (or less feature-rich) cards.

The demos for Java 3D 1.4 include shader examples in glsl_shader/ and cg_shader/.