Chapter 17.5. Terrain Generation with Terragen

A significant drawback of the FractalLand3D example from the last chapter is the programmer's lack of control over the generated terrain. Most games require mountains, fields, lakes, and so on, to be in fixed, specific places, rather than randomly located each time the program starts.

A theme of earlier chapters is the use of external modeling software to create complex-looking scenery, artifacts, and figures. The same argument applies to terrain building – there are many excellent tools far more suited to the task than a DIY approach coded in Java.

We will use Terragen, a popular scenery generation package. The landscape is designed with Terragen, then exported as a OBJ file (representing the landscape as a mesh), and as a BMP (showing the surface viewed from above). The BMP is subsequently edited and converted into a JPG.

Our Terra3D application loads the OBJ and JPG files, using the JPG as a texture to cover the Shape3D created from the landscape mesh. Figures 1 and 2 show two landscapes, originally designed with Terragen, then loaded by Terra3D.

Figure 1. Green Valleys with a Castle.

Figure 2. Desert with Sagebrush.

The figures show some of the other elements of Terra3D:

- the landscape can be decorated with two types of scenery: 3D models loaded with our PropManager class, and 2D ground cover images (e.g. the trees, sagebrush) which always stay oriented towards the viewer;
- the ground cover objects utilize SharedGroup nodes so that only a single instance of a shape is created by Java 3D;
- the edges of the landscape are surrounded by walls covered in a mountain range image;
- the sky is filled with stars (an approach suggested by Kevin Duling);

 © Andrew Davison. 2003

- picking is again employed for terrain following (as in chapter 17), but the implementation is complicated by a need to deal with large landscapes. A consequence is that a viewer can walk 'inside' a mountain, but will *eventually* be repositioned on its surface.

1. UML Diagrams for Terra3D

Figure 3 shows the UML diagrams for all the classes in the Terra3D application. The class names and public methods are shown.

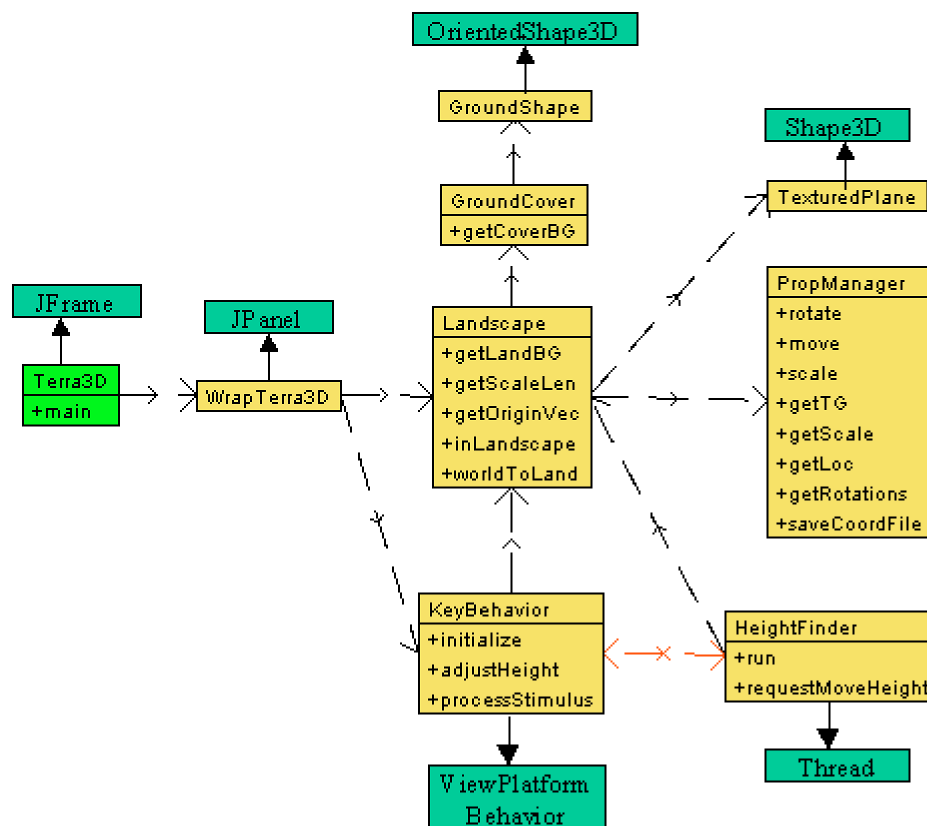

Figure 3. UML Class Diagrams for Terra3D.

Terra3D is the top-level JFrame for the application.

WrapTerra3D creates the world, including the lighting and background. However, most of the work is delegated to a Landscape object which loads the landscape (a combination of an OBJ file and JPG texture), and places walls around it. Each wall is an instance of TexturedPlane, so that a mountain range image can be wrapped over it.

The 3D scenery models are loaded with PropManager objects, and the 2D images are represented by GroundShape objects. A GroundCover object manages the placement of the ground cover, and the sharing of the GroundShape objects.

 © Andrew Davison. 2003

The KeyBehavior class is quite similar to the one in FractalLand3D in chapter 17. The user can move forward, back, left, right, up, down (but not below the ground), and turn left and right. In addition, the 'w' key prints the user's location in the landscape.

Terrain picking is handled by a HeightFinder thread, which communicates its answer back to KeyBehavior when the picking is completed.

2. Terragen

Terragen is a scenery generation package for Windows and the Mac, aimed at producing photorealistic landscape images and animations (http://www.planetside.co.uk/Terragen/). It is extremely easy to use: a beautiful scene can be generated in a few minutes. There are some examples at http://www.planetside.co.uk/Terragen/images.shtml. Terragen is currently free for personal, non-commercial use, although there are some restrictions on the size and resolution of the scenes that can be built.

A terrain can be created using fractals (as in the last chapter), or by painting the shape of the landscape, or as a combination of the two.

Terragen supports powerful surface and colour maps for decorating the landscape, water effects, clouds, atmospheric elements, lighting, and shadows.

Terragen can import and export a wide range of file formats, through the use of plugins. For example, the Firmament plugin allows Terragen to read BMPs, STMs, POV-RAY height fields, and US Geological Survey (USGS) DEM (digital elevation model) and SDTS (spatial data transfer standard) files. The FEO (For Export Only) plugin permits exports to BMP, DXF, OBJ, and RIB files.

Our approach requires exporting to the Wavefront OBJ format, and so FEO must be installed (http://homepages.ihug.co.nz/~jomeder/feo/), and the TGPGuiLib plugin interface for Terragen (http://homepages.ihug.co.nz/~jomeder/tgpguilib/).

Terragen's popularity means that there are numerous other terrain-related tools which can accept, manipulate, and help create Terragen landscapes. A long list is available at http://www.planetside.co.uk/Terragen/resources.shtml. For example, 3DEM is another terrain visualization system with a GIS emphasis (http://www.visualizationsoftware.com/3dem.html). It can read a wide range of USGS and NASA file formats, and save them as Terragen terrains.

2.1. Using Terragen

There is a user guide for Terragen, available from the main Terragen web site. Its emphasis is on explaining the various menu items, dialog boxes, and other GUI elements (http://www.planetside.co.uk/Terragen/guide/). Carol Brooksbank's tutorial takes a different approach, based around explanations of common tasks (http://caroluk2.crosswinds.net/terratut/); it is highly recommended.

Figure 4 shows the Landscape window inside Terragen after a random landscape has been generated with the "Generate Terrain" button. Previously, the water level had been set at –10m, and a surface map loaded for light snow.

 © Andrew Davison. 2003

The Rendering Control window is just visible behind the Landscape window, showing a preview of the user's current vantage point.

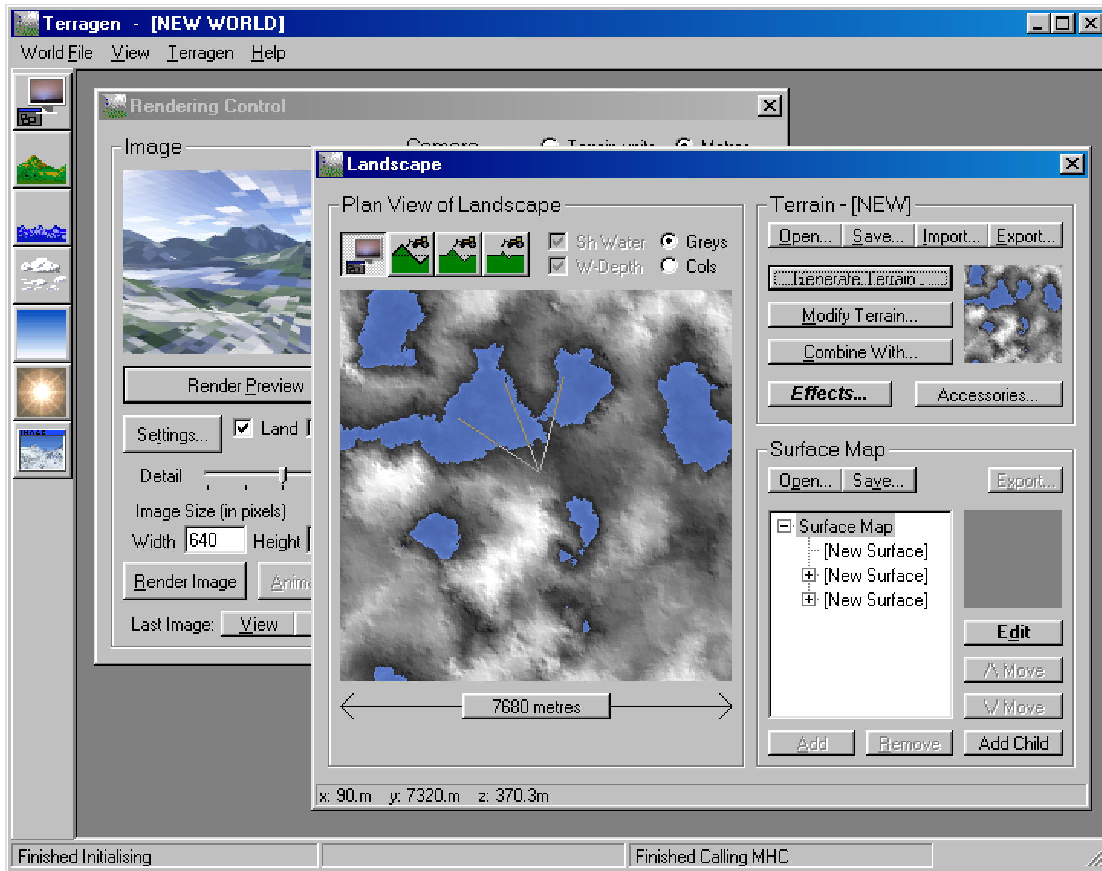

Figure 4. Terragen in Action.

 © Andrew Davison. 2003

Clicking on the "Render Image" button in the Rendering Control window displays the scene. Varying levels of detail can be selected; more detail takes longer to display.

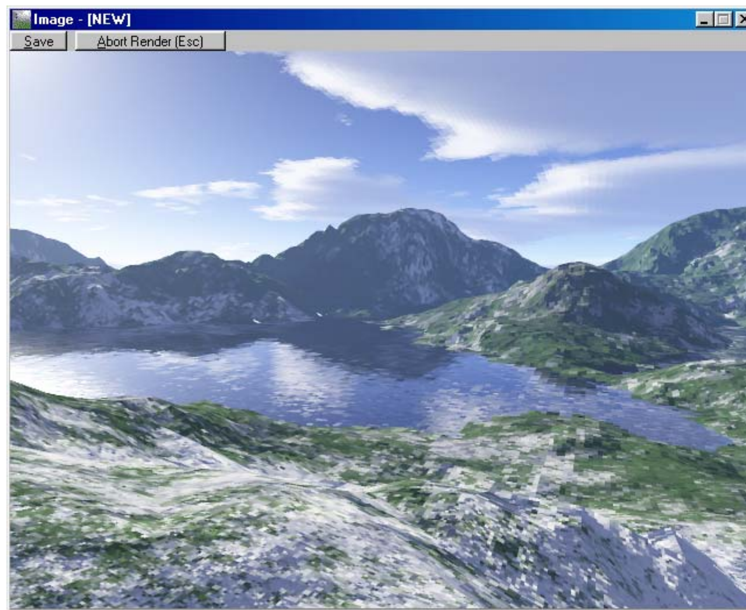

Figure 5. Rendered View.

The landscape mesh can be saved by selecting the FEO Wavefront OBJ item from the Accessories button in the Landscape window. This will only be present if FEO has been installed. Click "Ok" to save, but only after **unchecking** the "Swap Y and Z axes" radio box. This will make the floor of the mesh run along the XY plane, with heights along the z-axis.

A basic landscape like this one will usually generate a 5MB OBJ file, taking a few minutes to do so. An advantage of the OBJ file is its ASCII format, which means that any text editor can open it (but the editor must be capable of viewing very large files).

The resulting OBJ file for our work (named test3.obj) contains two groups of data: the vertices making up the mesh, and face information, with each face made from three vertices. For example, the first few lines of vertex data in test3.obj:

```
v     0.000000    0.000000   198.037763
v     30.000000   0.000000   190.554714
v     60.000000   0.000000   177.648668
v     90.000000   0.000000   172.938180
v     120.000000  0.000000   170.091105
                 :
```

The mesh's floor is the XY plane; the z-values are the heights. Each vertex is 30 landscape units apart, and the vertices are ordered by increasing column (y-axis), row (x-axis) values. The bottom left hand corner of the XY floor is at coordinate (0,0).

The last few line of 'v' data are:

```
            :
v     7620.000000   7680.000000   217.026916
v     7650.000000   7680.000000   212.388668
```

 © Andrew Davison. 2003

```
v     7680.000000   7680.000000   198.037763
```

The top right hand corner of the floor is at (7680,7680), forming a square.

Each 'v' line has an implicit index, starting at 1, and the face lines use them to refer to vertices:

```
f     1 2   258
f     2 259 258
f     2 3   259
f     3 260 259
f     3 4   260
          :
```

These define triangle strips, which will be loaded as a TriangleStripArray by Java 3D, a particularly efficient mesh data structure.

The OBJ file format is somewhat more sophisticated than this. A good place to read about the OBJ capabilities supported by Java 3D is in the documentation for the ObjectFile class. A file may include information about normals, textures, colours, and utilize a separate material file.

A specification of the complete OBJ format can be found at http://www.dcs.ed.ac.uk/home/mxr/gfx/3d/OBJ.spec.

The file can be viewed graphically by a wide variety of packages, including the ObjLoad demo example in the Java 3D distribution (which uses the ObjectFile class to load the file).

One issue is the size of the mesh created by Terragen, which may be too large for some software to handle. For example, ObjLoad requires 128MB of memory to run successfully:

```
java -Xmx128m ObjLoad test3.obj
```

Figure 6 shows the landscape as displayed by ObjLoad. Note that face culling is switched on, so the terrain becomes virtually invisible if turned over.

The beautiful surface texturing is missing, but can be obtained from Terragen by separate means.

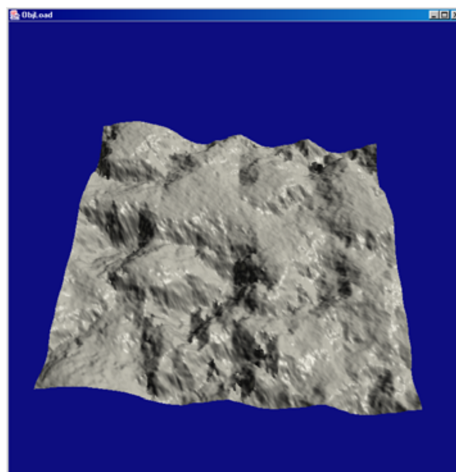

Figure 6. test3.obj in ObjLoad.

 © Andrew Davison. 2003

2.2. Extracting a Terragen Texture

The trick for obtaining a texture is to position the viewpoint directly above the terrain, looking straight down. Currently, Terragen does not support parallel projection, so the view will suffer from distortion at the edges due to perspective effects. These can be reduced by setting camera zoom to 1.

Sky rendering should be switched off, as well as terrain shadows. Setting the sun's altitude to 90 degrees (directly overhead) also helps to minimize shadow effects on the surface map.

The rendering window should be increased in size (e.g. to 800x800), and rendering set to the highest resolution to produce a detailed image. The result is shown in Figure 7.

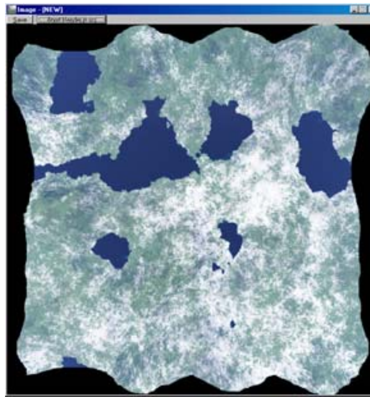

Figure 7. The Terrain from Above.

The image can be saved as a BMP (~2MB file sizes seem common), then converted to a JPG by any number of graphics packages (reducing to ~500K in size). The most important aspects of the conversion are to clip away the black background, and resize the image to form a square. This will remove some of the terrain surface, but the loss isn't noticeable inside Terra3D.

The image should be saved as a high quality JPG. Figure 8 contains the final image, ready to be used as a texture.

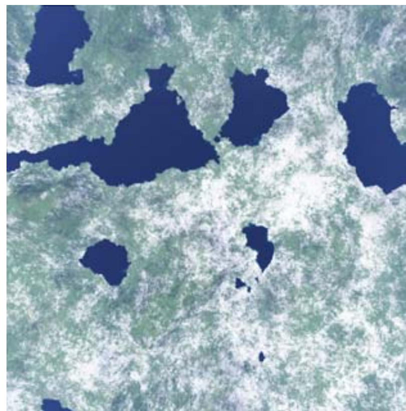

Figure 8. The Terrain Texture.

 © Andrew Davison. 2003

Figure 9 shows the terrain as loaded into Terra3D, with the user standing in roughly the same position as the view presented in Figure 5.

Figure 9. Terra3D's View of the Terrain.

The poor texture resolution compared to the surface maps in Terragen means that a lot of detail is lost. However, the scene is quite sufficient for game play.

Figure 10 displays the user's view after he/she has moved to the other side of the lake, and turned to look back towards the viewpoint employed in Figure 9.

Figure 10. The Other Side of the Lake.

 © Andrew Davison. 2003

3. WrapTerra3D

WrapTerra3D sets up the lighting, background, creates a Landscape object, and initializes the KeyControl object for managing user navigation.

The background code sets the sky to be medium blue, with stars added to its geometry:

```
private void addBackground()
{ Background back = new Background();
  back.setApplicationBounds( bounds );
  back.setColor(0.17f, 0.50f, 0.92f);
  back.setGeometry( addStars() );
  sceneBG.addChild( back );
}
```

Background geometry can be any type of Group node, but there are some restrictions on the shapes which can be held inside the group. For example, Shape3D is fine, but OrientedShape3D is prohibited. The background geometry is restricted to a unit sphere, but is drawn as if located at infinity.

addStars() creates a PointArray of coordinates at random places in the hemisphere above the XZ plane (there's no reason to have stars below the horizon). The colours of the points are randomly chosen.

This coding approach was suggested by Kevin Duling, in an example at http://home.earthlink.net/~kduling/Java3D/Stars/.

```
private BranchGroup addStars()
{
  PointArray starField =  new PointArray(NUM_STARS,
             PointArray.COORDINATES |  PointArray.COLOR_3);
  float[] pt = new float[3];
  float[] brightness = new float[3];
  Random rand = new Random();

  for (int i=0; i < NUM_STARS; i++) {
    pt[0] = (rand.nextInt(2) == 0) ?
              -rand.nextFloat() : rand.nextFloat();
    pt[1] = rand.nextFloat();     // only above the y-axis
    pt[2] = (rand.nextInt(2) == 0) ?
              -rand.nextFloat() : rand.nextFloat();
    starField.setCoordinate(i, pt);

    float mag = rand.nextFloat();
    brightness[0] = mag;
    brightness[1] = mag;
    brightness[2] = mag;
    starField.setColor(i, brightness);
  }

  BranchGroup bg = new BranchGroup();
  bg.addChild( new Shape3D(starField) );
  return bg;
}  // end of addStars()
```


© Andrew Davison. 2003

3.1. Setting up the Users Controls

createUserControls() is quite similar to the same named method in WrapFractalLand3D: it adjusts the clip distances and sets up the KeyBehavior object. The front clip distance is adjusted to reduce the chance of the terrain being clipped away when the user's viewpoint is very close to it.

An additional line sets up depth-sorting for transparent objects in the world:

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

Sorting multiple transparent objects is necessary in Terra3D since ground cover objects are implemented as transparent GIFs textured over shapes. It is likely that the user's viewpoint will include many overlapping instances of these.

The default behavior of Java 3D is to do no depth sorting, which may cause the transparent objects to be drawn in the wrong order, so that far-away trees/bushes are drawn in front of nearer ones.

TRANSPARENCY_SORT_GEOMETRY is limited to sorting independent geometries (e.g. a Shape3D object containing a single geometry). It will not correctly order multiple geometries in a shape.

4. The Landscape Class

The Landscape object is created by WrapTerra3D, and passed a reference to the world's scene (sceneBG), and the filename supplied on the command line (e.g. "test1").

Landscape's primary purpose is to display a terrain composed from a mesh and a texture. Landscape looks in the models/ subdirectory for a OBJ file containing the mesh (e.g. "test1.obj"), and a JPG (e.g. "test1.jpg") to act as the texture. The OBJ file is loaded, becoming the landBG BranchGroup linked to sceneBG. The texture is laid over the geometry stored within landBG.

Landscape can add two kinds of scenery to the terrain:

- 3D shapes, loaded with PropManager. This type of scenery includes irregular objects which the user can move around, and perhaps enter (e.g. the castle shown in Figure 1).
- ground cover, represented by 2D images that rotate to always face the user. This kind of scenery is for simple, symmetrical objects that decorate the ground, such as trees and bushes (see Figures 1 and 2 for examples). Ground cover shapes are managed by a GroundCover object.

The terrain is surrounded by walls covered in a mountain range image.

© Andrew Davison. 2003

4.1. Loading the Mesh

The Landscape() constructor loads the mesh, checks that the resulting Java 3D subgraph has the right characteristics, and extracts various mesh dimensions. At the end of the constructor, the land is added to the world, and the texture laid over it.

```
// globals
private BranchGroup sceneBG;
private BranchGroup landBG = null;
private Shape3D landShape3D = null;

// various mesh dimensions, set in getLandDimensions()
private double landLength, minHeight, maxHeight;
private double scaleLen;


public Landscape(BranchGroup sceneBG, String fname)
{
  loadMesh(fname);        // initialize landBG
  getLandShape(landBG);   // initialize landShape3D

  // set the picking capabilities so that intersection
  // coords can be extracted after the shape is picked
  PickTool.setCapabilities(landShape3D, PickTool.INTERSECT_COORD);

  getLandDimensions(landShape3D);   // extracts sizes from
                                    // landShape3D

  makeScenery(landBG, fname);       // add any scenery
  addWalls();                       // walls around the landscape
  GroundCover gc = new GroundCover(fname);
  landBG.addChild( gc.getCoverBG() );   // add any ground cover

  addLandtoScene(landBG);
  addLandTexture(landShape3D, fname);
}
```

loadMesh() uses Java 3D's utility class, ObjectFile, to load the OBJ file. If the load is successful, the geometry will be stored in a TriangleStripArray below a Shape3D node and BranchGroup. loadMesh() assigns this BranchGroup to the global landBG.

```
private void loadMesh(String fname)
{
  FileWriter ofw = null;
  String fn = new String("models/" + fname + ".obj");
  System.out.println( "Loading terrain mesh from: " + fn +" ..." );
  try {
    ObjectFile f = new ObjectFile();
    Scene loadedScene = f.load(fn);

    if(loadedScene == null) {
      System.out.println("Scene not found in: " + fn);
      System.exit(0);
    }

    landBG = loadedScene.getSceneGroup();    // the land's BG
    if(landBG == null ) {
```

 © Andrew Davison. 2003

```
        System.out.println("No land branch group found");
        System.exit(0);
      }
    }
    catch(IOException ioe)
    { System.err.println("Terrain mesh load error: " + fn);
      System.exit(0);
    }
  }
```

getLandShape() checks that the subgraph below landBG has a Shape3D node, and if the Shape3D is holding a single GeometryArray. The Shape3D node is assigned to the landShape3D global.

```
  private void getLandShape(BranchGroup landBG)
  {
    if (landBG.numChildren() > 1)
      System.out.println("More than one child in land branch group");
    Node node = landBG.getChild(0);
    if (!(node instanceof Shape3D)) {
      System.out.println("No Shape3D found in land branch group");
      System.exit(0);
    }
    landShape3D = (Shape3D) node;
    if (landShape3D == null) {
      System.out.println("Land Shape3D has no value");
      System.exit(0);
    }
    if (landShape3D.numGeometries() > 1)
      System.out.println("More than 1 geometry in land BG");
    Geometry g = landShape3D.getGeometry();
    if (!(g instanceof GeometryArray)) {
      System.out.println("No Geometry Array found in land Shape3D");
      System.exit(0);
    }
  }
```

getLandDimensions() is called from Landscape's constructor to initialize three globals related to the size of the mesh:

- landLength — the length of the X (and Y) sides of the floor of the landscape
- scaleLen — the scaling necessary to fit landLength into LAND_LEN units in the world. scaleLen will be used to scale the landscape.
- minHeight and maxHeight — min and max heights of the landscape

The underlying assumptions are that the floor runs across the XY plane, is square, with its lower left-hand corner at (0,0), and that the z-axis holds the height values.

```
  private void getLandDimensions(Shape3D landShape3D)
  {
    // get the bounds of the shape
    BoundingBox boundBox = new BoundingBox(landShape3D.getBounds());
    Point3d lower = new Point3d();
```

 © Andrew Davison. 2003

```
    Point3d upper = new Point3d();
    boundBox.getLower(lower); boundBox.getUpper(upper);
    System.out.println("lower: " + lower + "\nupper: " + upper );

   if ((lower.y == 0) && (upper.x == upper.y)) {
     // System.out.println("XY being used as the floor");
   }
   else if ((lower.z == 0) && (upper.x == upper.z)) {
     System.out.println("Error: XZ set as the floor;
                            change to XY in Terragen");
     System.exit(0);
   }
   else {
     System.out.println("Cannot determine floor axes");
     System.out.println("Y range should == X range, and
                                          start at 0");
     System.exit(0);
   }

   landLength = upper.x;
   scaleLen = LAND_LEN/landLength;
   System.out.println("scaleLen: " + scaleLen);
   minHeight = lower.z;
   maxHeight = upper.z;
  }  // end of getLandDimensions()
```

The lower and upper corners of the mesh can be obtained easily by extracting the BoundingBox for the shape. However, this approach only works correctly if the shape contains a single geometry.

4.2. Placing the Terrain into the World

The floor of the landscape runs across the XY plane, starting at (0,0), with sides of landLength units, heights in the z-direction. The world's floor is the XZ plane, with sides of LAND_LEN units, and the y-axis corresponding to up and down.

Consequently, the landscape (stored in landBG) must be rotated to lie on the XZ plane, and be scaled to have floor sides of length LAND_LEN. The scaling is a matter of applying the scaleLen global, which equals LAND_LEN/landLength.

In addition, the terrain is translated so that the center of its floor is at (0,0) in the world's XZ plane.

These changes are illustrated by Figure 11.

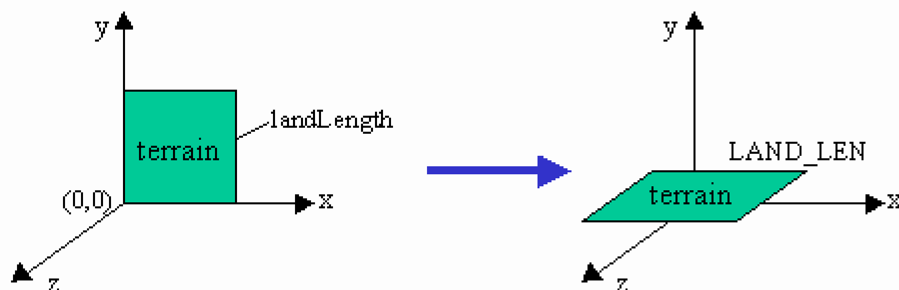

Figure 11. Placing the Terrain.

 © Andrew Davison. 2003

```
private void addLandtoScene(BranchGroup landBG)
{
   Transform3D t3d = new Transform3D();
   t3d.rotX( -Math.PI/2.0 );    // so land's XY resting on XZ plane
   t3d.setScale( new Vector3d(scaleLen, scaleLen, scaleLen) );
   TransformGroup sTG = new TransformGroup(t3d);
   sTG.addChild(landBG);

   // center the land, which starts at (0,0) on the XZ plane,
   // so move it left and forward
   Transform3D t3d1 = new Transform3D();
   t3d1.set( new Vector3d(-LAND_LEN/2, 0, LAND_LEN/2));
   TransformGroup posTG = new TransformGroup(t3d1);
   posTG.addChild( sTG );

   sceneBG.addChild(posTG);  // add to the world
}
```

The subgraph added to sceneBG is shown in Figure 12.

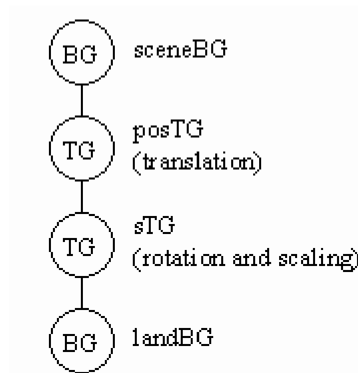

Figure 12. Subgraph for the Landscape.

An essential point is that any nodes added to landBG will be affected by the translation, rotation, and scaling applied to the landscape. This includes the scenery nodes (i.e. the 3D models and ground cover), but the landscape walls are connected to sceneBG, so are not transformed.

The principal reason for connecting nodes to landBG is so their positioning in space can utilize the local coordinate system in landBG. These are the coordinates specified in Terragen: the floor in the XY plane, heights along the z-axis.

 © Andrew Davison. 2003

4.3. Adding Texture to the Terrain

The texture is stretched to fit the terrain stored in landShape3D below landBG. The texture coordinates (s,t), which define a unit square, must be mapped to the (x,y) coordinates of the terrain whose lower left hand corner is at (0,0), top right hand corner at (landLength, landLength). The intended mapping is captured by Figure 13. The simplest way of doing this is define generation planes to translate (x,y) coordinates to (s,t) values.

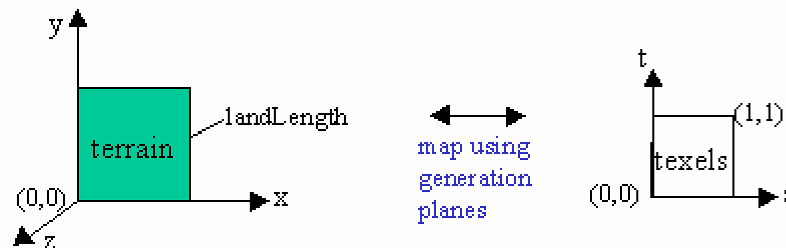

Figure 13. Mapping the Terrain to the Texture.

addLandTexture() sets up the generation planes via a call to stampTexCoords(). It creates an Appearance node for landShape3D, and loads the texture.

```
private void addLandTexture(Shape3D shape, String fname)
{
  Appearance app = shape.getAppearance();

  // generate texture coords
  app.setTexCoordGeneration( stampTexCoords(shape) );

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

  // apply texture to shape
  Texture2D tex = loadLandTexture(fname);
  if (tex != null) {
    app.setTexture(tex);
    shape.setAppearance(app);
  }
}
```

The generation planes are specified using:

s = x / landLength and t = y / landLength

```
private TexCoordGeneration stampTexCoords(Shape3D shape)
{
  Vector4f planeS =
     new Vector4f( (float)(1.0/landLength), 0.0f, 0.0f, 0.0f);
  Vector4f planeT =
     new Vector4f( 0.0f, (float)(1.0/landLength), 0.0f, 0.0f);

  // generate new texture coordinates for GeometryArray
```

 © Andrew Davison. 2003

```
    TexCoordGeneration texGen = new TexCoordGeneration();
    texGen.setPlaneS(planeS);
    texGen.setPlaneT(planeT);
    return texGen;
  }
```

5. Making 3D Scenery

3D scenery are models which the user may enter, move around, and view from different angles. An example is the castle in the test2.obj landscape, as seen in Figures 1 and 14.

Figure 14. The Castle in the test2.obj Terrain

The user can enter a model; in fact the user can walk right through its walls. Terra3D does not impose any constraints on the user's movements around models.

The placement of models, and the user's initial position in the terrain, are specified in a text file with the same name as the landscape's OBJ file; for this example, the text file is test2.txt in the directory models/:

```
start 3960 1800 255.64
Castle.cob    4100  4230  220  70
bldg4.3ds     6780  3840  780  90
hand1.obj     1830  570  781.98   120
```

The file has the format:

```
start x y z
<model file>  x y z scale
<model file>  x y z scale
       :
```

A start line must be included -- it says where the user is placed in the landscape at start time; the scenery objects are optional. The (x,y,z) values are in landscape coordinates, and the scale value is used to adjust the model's size in the terrain.

Each scenery object is loaded with a PropManager object; we assume the existence of "coords" data files for the models.

A difficult question is how to decide on suitable (x,y,z) and scale values? One approach is to jot down likely coordinates while running Terragen. Another technique

 © Andrew Davison. 2003

is to move over the loaded terrain in Terra3D and print out the current position by pressing the ‘w’ button (‘w’ for ‘where’). This functionality is supported by KeyBehavior, described later. Yet another possibility is to open the OBJ file with a text editor, and search through the ‘v’ lines for likely looking coordinates.

None of these approaches help with scale factors, which are mostly ‘guess-timates’.

The scenery models are attached to landBG, and so will be translated, rotated, and scaled in the same way as the landscape. The scale factor for the terrain is stored in scaleLen (it is 0.0078125 in the test2 example). Thus, to render the model at the same size as it was created, the scaling must be ‘undone’ by enlarging it by 1/scaleLen (128 in test2).

Another consideration is the height of the user’s viewpoint. In KeyBehavior, the user’s height above the XZ plane is set to the USER_HEIGHT constant (0.5 world units), which is equivalent to 0.5/0.0078125, or 64 landscape units.

In practice, it is a good idea to start with a scaling factor between 64 and 128: 90 seems a good value.

Landscape’s constructor calls makeScenery() to add scenery to landBG:

```
makeScenery(landBG, fname);
```

fname is the file name supplied on the command line (e.g. “test2”).

makeScenery() parses the scenery file, storing the user’s starting position in originVec, and calls placeScenery() to place each model in the terrain.

originVec is used by KeyBehavior, and so must be specified in world coordinates. However, the input from the scenery file is in terrain coordinates. The translation between the two coordinate systems is done through a call to landToWorld():

```
    originVec = landToWorld(xCoord, yCoord, zCoord);
                  :

private Vector3d landToWorld(double xCoord, double yCoord,
                                                double zCoord)
{ double x = (xCoord * scaleLen) - LAND_LEN/2;
  double y = zCoord * scaleLen;    // z-axis -> y-axis
  double z = (-yCoord * scaleLen) + LAND_LEN/2;   // y- -> z-axis
  return new Vector3d(x, y, z);
}
```

landToWorld() applies the rotation, scaling, and translation utilized when the terrain is connected to the world, as illustrated in Figure 12. The rotation is achieved by swapping the terrain (z,y) values to becomes world (y,z) coordinates. The scaling must be done before the translation in order to mimic the ordering of the transforms in Figure 12.

 © Andrew Davison. 2003

5.1. Placing the Models

placeScenery() attaches a model to landBG. However, some additional transformations must be applied, as shown in Figure 15.

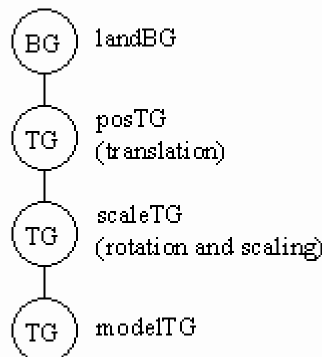

Figure 15. A Model Subgraph below landBG.

The translation and scaling values are supplied for the model in the scenery file. These are given in terms of terrain coordinates because the model is being added to landBG. The rotation is slightly harder to fathom.

The landscape is rotated -90 degrees around the x-axis, and this is applied to the model as well. However, the model is already correctly orientated to rest on the XZ plane, and so the terrain rotation must be undone, by rotating the model +90 degrees back around the x-axis.

```
private void placeScenery(BranchGroup landBG,
         TransformGroup modelTG, double x, double y,
         double z, double scale)
{
  modelTG.setPickable(false);   // so not pickable in scene

  Transform3D t3d = new Transform3D();
  t3d.rotX( Math.PI/2.0 );    // to counter the -ve rot of land
  t3d.setScale( new Vector3d(scale, scale, scale) );   // scaled
  TransformGroup scaleTG = new TransformGroup(t3d);
  scaleTG.addChild( modelTG );

  Transform3D t3d1 = new Transform3D();
  t3d1.set( new Vector3d(x,y,z));  // translated
  TransformGroup posTG = new TransformGroup(t3d1);
  posTG.addChild( scaleTG );

  landBG.addChild( posTG );
}
```

The model is made unpickable, since picking could interact with it if the user went inside the model at run time.

 © Andrew Davison. 2003

6. Adding Landscape Walls

The landscape walls surround the terrain, and are covered with a mountain range image. Calculating the wall's position in space is somewhat simpler if they are added to sceneBG, which permits world coordinates to be used. The walls surround the terrain after it has been rotated and scaled, so lie in the XZ plane, with lengths LAND_LEN (see Figure 16). Their y-axis extent is obtained from the minimum and maximum heights for the terrain, scaled to world-size.

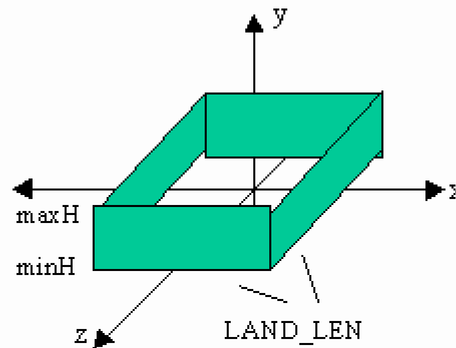

Figure 16. Landscape Walls.

```
// Landscape globals
private static final double LAND_LEN = 60.0;
         // length of landscape in world coordinates
private static final String WALL_PIC = "models/mountain2Sq.jpg";
                :

private void addWalls()
{ // heights used for the walls, in world coords
  double minH = minHeight * scaleLen;
  double maxH = maxHeight * scaleLen;

  // the eight corner points
  // back, left
  Point3d p1 = new Point3d(-LAND_LEN/2.0f, minH, -LAND_LEN/2.0f);
  Point3d p2 = new Point3d(-LAND_LEN/2.0f, maxH, -LAND_LEN/2.0f);

  // front, left
  Point3d p3 = new Point3d(-LAND_LEN/2.0f, minH, LAND_LEN/2.0f);
  Point3d p4 = new Point3d(-LAND_LEN/2.0f, maxH, LAND_LEN/2.0f);

  // front, right
  Point3d p5 = new Point3d(LAND_LEN/2.0f, minH, LAND_LEN/2.0f);
  Point3d p6 = new Point3d(LAND_LEN/2.0f, maxH, LAND_LEN/2.0f);

  // back, right
  Point3d p7 = new Point3d(LAND_LEN/2.0f, minH, -LAND_LEN/2.0f);
  Point3d p8 = new Point3d(LAND_LEN/2.0f, maxH, -LAND_LEN/2.0f);

  // load texture; set mag filter since the image is enlarged
  TextureLoader loader = new TextureLoader(WALL_PIC, null);
  Texture2D texture = (Texture2D) loader.getTexture();
  if (texture == null)
    System.out.println("Cannot load wall image from " + WALL_PIC);
```


© Andrew Davison. 2003

```
    else {
      System.out.println("Loaded wall image: " + WALL_PIC);
      texture.setMagFilter(Texture2D.BASE_LEVEL_LINEAR);
    }

    // left wall
    sceneBG.addChild( new TexturedPlane(p3, p1, p2, p4, texture));
    // front wall
    sceneBG.addChild( new TexturedPlane(p5, p3, p4, p6, texture));
    // right wall
    sceneBG.addChild( new TexturedPlane(p7, p5, p6, p8, texture));
    // back wall
    sceneBG.addChild( new TexturedPlane(p1, p7, p8, p2, texture));
  } // end of addWalls()
```

The same image is used for each wall, mountain2Sq.jpg, shown in Figure 17.

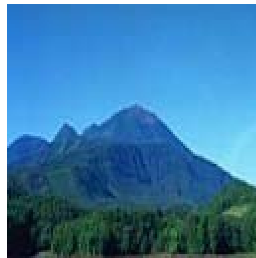

Figure 17. mountain2Sq.jpg: the Mountain Range

It was originally a wide, thin image, but resized to form a square suitable for a texture. It is only 7K in size, and so becomes very pixilated when viewed up close inside Terra3D. For that reason, mag filtering is switched on, to smooth the image's enlargement. It may also be advisable to use a larger image.

Figure 18 shows one of the walls in the test2.obj terrain.

Figure 18. A Landscape Wall in Terra3D.

6.1. The TexturedPlane Class

Each wall is an instance of the TexturedPlane class, which is a simplified descendent of TexturedPlanes in the FractalLand3D example of chapter 17.

The geometry is simpler since only a single QuadArray is created to hold the four corners of the wall. Also, no normals are specified since the node's appearance does

 © Andrew Davison. 2003

not enable lighting or utilize a Material node. The appearance is only the texture, with no lighting effects.

7. Ground Cover

Ground cover is the other kind of scenery for decorating a landscape. A piece of ground cover is represented by a transparent GIF pasted onto a 'screen' which stands on the terrain's surface, and is always oriented towards the viewer. The 'screen' is implemented as a 4-sided QuadArray, inside a GroundShape object (a subclass of OrientedShape3D).

Typical ground cover includes trees, bushes, and road signs. Such elements will appear many times inside a scene, and it would be inefficient to create a separate shape for each one. Instead, the GroundShape object (e.g. a tree) is embedded in a SharedGroup node, which allows the geometry to be shared by multiple TransformGroups. Each TransfromGroup specifies a location for a particular ground cover element, but the actual object is a shared node.

The approach is illustrated by Figure 19.

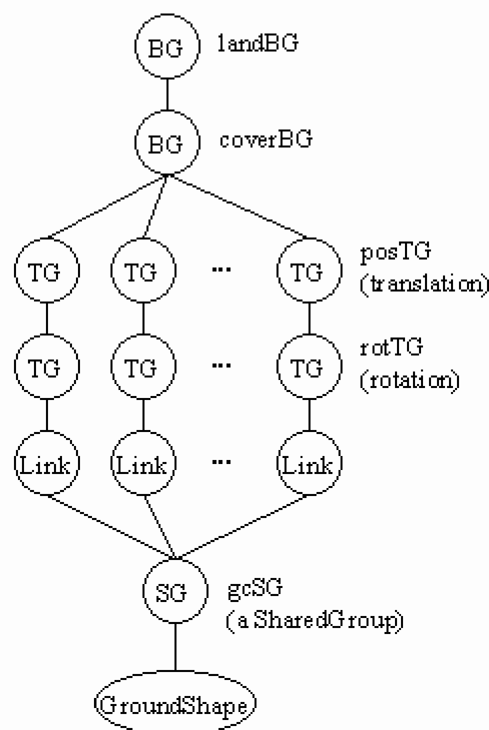

Figure 19. A Subgraph using a Shared GroundShape Object.

Since a GroundShape object is ultimately attached to landBG, terrain coordinates can be used to set its position inside the landscape.

The constructor for Landscape() adds ground cover by calling:

```
GroundCover gc = new GroundCover(fname);
```

 © Andrew Davison. 2003

```
landBG.addChild( gc.getCoverBG() );
```

The GroundCover object, gc, manages the creation of the subgraph holding the ground cover items. The call to getCoverBG() returns the coverBG BranchGroup, the top-level of the subgraph.

7.1. The GroundCover Class

Information about what ground cover should be added to the terrain is given in a <fname>GC.txt text file, where <fname> is the name of the landscape OBJ file. For instance, the information for the test2 scene is stored in test2GC.txt in models/.

```
range 0 500
tree1.gif 90 200
tree2.gif 90 200
tree3.gif 90 200
tree4.gif 90 200
```

The format of a 'GC' file is:

```
range [ min max ]
<gc file1> scale1 number1
<gc file2> scale2 number2
    :
```

The range line specifies the height range within which ground cover may appear. A range restriction is useful for stopping scenery appearing on the surface of lakes or on the tops of peaks.

A <gc file> holds a transparent GIF, which is loaded into its own GroundShape object. The scale argument is used to set the size of the 'screen', and number determines the number of copies of the ground cover placed in the scene.

As with 3D models, the best scale value is largely a matter of trial-and-error, but 90 is a reasonable starting value – it makes the cover a bit taller than the user's viewpoint.

The programmer does not supply terrain coordinates for the scenery. Instead, they are positioned at random locations. Even this is not that simple, since the (x,y,z) data must be obtained from somewhere.

The solution in GroundCover is to read in the OBJ file (e.g. test2.obj) as a text file, and record the 'v' data (the (x,y,z) coordinates of its mesh) as Vector3d objects in the coords ArrayList. Positioning is then a matter of randomly selecting a coordinate from the coords list.

This approach is surprisingly fast, although memory intensive. The principal drawback is the lack of programmer control over placement, including the inability to 'clump' ground cover together: each mesh coordinate is 30 terrain units apart.

 © Andrew Davison. 2003

Figure 20 gives class diagrams for GroundCover and GroundShape, showing all the data and methods.

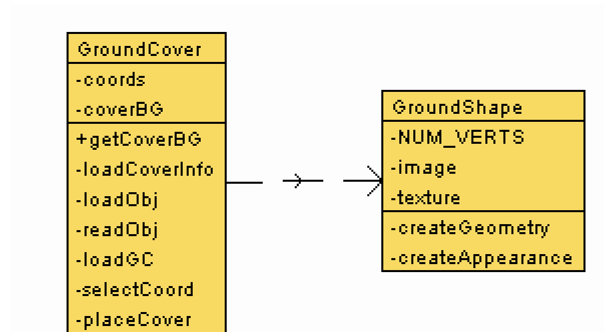

Figure 20. GroundCover and GroundShape Classes.

In GroundCover, loadCoverInfo() loads and parses the 'GC' text file. It calls loadObj() and readObj() to parse the coordinates information extracted from the OBJ file, storing them in the coords ArrayList. Each <gc file> line from the 'GC' file triggers a call to loadGC(), which attaches a subgraph to coverBG similar to the one in Figure 19.

```
private void loadGC(String gcFnm, double scale, int gcNo)
{
  String gcFile = new String("models/" + gcFnm);

  SharedGroup gcSG = new SharedGroup();
  gcSG.addChild( new GroundShape((float) scale, gcFile) );
  gcSG.setPickable(false);    // so not pickable in scene

  Vector3d coordVec;
  for(int i=0; i < gcNo; i++) {   // make multiple TGs using same SG
    coordVec = selectCoord();
    placeCover(gcSG, coordVec);
  }
}
```

The arguments passed to loadGC() are the three values supplied on the <gc file> line in the 'GC' file. A single GroundShape object is created, and multiple transforms are connected to it by repeatedly calling placeCover().

```
private void placeCover(SharedGroup gcSG, Vector3d coordVec)
{
  Transform3D t3d = new Transform3D();
  t3d.rotX( Math.PI/2.0 );     // to counter rotation of the land
  TransformGroup rotTG = new TransformGroup(t3d);
  rotTG.addChild( new Link(gcSG) );

  Transform3D t3d1 = new Transform3D();
  t3d1.set( coordVec );
  TransformGroup posTG = new TransformGroup(t3d1);
  posTG.addChild( rotTG );

  coverBG.addChild( posTG );
```

 © Andrew Davison. 2003

```
}
```

placeCover() creates a single branch down from coverBG to the GroundShape node (see Figure 19).

A common coding error when using a SharedGroup node is to attempt to link it directly to the rest of the scene graph. Each link must be through a Link node.

Figure 19 shows that each branch from coverBG to a Link node holds two TransformGroups. The first (posTG) moves the shape to a location on the terrain's surface; its value comes from a call to selectCoord() in loadGC().

```
private Vector3d selectCoord()
// randomly select a landscape coordinate
{ int index = (int) Math.floor( Math.random()*coords.size());
  return (Vector3d) coords.get(index);
}
```

The second TransformGroup (rotTG) plays a similar role to the sTG node for 3D models (see Figure 12): it counters the –90 degree rotation around the x-axis applied to the landscape.

The reader may notice that sTG in Figure 12 also scales the 3D model, which is not done in the ground cover subgraph in Figure 19. Ground cover scaling is applied directly to the quad geometry inside GroundShape.

7.2. The GroundShape Class

A GroundShape object displays a transparent GIF drawn on the front face of a 4-sided QuadArray. The center of the quad's base is at (0,0,0), resting on the ground. It has sides of screenSize, and is always oriented towards the viewer. The orientation is achieved by making GroundShape a subclass of OrientedShape3D, and setting its axis of rotation to be the y-axis.

GroundShape's createAppearance() sets up the necessary transparency attributes for the shape, and loads the GIF as a texture.

```
private void createAppearance(String fnm)
{
  Appearance app = new Appearance();

  // blended transparency so texture can be irregular
  TransparencyAttributes tra = new TransparencyAttributes();
  tra.setTransparencyMode( TransparencyAttributes.BLENDED );
  app.setTransparencyAttributes( tra );

  // Create a two dimensional texture with min and mag filtering
  TextureLoader loader = new TextureLoader(fnm, null);
  Texture2D tex = (Texture2D) loader.getTexture();
  if (tex == null)
    System.out.println("Image Loading Failed for " + fnm);
  else {
    tex.setMinFilter(Texture2D.BASE_LEVEL_LINEAR);
    tex.setMagFilter(Texture2D.BASE_LEVEL_LINEAR);
    app.setTexture(tex);
```

```
    }

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

Min and mag filtering is used to improve the texture's appearance when viewed close to and from far away.

8. The KeyBehavior Class

The KeyBehaviour class is quite similar to the one in FractalLand3D, it permits the user to move over the surface of the terrain, and to 'float' above it. However, there is one small change, and one large one.

The small change is the addition of the 'w' key, which prints the user's current location on the landscape.

The major change is that a move does not *immediately* affect the user's vertical position on the terrain. The height calculation is delegated to a HeightFinder object, which may take 1-2 secs to obtain the result through picking. In the meantime, KeyBehavior continues to use the old value. As a consequence, the user can move inside mountains, but his vertical position will *eventually* be corrected.

The reason for the slow picking is the large size of the terrain (e.g. over 66,000 vertices in test2.obj, 131,000 faces), all packaged inside a single GeometryArray.

Our approach has the advantage that key processing is decoupled from the height calculation. This means that the KeyBehavior object does not have to wait 1-2 seconds after each move-related key press, which would quickly drive the user to distraction.

At the end of the chapter, we discuss various alternatives to this coding technique.

8.1. Where am I?

When the user presses the 'w' key, printLandLocation() is called.

```
  private void printLandLocation()
  { targetTG.getTransform(t3d);
    t3d.get(trans);
    trans.y -= MOVE_STEP*zOffset;  // ignore user floating
    Vector3d whereVec = land.worldToLand(trans);

    System.out.println("Land location: (" + df.format(whereVec.x) +
          ", " + df.format(whereVec.y) + ", " +
                 df.format(whereVec.z) + ")");
  }
```

The slight problem is that the KeyBehaviour object is attached to sceneBG, and so utilizes world coordinates. However, printing the *landscape* coordinates is more helpful, and so the (x,y,z) data maintained in KeyBehavior must be transformed. This is achieved by calling worldToLand() in Landscape:

 © Andrew Davison. 2003

```
public Vector3d worldToLand(Vector3d worldVec)
{
  double xCoord = (worldVec.x + LAND_LEN/2) / scaleLen;
  double yCoord = (-worldVec.z + LAND_LEN/2) / scaleLen;
                                        // z-axis --> y-axis
  double zCoord = worldVec.y / scaleLen;   // y-axis --> z-axis
  return new Vector3d(xCoord, yCoord, zCoord);
}
```

The transformations apply the operations implicit in the subgraph in Figure 12: the world coordinates are translated, scaled and rotated, to make them into terrain coordinates.

The rotation (a –90 degree rotation around the x-axis) can be conveniently expressed as a switching of the y and z coordinates.

Another way of understanding worldToLand() is as the reverse of landToWorld(), which is also in Landscape.

The ‘w’ key is quite useful when the programmer is trying to decide where to place scenery on the terrain. The user can move over the landscape inside Terra3D, and press ‘w’ when a promising location is encountered. These coordinates can be used in the scenery file to position 3D models.

8.2. Strolling Around the Terrain

The principal method for moving is moveBy(), which has the same interface as the moveBy() method in KeyBehavior in FractalLand3D. The method is called with a predefined ‘step’ for moving forward, back, left, or right.

In FractalLand3D, KeyBehavior’s moveBy() moves the viewpoint in three stages:

1. The next (x,z) position on the floor is calculated by carrying out the move but not updating the user’s actual position. This is done by the tryMove() method.
2. The resulting (x,z) data is passed to getLandHeight() in the Landscape object which employs picking to get the floor height at that location.
3. The viewpoint’s movement along the y-axis is calculated as the change between the current floor height and the height at the new location.

In Terra3D, KeyBehavior’s moveBy() has *four* stages:

1. The next (x,z) position on the floor is calculated with tryMove(), which is identical to the version in FractalLand3D.
2. The (x,z) data is *not* passed to Landscape, but instead to a HeightFinder thread. HeightFinder uses picking to get the floor height.
3. In the meantime, moveBy() uses the current floor height as the height of the new location.

 © Andrew Davison. 2003

4. Later, perhaps 1-2 seconds later, HeightFinder calls adjustHeight() in KeyBehavior. adjustHeight() updates the user's height by the difference between the current floor height and the height at the new location. This is the same as step (3) in FractalLand3D, but is done in a separate method, initiated by a separate thread.

```
private void moveBy(Vector3d theMove)
{
  Vector3d nextLoc = tryMove(theMove);   // next (x,?,z) position
  if (!land.inLandscape(nextLoc.x, nextLoc.z))   //not on landscape
     return;

  hf.requestMoveHeight(nextLoc);  // height request to HeightFinder

  Vector3d actualMove = new Vector3d(theMove.x, 0, theMove.z);
                                     // no y-axis change... yet
  doMove(actualMove);
}


public void adjustHeight(double newHeight)
{
  double heightChg = newHeight - currLandHeight -
                            (MOVE_STEP*zOffset);
  Vector3d upVec = new Vector3d(0, heightChg, 0);
  currLandHeight = newHeight;     // update current height
  zOffset = 0;                    // back on floor, so no offset
  doMove(upVec);
}
```

Both moveBy() and adjustHeight() call doMove() which updates the viewpoint position. This method is unchanged from the one in FractalLand3D, except that is it now prefixed with the synchronized keyword. This prevents KeyBehavior and HeightFinder from calling it at the same time.

9. The HeightFinder Class

KeyBehaviour interacts with HeightFinder by calling requestMoveHeight() which stores a (x,y,z) coordinate in the global theMove Vector3d object.

```
// globals
private Vector3d theMove;  // current move request from KeyBehavior
private boolean calcRequested;
     :


synchronized public void requestMoveHeight(Vector3d mv)
{ theMove.set(mv.x, mv.y, mv.z);
          // will overwrite any pending request in theMove
  calcRequested = true;
}
```

 © Andrew Davison. 2003

The (x,z) values in theMove will be used for the picking calculation. The calcRequested boolean signals a pending request.

If the user presses the move keys very rapidly, KeyBehavior will call requestMoveHeight() frequently, which will cause theMove to be updated. Thus, when HeightFinder eventually processes the next request it will find only the most recent one in theMove, saving itself unnecessary work. This illustrates the decoupling of key processing from height calculation – the user can move as fast (or as slow) as they like.

requestMoveHeight() returns immediately; KeyBehavior does not wait for an answer, or it might be waiting for 1-2 seconds. Instead, KeyBehavior uses the current y-height for its move.

HeightFinder's run() method constantly loops: reading the current move request from theMove, then calling getLandHeight(). At the end of getLandHeight(), the new height is passed back to KeyBehavior.

```
public void run()
{ Vector3d vec;
  while(true) {
    if (calcRequested) {
      vec = getMove();      // get the requested move
      getLandHeight(vec.x, vec.z);   // pick with it
    }
    else {   // no pending request
      try {
        Thread.sleep(200);    // sleep a little
      }
      catch(InterruptedException e) {}
    }
  }
}
```

The data in theMove is obtained via getMove():

```
synchronized private Vector3d getMove()
{ calcRequested = false;
  return new Vector3d(theMove.x, theMove.y, theMove.z);
}
```

The method is synchronized, as is requestMoveHeight(), since both access/change theMove and calcRequested, and we want to impose mutual exclusion on those operations.

9.1. Picking in HeightFinder

getLandHeight() in HeightFinder implements picking on the landscape, and essentially uses the same code as getLandHeight() in the Landscape class in FractalLand3D.

Placing the code in the HeightFinder class is a matter of taste – all the height calculations should be located in a single object.

 © Andrew Davison. 2003

The HeightFinder constructor is passed a reference to the Landscape object so that terrain-related data and methods can be accessed.

```
// globals
private Landscape land;
private KeyBehavior keyBeh;
private PickTool picker;
private double scaleLen;
           :

public HeightFinder(Landscape ld, KeyBehavior kb)
{
  land = ld;
  keyBeh = kb;
  picker = new PickTool(land.getLandBG());   //only check landscape
  picker.setMode(PickTool.GEOMETRY_INTERSECT_INFO);

  scaleLen = land.getScaleLen();  // scale factor for terrain
  theMove = new Vector3d();
  calcRequested = false;
}
```

The PickTool object, picker, is configured to examine only landBG. However, 3D models and ground cover are also attached to this node, so they should be made unpickable; we are only interested in obtaining floor coordinates at pick time.

getLandHeight() is like the same named method in FractalLand3D. One, perhaps surprising, feature of the code is that the pick ray is specified in world coordinates even though the PickTool is attached to landBG (which uses landscape coordinates).

```
// global
private final static Vector3d DOWN_VEC =
                        new Vector3d(0.0,-1.0,0.0);
        :


private void getLandHeight(double x, double z)
{
  // high up in world coords; shoot a ray downwards
  Point3d pickStart = new Point3d(x, 2000, z);
  picker.setShapeRay(pickStart, DOWN_VEC);

  PickResult picked = picker.pickClosest();
  if (picked != null) {   // pick sometimes misses at edge/corner
    if (picked.numIntersections() != 0) { //sometimes no intersects
      PickIntersection pi = picked.getIntersection(0);
      Point3d nextPt;
      try {   // handles 'Interp point outside quad' error
        nextPt = pi.getPointCoordinates();
      }
      catch (Exception e) {
         System.out.println(e);
         return;   // don't talk to KeyBehavior as no height found
      }

      double nextYHeight = nextPt.z * scaleLen;
```

 © Andrew Davison. 2003

```
                              // z-axis land --> y-axis world
        keyBeh.adjustHeight(nextYHeight);  //tell KeyBehavior height
      }
    }
  }  // end of getLandHeight()
```

The intersection point is obtained in local (i.e. terrain) coordinates by calling getPointCoordinates(); there is also a getPointCoordinatesVW() which would return the point in world coordinates.

KeyBehavior utilizes world coordinates, and so the height value (nextPt.z) must be converted, but this can be done inexpensively by scaling the point directly, to give nextYHeight. This is passed to KeyBehavior by calling adjustHeight().

10. Speeding up Terrain Following

The code for terrain following sacrifices accuracy for speed, meaning that a user can move relatively quickly over the terrain with the disadvantage that he/she can move straight into the side of a mountain or float off the edge of a cliff. Eventually, their vertical position will be corrected, but even a temporary position disparity is disconcerting. What can be done?

There are a number of tricks that can be utilized, without making any fundamental changes to the application.

The step distance used in KeyBehavior is 0.2 world units, which is 26 terrain units (0.2/0.0078125). As a rough comparison, each mesh coordinate is 30 units apart. If the step distance was reduced, the user would move over the terrain slower, and height changes would occur more gradually. This would help the picking code keep up with the height adjustments.

It is possible to increase the user's height, which is 0.5 world unit in KeyBehavior, or 64 terrain units. If the eye line is higher off the floor, then a larger height change would be required before the user noticed that he had sunk beneath the surface.

Two other 'solutions' are to create landscapes with gentler slopes, and make it impossible for a user to move into rough terrain by imposing restrictions on KeyBehavior. This kind of behavior is typical of racing games, where a vehicle can only move a short distance off the racing track.

Since HeightFinder is a thread, why not throw more threads at the task? The single HeightFinder thread takes 1-2 seconds to finish a calculation before it can start the next one, which adds a considerable delay to the processing time for the next request. By having several threads, the turn-around time would be reduced since the wait time is less.

I tried this and discovered that picking cannot be threaded! Multiple threads can initiate picking at roughly the same time, but each operation is sequentialised, and a bit of extra time added in the process. Threads may be more successful if the picking is carried out on distinct GeometryArrays, but I have yet to try this.

 © Andrew Davison. 2003

The real problem is that the mesh is too large for efficient picking. It should be divided into pieces, perhaps based on the terrain's latitude and longitude. Then, as a user moves over the terrain, the relevant terrain piece can be selected easily, and picking would be much faster due to the smaller size of the piece.

Multiple pieces also lend themselves to the support of multiple levels of detail and dynamic loading. Most of the approaches described below use these ideas.

11. Other Approaches

The reader should look back to sections 8 and 10 of chapter 17 for some discussion of fractal landscapes and ROAM.

The rendering of USGS height field data is a popular way of creating a real-world scene. As mentioned in section 10 of chapter 17, j3d.org contains a DEM loader, and the JLand applet/application can read DEM files and several other formats.

Terragen can utilize a variety of USGS and NASA file formats, when used in conjunction with plugins and/or other terrain-based applications, such as 3DEM. This makes the data available to Terra3D, once it has been saved as OBJ and JPG files.

11.1. DEM Viewer

The JavaWorld article "Navigate through Virtual Worlds using Java 3D" by Mark O. Pendergast is an excellent tutorial on utilizing DEM files, and includes fly-through navigation controls using the keyboard and mouse, and level-of-detail (LOD) displays of the data (http://www.javaworld.com/javaworld/jw-07-2003/jw-0704-3d_p.html).

All the source code can be downloaded, and includes a Grand Canyon example.

The data is stored internally in TriangleStripArrays using interleaving and by-reference, which is the most efficient format in terms of memory and processor usage. The data is stored as floats, which is adequate for the terrain resolution, and matches Java 3D's internal data representation.

Interleaving is utilized to support colour and normal information alongside the coordinates. The normals are generated in the code, rather than via a NormalGenerator. Texturing is not used. Although interleaved data complicates the coding, it speeds up rendering.

Rather than using a single TriangleStripArray (as in Terra3D), Pendergast employs many arrays so that LOD features can be offered. The terrain is divided into segments, with each segment represented by several TriangleStripArray holding varying amounts of (by-reference) mesh detail. By-reference means that mesh data is shared between the various segments, saving considerable space.

Switch nodes multiplex between the arrays at run-time: as the user approaches a segment (moves away from one), then more (less) mesh detail is displayed.

Pendergast's article is a work-in-progress. He is currently working on swapping segments in and out of memory at run time.

 © Andrew Davison. 2003

11.2. The CLOD Algorithm

Martin Barbisch implemented a Java 3D application for his student project which illustrates the CLOD (Continuous Level of Detail) algorithm (http://wwwvis.informatik.uni-stuttgart.de/javatevi/). The amount of detail visible to the user is adjusted at run time as the viewer moves. The drawback is a 'popping' effect as details suddenly appear as the user gets closer to some part of the terrain. The solution is *geomorphing*, a form of mesh morphing

11.3. DTV Applet

Nathan Bower has written a Java 3D applet which displays a UK ordinance survey map, textured over a height field for the same area (http://www.nathanbower.com/cms?page_id=03_Java3D_Applet.htm&folder=/02_My_Work). No source code is available, but the information supplied on the Web page suggests that he has implemented most of the components himself: the triangulation scheme, viewpoint clipping, the mouse controls.

He also mentions an application that can display multiple ordinance survey maps, tiled together, sufficient in number to cover the entire UK.

11.4. JCanyon

JCanyon is a flight simulator which visualizes its large data set (about 300MB) using "OpenGL for Java" rather than Java 3D (http://java.sun.com/products/jfc/tsc/articles/jcanyon/).

The terrain is divided into tiles, and multiple samples are prepared for each tile, with varying coordinate and texture resolutions. Only the data for the required resolution is mapped into memory at any given time. Tiles are fetched from the disk in a background thread.

Two novelties of JCanyon are its texture selection technique and a method for eliminating cracks between tiles, called *filleting*.

 © Andrew Davison. 2003