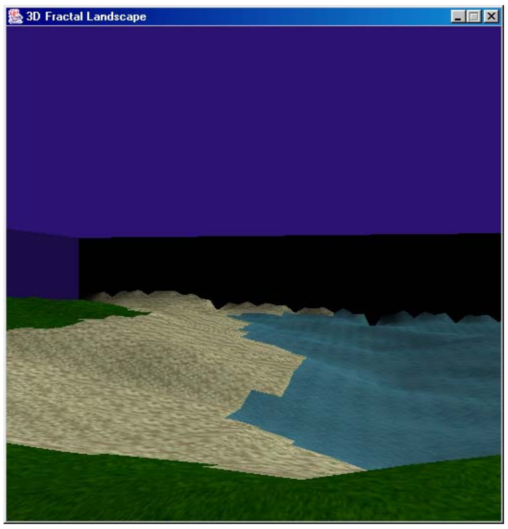
Chapter 17. Fractal Land

The FractalLand3D application creates a landscape using a plasma fractal to generate height values for the landscape’s component quads. The flatness/roughness of the terrain is controlled by a number entered at the command line which affects the fractal.

The quads are covered with textures, which vary depending on the the quad’s average height. They are affected by lighting, and can be cast into a foggy gloom.

The user navigates with the usual FPS key controls, and automatically follows the lie of the land, moving up hills, and down into valleys.

Figure 1 shows two screenshots: one with the default flatness value, the other with increased roughness. No fog is present in these runs.



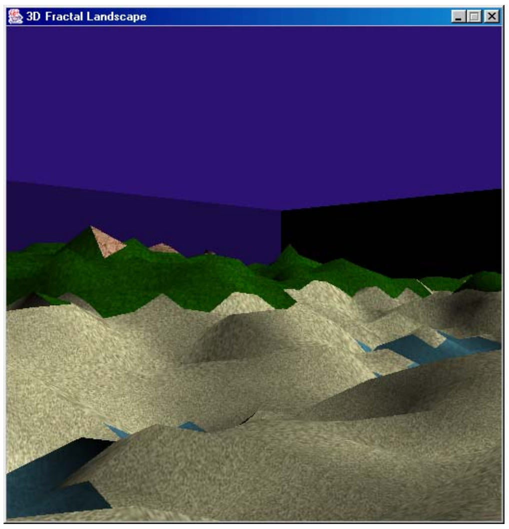


Figure 1. Sprites in Motion in Tour3D.

Other features;

- picking is used to implement terrain following;
- the component quads of the landscape are grouped together based on their average heights to reduce the number of Shape3D nodes in the scene;
- textures are minified to reduce shimmering when seen at a distance;
- normals are generated for the quads automatically, with the aid of the GeometryInfo and NormalGenerator classes;
- the geometry is optimised with the Stripifier utility;
- a linear fog reduces visibility.

1. UML Diagrams for FractalLand3D

Figure 2 shows the UML diagrams for all the classes in the FractalLand3D, including public methods.

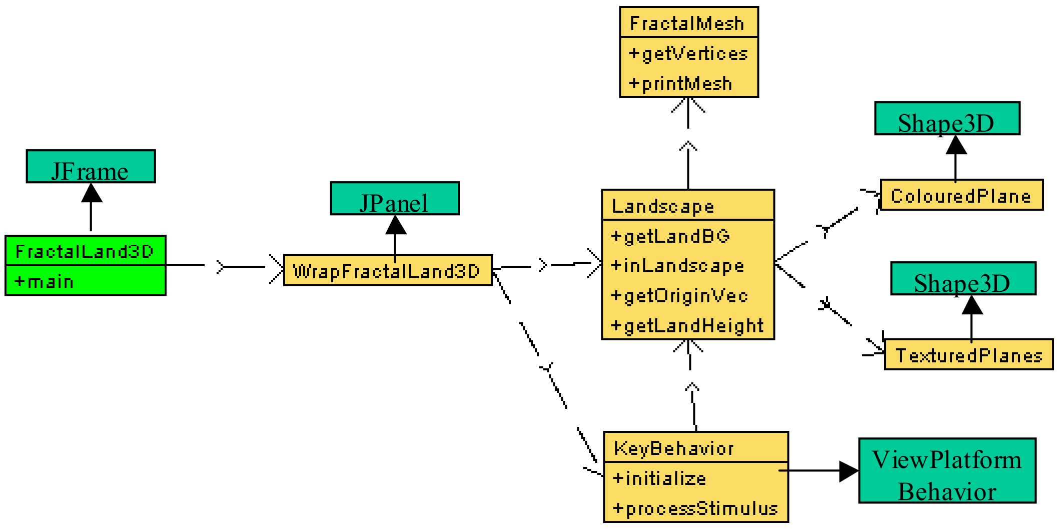


Figure 2. UML Class Diagrams for FractalLand3D.

FractalLand3D is the top-level JFrame for the application, and is very similar to our earlier examples, except that it extracts a double from the command line to be used as a 'flatness' value by the fractal.

WrapFractalLand3D creates the scene, using Landscape to generate the terrain and walls. It also initiates a KeyBehavior object.

Landscape uses FractalMesh to generate a collection of coordinates for the terrain, and then converts them into quads grouped into at most five TexturedPlanes objects. All the quads in a given TexturedPlanes object use the same texture.

Landscape creates the walls around the terrain with four ColouredPlane objects.

Landscape also contains a getLandHeight() method, used by KeyBehavior to calculate the floor height at a given (x,z) location. KeyBehavior employs this information to position the user's viewpoint at the correct height above the floor as it moves.

The code can be found in Code/FractalLand3D/

2. What is Flatness?

The flatness value is passed from FractalLand3D, through WrapFractalLand3D and Landscape, into FractalMesh, where it controls the rate at which the height variation reduces to 0. By trial-and-error, we have set the number to be adjustable in the range 1.6 to 2.5, with 2.3 being the default. A 1.6 value creates a very hilly landscape, while 2.5 makes it almost flat.

3. WrapFractalLand3D

createSceneGraph() brings the various elements of the scene together:

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

  lightScene();      // add the lights
  addBackground();   // add the sky
  // addFog();       // add the fog

  // create the landscape: the floor and walls
  land = new Landscape(flatness);
  sceneBG.addChild( land.getLandBG() );

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

lightScene() creates a single directional light, with no ambient backup, which makes the scene suitably dim. The sky colour is set to be dark blue, which is also used for the fog.

3.1. Linear Fog

The examples in Figure 1 use the code shown above, with the call to addFog() commented out. When uncommented, the scene looks something like Figure 3.

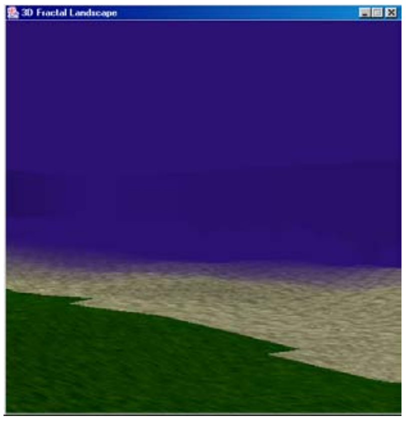


Figure 3. A Foggy Scene.

The addFog() method:

```
private void addFog()
{ LinearFog fogLinear = new LinearFog( skyColour, 15.0f, 30.0f);
  fogLinear.setInfluencingBounds( bounds );
  sceneBG.addChild( fogLinear );
}
```

The LinearFog node makes the scene start to fade at 15.0 units from the user, and be totally obscured at 30.0f. Since the sides of the scene are 64 units long, visibility is consequently reduced. The fog colour, skyColour, is also used for the background, creating an effect of things fading away because of their distance from the user.

Java 3D also has an ExponentialFog node which makes a fog that seems ‘heavier’ and closer, more akin to real-world fog.

Aside from the added realism, fog also may also allow Java 3D to cull more objects from the scene, so speeding up rendering.

3.2. User Controls

WrapFractalLand3D calls createUserControls() to adjust the clip distances and to set up the KeyBehavior object:

```
private void createUserControls()
{
  // original clips are 10 and 0.1; keep ratio between 100-1000
  View view = su.getViewer().getView();
  view.setBackClipDistance(20);      // can see a long way
  view.setFrontClipDistance(0.05);   // can see close things

  ViewingPlatform vp = su.getViewingPlatform();
  TransformGroup steerTG = vp.getViewPlatformTransform();

  // set up keyboard controls (and position viewpoint)
  KeyBehavior keybeh = new KeyBehavior(land, steerTG);
  keybeh.setSchedulingBounds(bounds);
  vp.setViewPlatformBehavior(keybeh);
}
```

The clip distances are adjusted in the same way, and for the same reasons, as in Maze3D in chapter 16. If the front clip is left at 0.1 units then the user can often see through the side of a hill when the terrain is very steep, and the viewpoint is positioned directly in front of the hill.

KeyBehavior handle the processing triggered by key presses, and also sets up the initial viewpoint position; in previous examples the starting point was managed by the ‘Wrap’ class itself.

4. Landscape

Landscape's initial task is to build the terrain and the four walls around it. The resulting scene graph is shown in Figure 4.

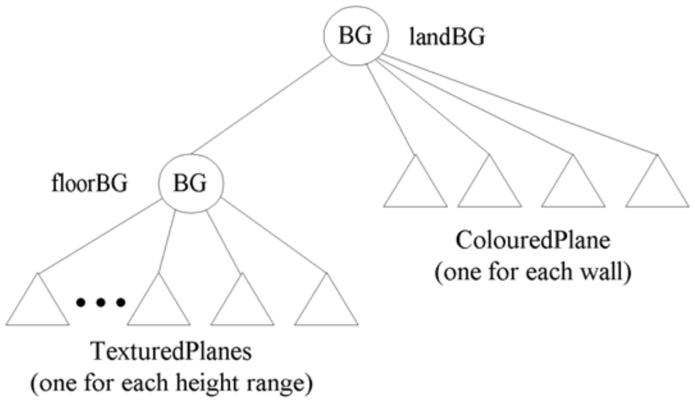


Figure 4. Landscape's Scene Graph.

The number of TexturedPlanes objects will vary depending on the height of the generated quads. Each TexturedPlanes holds all the quads within a given height range, so allowing them to be assigned the same texture.

This approach means that the number of TexturedPlanes in a scene does not depend on the number of quads, but on the number of distinct height ranges hardwired into the code. The height boundaries are calculated by dividing the number of textures into the minimum-to-maximum height range.

The TexturedPlanes are grouped under the floorBG BranchGroup so that picking can be localised to everything below floorBG, excluding the walls attached to the landBG BranchGroup.

There are no TransformGroup nodes in the graph, and landBG is attached to the top-level sceneBG node by WrapFractalLand3D. This means that the local coordinates used inside the TexturedPlanes and ColouredPlane objects are also scene coordinates, and no mapping between local and world values is necessary. This is an important optimisation when picking is being employed.

4.1. Floor Creation

Landscape calls on FractalMesh to generate the coordinates for the floor:

```
FractalMesh fm = new FractalMesh(flatness);
// fm.printMesh(1);   // for debugging: x=0; y=1; z=2
vertices = fm.getVertices();
```

The size of the floor is FLOOR_LEN, centered around the origin in the XZ plane.

The vertices[] array holds Point3d objects organised by FractalMesh into quad order. Each group of four points are in counter-clockwise order of (x,z) when the points are viewed from above, with the bottom leftmost point first. For example, a possible sequence of points might be: (0,5,1), (1,5,1), (1,5,0), (0,5,0), which corresponds to the quad shown in Figure 5.

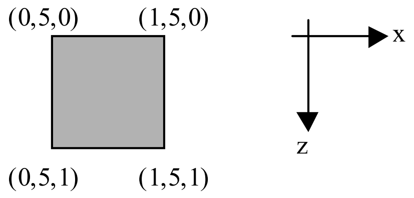


Figure 5. A Quad from above.

The quads in vertices[] are sorted into separate ArrayLists in platifyFloor() based on their average heights, and then passed to TexturedPlanes objects.

```
private void platifyFloor()
{
  ArrayList[] coordsList = new ArrayList[NUM_TEXTURES];
  for (int i=0; i < NUM_TEXTURES; i++)
    coordsList[i] = new ArrayList();

  int heightIdx;
  for (int j=0; j < vertices.length; j=j+4) {   // test each quad
    heightIdx = findHeightIdx(j);   //which height applies to quad?
    addCoords( coordsList[heightIdx], j);   // add quad to list
    checkForOrigin(j);       // check if (0,0) in the quad
  }

  // use coordsList and texture to make TexturedPlanes object
  for (int i=0; i < NUM_TEXTURES; i++)
    if (coordsList[i].size() > 0)    // if used
      floorBG.addChild( new TexturedPlanes(coordsList[i],
                 "images/"+textureFns[i]) );  // add to floor
} // end of platifyFloor()
```

Also passed to each TexturedPlanes is a filename holding the texture that will be laid over each of the quads. The resulting TexturedPlanes object is a Shape3D which is added to floorBG, as shown in Figure 4.

platifyFloor() also checks the coordinates in vertices[] to find the one positioned at (0,0) on the XZ plane, to obtain its height. The coordinate is stored in originVec, and made accessible with the public method getOriginVec(). That method is utilised by KeyBehavior to position the viewpoint.

4.2. Walls Creation

addWalls() generates eight coordinates which specify corners around the floor. They are passed to four ColouredPlane objects to create the walls that are attached to landBG (see Figure 4). A code fragment:

```
private void addWalls()
{
  Color3f eveningBlue = new Color3f(0.17f, 0.07f, 0.45f);

  // the eight corner points
  // back, left
  Point3d p1 = new Point3d(-WORLD_LEN/2.0f, MIN_HEIGHT,
                                             -WORLD_LEN/2.0f);
  Point3d p2 = new Point3d(-WORLD_LEN/2.0f, MAX_HEIGHT,
                                             -WORLD_LEN/2.0f);
        :
  // back, right
  Point3d p7 = new Point3d(WORLD_LEN/2.0f, MIN_HEIGHT,
                                             -WORLD_LEN/2.0f);
  Point3d p8 = new Point3d(WORLD_LEN/2.0f, MAX_HEIGHT,
                                                   ORLD_LEN/2.0f);

  // left wall; counter-clockwise
  landBG.addChild( new ColouredPlane(p3, p1, p2, p4,
                        new Vector3f(-1,0,0), eveningBlue) );
        :
  // back wall
  landBG.addChild( new ColouredPlane(p7, p8, p2, p1,
                        new Vector3f(0,0,1), eveningBlue) );
}
```

As usual, care must be taken to supply the points in the right order so that the front face of each plane is pointing into the scene. Otherwise, the lighting effects will appear on the face pointing out, hiding them from the user.

4.3. On the Floor

Landscape offers the public method inLandscape() which is used by KeyBehavior to prevent a user walking off the edge of the floor. inLandscape() tests the user’s current position against the extent of the floor.

```
public boolean inLandscape(double xPosn, double zPosn)
{
  int x = (int) Math.round(xPosn);
  int z = (int) Math.round(zPosn);
  if ((x <= -WORLD_LEN/2) || (x >= WORLD_LEN/2) ||
      (z <= -WORLD_LEN/2) || (z >= WORLD_LEN/2))
    return false;
  return true;
}
```

4.4. Picking a Height

getLandHeight() is called from KeyBehavior to get the height of the floor at a given (x,z) position. It uses picking on the floor's BranchGroup to obtain this information. However, before getLandHeight() can be employed, a PickTool object is created in Landscape's constructor:

```
private PickTool picker;   // global

public Landscape(double flatness)
{   :
  picker = new PickTool(floorBG);   // only check the floor
  picker.setMode(PickTool.GEOMETRY_INTERSECT_INFO);
    :
}
```

picker's mode is set so that the intersection coordinates can be obtained. It is also restricted to the floor's BranchGroup, eliminating the walls from the intersection calculations.

getLandHeight() is supplied with the intended next (x,z) position for the user. It uses these values as a starting point for a downwards pointing pick ray which intersects with the floor. The returned intersection information includes the height of the floor at that (x,z) point, which is passed back to KeyBehavior. Unfortunately, getLandHeight () is made more complicated by having to deal with several error cases:

```
public double getLandHeight(double x, double z, double currHeight)
{
  Point3d pickStart = new Point3d(x,MAX_HEIGHT*2,z);  // start high
  picker.setShapeRay(pickStart, DOWN_VEC);   // shoot ray downwards

  PickResult picked = picker.pickClosest();
  if (picked != null) {    // pick sometimes misses an 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 currHeight;
      }
      return nextPt.y;
    }
  }
  return currHeight;  // if we reach here, return old height
}
```

The approach is to shoot a ray downwards with setShapeRay(). DOWN_VEC is:

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

pickClosest() gets the PickResult nearest to the ray's origin, which should be a quad inside a TexturedPlanes node. The PickResult object is then queried to get a PickIntersection object, and the intersection coordinate is accessed with getPointCoordinates(). This level of access requires that the TexturedPlanes objects have the picking capability INTERSECT_COORD:

```
public TexturedPlanes(...)
{    :
    PickTool.setCapabilities(this, PickTool.INTERSECT_COORD);
}
```

The potential errors dealt with by getLandHeight() are:

- no quad is found by the pick ray, so the PickResult object is null;
- the PickResult object contains no intersections;
- the extraction of the intersection coordinate from the PickIntersection object raises an "Interp point outside quad" exception.

In fact, none of these exceptional cases should occur, but occasionally do nevertheless, especially if the ray intersects with a quad at its edge or corner.

The recovery strategy is to have KeyBehavior pass the current floor height into getLandHeight() (i.e. the height of the floor where the user is currently standing). If an error occurs, then that height is returned.

The effect is that when the user moves, the viewpoint may remain at the same height even if the user is moving up or down a hill. In practice, this is not much of a problem since the errors occur infrequently, and a single move only adjusts the height marginally.

5. TexturedPlanes

The geometry for a TexturedPlanes object consists of a series of quads, each requiring texture coordinates and normals. Creating the texture coordinates is straightforward since the ordering of the quad (counter-clockwise, bottom-left point first) is fixed by FractalMesh. The normals are more problematic since the y-values of a quad can make it twist in any direction.

The geometry is a QuadArray, and the ArrayList of input coordinates are converted to an array before they can be used:

```
private void createGeometry(ArrayList coords)
{
  int numPoints = coords.size();
  QuadArray plane = new QuadArray(numPoints,
          GeometryArray.COORDINATES |
          GeometryArray.TEXTURE_COORDINATE_2 |
          GeometryArray.NORMALS );

  // obtain coordinates
  Point3d[] points = new Point3d[numPoints];
  coords.toArray( points );
       :
```

The texture coordinates are created in the same order as the points in a quad, and repeated for each quad:

```
TexCoord2f[] tcoords = new TexCoord2f[numPoints];
for(int i=0; i < numPoints; i=i+4) {
  tcoords[i] = new TexCoord2f(0.0f, 0.0f);   // for 1 point
  tcoords[i+1] = new TexCoord2f(1.0f, 0.0f);
  tcoords[i+2] = new TexCoord2f(1.0f, 1.0f);
  tcoords[i+3] = new TexCoord2f(0.0f, 1.0f);
}
```

The calculation of the normals is carried out by a NormalGenerator object, which requires the coordinates and texels to be stored in a GeometryInfo object.

```
// create GeometryInfo
GeometryInfo gi = new GeometryInfo(GeometryInfo.QUAD_ARRAY);
gi.setCoordinates(points);
gi.setTextureCoordinateParams(1, 2); // one set of 2D texels
gi.setTextureCoordinates(0, tcoords);

// calculate normals with very smooth edges
NormalGenerator ng = new NormalGenerator();
ng.setCreaseAngle( (float) Math.toRadians(150));  //default is 44
ng.generateNormals(gi);
```

The setTextureCoordinatesParams() specifies how many texture coordinate sets will be used with the geometry, and their dimensionality (Java 3D offers 2D, 3D, and 4D texture coordinates).

The NormalGenerator adds vertex normals to the geometry inside GeometryInfo. The crease angle is used to decide when two adjacent quads should be given separate normals, which occurs when the angle between them is greater than the crease angle.

By increasing the angle from 44 to 150 degrees, the edges between most quads are made 'smoother' when light is 'reflected' off them: instead of a sharp separation in light and shade at the shared edges, the pattern of light is more diffuse, giving the impression that the underlying geometry is more rounded. This is a useful trick for landscaping since no changes need to be done to geometry's coordinates, only to the normals.

After the NormalGenerator, the Stripifier utility converts the geometry into triangle strips.

```
Stripifier st = new Stripifier();
st.stripify(gi);
```

Triangle strips are sequences of triangles where the second and third vertices of one triangle are used as the first and second vertices of the next triangle, as in Figure 6.

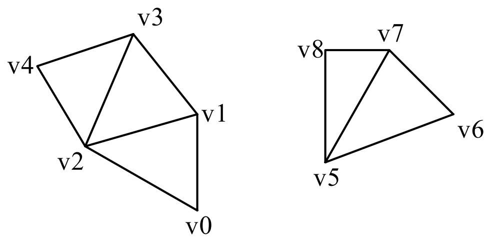


Figure 6. Triangle Strips Examples.

The reordering of the points should allow the underlying graphics hardware to render the shape more quickly. However, the speed-up depends on the geometry being made up of adjacent quads, which will vary depending on the heights assigned by FractalMesh.

Finally, the geometry inside the GeometryInfo object is made the geometry for the TexturedPlanes object:

```
// extract and use GeometryArray
setGeometry( gi.getGeometryArray() );
```

5.1. Texture Minification

The Appearance node for the TexturedPlanes object is the usual mix of Texture2D and Material nodes, with lighting enabled. However, the size of the landscape will mean that textures will appear on quads at a great distance from the viewpoint. In terms of rendering, this will require the texture image to be reduced in size, with a consequently reduction in its resolution. As the viewpoint moves, the quads and textures will be redrawn, and the texture reduction will change, perhaps resulting in a different mix of colours assigned to the quad. This change is seen by the user as a 'shimmering', which can become very annoying in a highly textured environment (as here).

The reduction of a texture to match a smaller pixel area on screen is called *minification*. (You may recall that we dealt with texel magnification in the FPShooter3D application of chapter 15.)

The solution is to set a minification filter using the setMinFilter() method, such as:

```
texture.setMinFilter(Texture2D.BASE_LEVEL_LINEAR);
```

This causes the pixel colouring to based on an average of the four nearest texels in the texture. It deals with shimmering, but only until the texture gets small enough so that 4 or more texels are mapped to a pixel.

A better solution is to use *mipmapping*: where several lower resolution versions of the texture are pre-computed as the texture is loaded, and then used as needed at run-time. The relevant code:

```
// load and set the texture; generate mipmaps for it
TextureLoader loader = new TextureLoader(fnm,
                 TextureLoader.GENERATE_MIPMAP, null);
```

```
  Texture2D texture = (Texture2D) loader.getTexture();
  texture.setMinFilter(Texture2D.MULTI_LEVEL_LINEAR);
```

The GENERATE_MIPMAP flag switches on mipmapping, and the MULTI_LEVEL_LINEAR mode specifies that the colour for a pixel comes from eight texels: the four closest texels from each of the two closest texture resolutions.

This approach removes shimmering, and replaces the need for run-time scaling of the texture.

6. FractalMesh

FractalMesh uses a plasma fractal to generate a mesh of Point3d objects, centered at (0,0) on the (x,z) plane, at intervals of 1 unit, extending out to WORLD_LEN/2 units in the positive and negative x- and z- directions. (In our code, WORLD_LEN is 64 units.) The y coordinates of these points become their heights in the scene.

The objects are stored in a 2D array called mesh, with mesh[0][0] storing the back, left-most point in the scene. A row in mesh[][] stores all the points for a given z-value

The mesh is generated using the algorithm described in:

> "Fractal Terrain Generation - Midpoint Displacement"
> Jason Shankel,
> In *Game Programming Gems*, section 4.18, pp.503-507
> Mark Deloura (ed.), Charles River Media, 2000

(You may recall that we used this same article as the basis of our line fractal code in chapter ??.)

The mesh is first seeded with four corner points, and then a two-stage process is repeated until sufficient extra points have been created. In the first stage (the *diamond step*), the height of the midpoint of the four corner points is calculated by averaging their heights and adding a random displacement in the range -dHeight/2 to dHeight/2. For example, the height of the E point in Figure 7 is calculated like so:

$$E = (A + B + C + D)/4 + \text{random}(-dHeight/2, dHeight/2)$$

The next stage (the *square step*) is to calculate the heights of the midpoints of the 4 sides (F, G, H, and I in Figure 7). For example, G's height is:

$$G = (A + E + C + E)/4 + \text{random}(-dHeight/2, dHeight/2)$$

Note that if a point is on an edge (as G is) then we use a neighbour from the opposite edge by thinking of the mesh as wrapping around from left-to-right, and top-to-bottom. That is why G's calculation uses E twice: once as the left neighbour of G, once as its right neighbour.

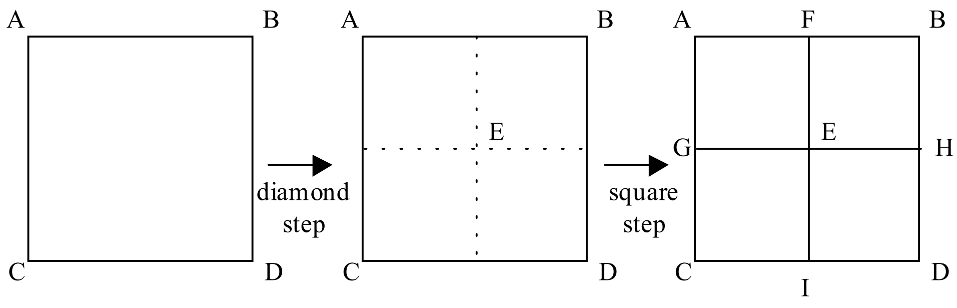


Figure 7. Mesh Creation: First Iteration.

By the end of the two stages, the mesh can be viewed as 4 quarter-size squares (AFEG, FBHE, GEIC, EHDI). Now the process begins again, on each of the 4 smaller squares, as shown in Figure 8. The difference is that the sides of the squares are half the length of the original. Also, dHeight is divided by the flatness value (the number entered by the user).

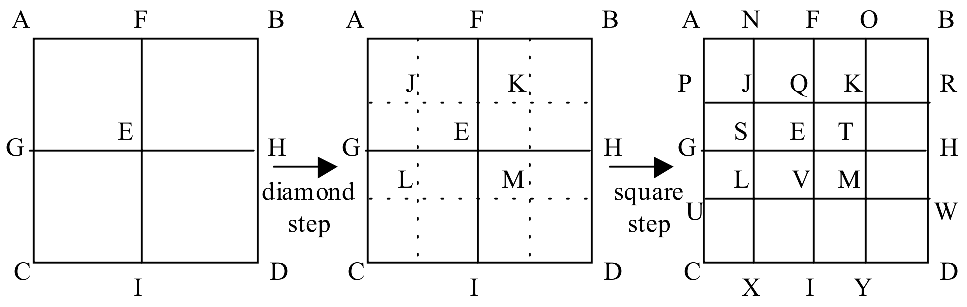


Figure 8. Mesh Creation: Second Iteration.

When flatness is > 2, dHeight will decrease faster than the sides of the squares, so that after the initial creation of hills and valleys, the rest of the terrain will generally consist of smooth slopes between those features. When flatness < 2, the randomness will be a significant component of the height calculations for longer. This will mean that hills and valleys will be created for longer, resulting in a rockier landscape.

The creation of the corner points is done by makeMesh():

```
private void makeMesh()
{
  System.out.println("Building the landscape...please wait");
  mesh[0][0] =    // back left
    new Point3d( -WORLD_LEN/2, randomHeight(), -WORLD_LEN/2 );

  mesh[0][WORLD_LEN] =    // back right
    new Point3d( WORLD_LEN/2, randomHeight(), -WORLD_LEN/2 );
  mesh[WORLD_LEN][0] =    // front left
    new Point3d( -WORLD_LEN/2, randomHeight(), WORLD_LEN/2 );

  mesh[WORLD_LEN][WORLD_LEN] =    // front right
    new Point3d( WORLD_LEN/2, randomHeight(), WORLD_LEN/2 );

  divideMesh( (MAX_HEIGHT-MIN_HEIGHT)/flatness, WORLD_LEN/2);
}
```

randomHeight() selects a random number between the maximum and minimum heights fixed in the class.

divideMesh() carries out the diamond and square steps outlined above, and continues the process by recursively calling itself. The code in outline:

```
  private void divideMesh(double dHeight, int stepSize)
  {
    if (stepSize >= 1) {   // stop recursing once stepSize is < 1
      // diamond step for all midpoints at this level
      // square step for all points surrounding diamonds
      divideMesh(dHeight/flatness, stepSize/2);
    }
  }
```

divideMesh() keeps dividing stepSize by 2 until it reaches 1 unit, and it starts with the value WORLD_LEN/2. In order for the points to be equally spaced over the XZ plane, WORLD_LEN should be a power of 2 (it is 64 in our code).

divideMesh() stores the generated points as Point3d objects in the mesh[][] array.

The Landscape object accesses the points by calling FractalMesh's getVertices() method. getVertices() creates a 1D array called vertices[] and stores references to mesh[][]'s points in counter-clockwise quad order, starting with the bottom left corner of the quad. For instance, when considering coordinate (x, z), it will copy the points in the order (x, z+1), (x+1, z+1), (x+1, z), (x, z). This is somewhat clearer by considering Figure 9.

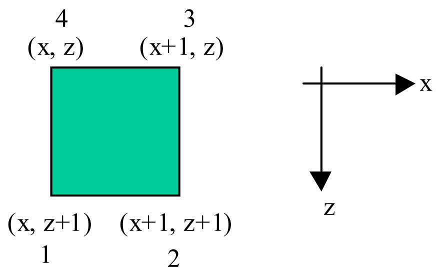


Figure 9. Quad Ordering for Point (x,z).

The getVertices() method:

```
  public Point3d[] getVertices()
  {
    int numVerts = WORLD_LEN*WORLD_LEN*4;
    Point3d vertices[] = new Point3d[numVerts];
    int vPos = 0;
    for(int z=0; z<WORLD_LEN; z++) {
      for(int x=0; x<WORLD_LEN; x++) {
        vertices[vPos++] = mesh[z+1][x];      // counter-clockwise
        vertices[vPos++] = mesh[z+1][x+1];    // from bottom-left
        vertices[vPos++] = mesh[z][x+1];
        vertices[vPos++] = mesh[z][x];
      }
    }
    return vertices;
  }
```

6.1. Printing the Mesh

FractalMesh contains a printMesh() method, which is primarily for debugging purposes: it prints either the x-, y- or z- values stored in mesh[][] to a text file.

This method could be easily extended to store the complete mesh to a file, which would allow FractalMesh to be separated off from FractalLand3D. Landscape would create its floor by reading in the mesh from the file rather than by contacting FractalMesh directly. This is similar to the way that Maze3D in chapter 16 reads its maze information from a file created by the MazeGen application.

The advantage of this approach is that FractalLand3D could choose to reuse an existing landscape instead of having to generate a new one every time it was called.

7. KeyBehavior

KeyBehavior is very similar to the KeyBehavior classes we developed for FPShooter3D in chapter 15 and for Maze3D in chapter 16. A minor difference is that this class is given charge of positioning the viewpoint, which it does by asking the Landscape object for the origin’s coordinates:

```
  private void initViewPosition(TransformGroup steerTG)
  // place viewpoint at (0,?,0), facing into scene
  {
    Vector3d startPosn = land.getOriginVec();
    // startPosn is (0, <height of floor>, 0)

    currLandHeight = startPosn.y;   // store current floor height
    startPosn.y += USER_HEIGHT;     // add user's height

    steerTG.getTransform(t3d);      // targetTG not yet available
    t3d.setTranslation(startPosn);  // so use steerTG
    steerTG.setTransform(t3d);
  }
```

The principal reason for moving the code into KeyBehavior is that it needs to record the current floor height in order to position the viewpoint as it moves.

The operations carried out by KeyBehavior can be grouped into three categories:

1. movements requiring floor height information (i.e. moves forwards, backwards, to the left, and right);

2. movements requiring height offset information (i.e. moves up and down);
3. rotations around the current location (i.e. turns to the left and right).

Rotations do not require floor height information, so are implemented in the usual way as rotations of the ViewPlatform's TransformGroup.

Movements up and down are made more efficient by KeyBehavior storing a zOffset counter which records how many upward moves have been made by the user. Consequently, a move down will only be allowed if zOffset is > 0. The efficiency gain is that there is no need to access floor height information.

Movements over the terrain are implemented by a call to moveBy(), which has three stages:

1. The next (x,z) position on the floor is calculated by carrying out the move but not updating the ViewPlatform's TransformGroup.
2. The resulting (x,z) data is passed to getLandHeight() in the Landscape object so that it can look up the floor height for 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.

The moveBy() method:

```
private void moveBy(Vector3d theMove)
{
  Vector3d nextLoc = tryMove(theMove);   // next (x,?,z) position
  if (!land.inLandscape(nextLoc.x, nextLoc.z))
     return;

  // Landscape returns floor height at (x,z)
  double floorHeight = land.getLandHeight(nextLoc.x, nextLoc.z,
                                          currLandHeight);
  // Calculate the change from the current y-position.
  // Reset any offset upwards back to 0.
  double heightChg = floorHeight - currLandHeight -
                             (MOVE_STEP*zOffset);

  currLandHeight = floorHeight;    // update current height
  zOffset = 0;                     // back on floor, so no offset
  Vector3d actualMove =
          new Vector3d(theMove.x, heightChg, theMove.z);
  doMove(actualMove);
}
```

The method is a little more complicated than the steps above, for two reasons. There is a call to inLandscape() to check if the proposed move will take the user off the floor, in which case the move is ignored. Also, a move always cancels out any existing upward offset, returning the user to the floor.

The actual move is carried out by doMove() which applies the translation to the ViewPlatform's TransformGroup.

8. Other Kinds of Fractal Landscapes

Aside from the plasma fractal used in our code, two other popular approaches are the *fault fractal* and *Fractal Brownian Motion* (FBM).

The fault fractal creates a height map by drawing a random line through a grid, and increasing the height values on one side of the line. If this is repeated several hundred times then a reasonable landscape appear. The main drawbacks are the lack of programmer control over the finished product, and the length of time required to produce a convincing geography.

An FBM fractal is a combination of mathematical noise functions which generate random height values within a certain range. An important advantage of FBM is that each noise function has several parameters which can be adjusted to alter its effect. C++ code for creating clouds and landscapes with FBMs can be found in the article:

> "Programming Fractals"
> Jesse Laeuchi (jesse@laeuchli.com)
> In *Games Programming Gems 2*, section 2.8, pp.239-246
> Mark Deloura (ed.), Charles River Media, 2001

The j3d.org code repository (http://code.j3d.org) has an extensive set of packages related to terrain creation. In org.j3d.geom.terrain, there is a FractalTerrainGenerator class which uses a plasma fractal approach similar to ours – heights are generated, and converted to geometry with ElevationGridGenerator. Then colors are assigned per vertices, based on ramp values specified in ColorRampGenerator.

The use of colours at the vertices produces a pleasant blending effect. Unfortunately, it is not possible to combine texture mappings and coloured nodes in Java 3D, due to restrictions in the underlying OpenGL and DirectX systems.

The org.j3d.geom.terrain package also contains classes for creating height maps from images. The org.j3d.loaders package supports the loading of grid-aligned data from files, which can be treated as heights.

9. Issues with our Terrain Representation

The Landscape class groups quads with similar heights together in the same Shape3D, so they all share the same Appearance node component. The application only has a few Shape3D nodes at run-time, and normal generation and stripifying are only applied to a few, large geometries.

However, this approach has some potential drawbacks, especially if the terrain size is increased. One issue is view frustum culling, where Java 3D renders only what can be seen in the user's field of view. The culling is based on bounds calculations for the geometries, and our Shape3Ds typically contain quads spread all over the landscape. This means that very little culling can be achieved in most situations.

Another problem is the cost of picking, which begins by employing bounds checks to exclude Shape3Ds from more detailed intersection calculations. Again, the grouping of dispersed geometries means that these checks may not be able to exclude many shapes.

A drawback of a large geometry in a Shape3D is the increased cost of changing it at run time, which is a common task when using more advanced terrain representations, as explained below.

10. Better Landscape Models

A crucial problem for an ‘industrial-strength’ landscape representation is the enormous amount of memory required for a large map. For example, a height map made up of floats for (x,z) coordinates at 1mm intervals, covering a 10m square area, would require about 400 MB of RAM, and that’s before the cost of rendering and adding objects to the scene. The obvious answer is to reduce the sampling density, but this will also reduce the map’s resolution.

A better solution is *adaptive meshing*, as typified by ROAM (Real-time Optimally Adapting Meshes). It is adaptive in the sense that areas that are flat or distant from the viewpoint are covered by low resolution sub-meshes, while nearby or rocky areas use more detailed grids. Also, since viewpoints can move, the level of detail apparent at a particular terrain location will vary over time, as the user comes closer and moves away.

ROAM creates its dynamic mesh with triangle bintrees (a close relative of the quadtree), which are split/merged to increase/decrease the resolution. The standard reference on ROAM is the paper:

> “ROAMing Terrain: Real-time Optimally Adapting Meshes”
> Mark Duchaineau, et al.
> In *Proc. IEEE Visualization '97*, pp.81-88, 1997
> http://www.llnl.gov/graphics/ROAM/

The good news is that there is a Java 3D implementation of ROAM in the j3d.org code repository package org.j3d.terrain.roam.

Each terrain tile is implemented as a Shape3D, so that view frustum culling and intersection testing can be speeded up. A tile’s geometry is stored in a TriangleArray so that the overhead of translating down to triangles at the hardware level is reduced. Changes to a tile’s geometry is performed with a GeometryUpdater object which recalculates its vertices.

TransformGroups are avoided; instead a tile is positioned at its intended location directly. This means that the shape’s coordinate system does not need to be mapped to world coordinates when being manipulated, making costly transforms unnecessary.

The code is explained in the textbook:

> *Java Media APIs: Cross-Platform Imaging, Media, and Visualization*
> A.Terrazas, J. Ostumi, and M. Barlow
> Sams Publishing, 2002, p.751-800

Incidentally, I recommend this book for its coverage of more advanced Java 3D topics, such as sensors, stereo viewing, head tracking, and JMF integration.

Although our focus is on gaming, terrain creation is also used by simulation and GIS applications. A popular file format for geographic data is the Digital Elevation Model (DEM), which represents grids of regularly spaced elevation values.

The U.S. Geological Survey (USGS) produces five primary types of DEM data: 7.5-minute DEM, 30-minute DEM, 1-degree DEM, 7.5-minute Alaska DEM, and 15-minute Alaska DEM.

Some useful DEM links:

- http://www.cis.ksu.edu/~dha5446/topoweb/demtutorial.html
- http://www.vterrain.org/Elevation/dem.html
- http://edcwww.cr.usgs.gov/products/elevation/dem.html

The j3d.org code repository contains a DEM loader in org.j3d.loaders.dem, but it is poorly documented, and there are no examples of its use. However, the loader is used as part of the GeoSim application, to display DEM files (http://www.bulletprf.com/lab-index.htm). The program comes with source code.

JLand is a Java 3D applet/application for displaying and generating landscapes (http://www.cyberhipster.com/j3d/jland). It can read elevation maps stored as compressed (gzip) or uncompressed PGM, DEM, or POV TGA files. Only the class files are available at the moment.

GameDev.net has a collection of good articles about landscape modeling at:

http://www.gamedev.net/reference/list.asp?categoryid=45#88

11. Multiple Textures

The edges between one texture and another in FractalLand3D are sharply highlighted by the quad geometry, which tends to destroy the illusion of landscaping. One way to improve matters is to use multiple textures for each Shape3D. The idea is to use a basic 'ground' texture and then add variations with additional layers of texturing, such as textures representing different kinds of vegetation, or even forms of lighting and shading.

The texture attributes for multiple layers are specified with TextureUnitState objects, one for each layer. The setTextureUnitState() method is applied to the shape's Appearance node to add the textures. Several texture coordinates can be linked to each vertex of the shape's geometry, and mapped to a particular TextureUnitState. The Java 3D demo program TextureTest/MultiTextureTest.java shows how multitexturing can be utilised.

A possible reason for avoiding multi-texturing is that older graphics card may not be able to support it. The QueryProperties.java utility, in the Java 3D demo directory PackageInfo/, prints out several properties related to textures – multiple textures require that textureUnitStateMax is 2 or more.

12. Terrain Following and Collision Avoidance

Realistic terrain following must handle issues such as gravity effects (e.g. falling off a cliff), how high to step up/down at a time, holes, and water. The code in KeyBehavior and Landscape does not deal with any of these concerns.

A programming style question is whether picking should be used as a walking aid? One reason for employing it is that the same mechanism can also help with movement through scenery, such as up and down staircases, so long as the objects can be picked. Also, picking makes it possible to walk over a terrain without needing a predefined 'map'.

A downside, until recently, was the large amount of garbage that could accumulate over time, as a consequence of repeatedly computing intersections. The resulting garbage collection could seriously degrade the execution of the application. Fortunately, the PickRay and PickSegment intersection code has been rewritten in Java 3D v.1.3.1 to reduce the overhead, but the other picking shapes, such as PickCylinderRay, are unchanged.

Actually, garbage collection may only become a serious issue when picking is also utilised for collision avoidance: the moving object typically sends out multiple rays in several direction at each frame update, each requiring intersection testing. However, this approach is used without problems in the *Pernica* multiplayer role playing game (http://www.starfireresearch.com/pernica/pernica.html) from Starfire Research.

An important coding strategy when employing picking is to divide the terrain into clearly bounded pieces, which will increase the speed of the intersection computations. Unfortunately, compile() can cause divided structures to be re-combined, undoing the careful separation.

An interesting article on terrain following and collision detection in Java 3D, written by Justin Couch, can be found at:

http://www.j3d.orfg/tutorials/collision

GameDev.net has a good collision detection section:

http://www.gamedev.net/reference/list.asp?categoryid=45#99

13. Placing Objects in the Scene

Knowing where to place something in the scene at scene creation time is really a question of knowing the height of the ground at a particular (x,z) location. This is problematic because most ground positions will have been extrapolated by Java 3D from the corner points of quads. Picking is not useful in this case since we want to position objects before the complete scene is made live.

If a quad is coplanar (i.e. positioned in a single plane), then calculating interior points is straightforward. Unfortunately, most quads are not coplanar. Furthermore, because the underlying graphics hardware works with triangles, the geometry in a Shape3D will have been triangulated, and the shape of those triangles is hard to predict when their vertices are highly non-coplanar.

Java 3D uses a subset of the FIST triangulation algorithm – see http://www.cosy.sbg.ac.at/~held/projects/triang/triang.html for a non-technical discussion of the complete algorithm. Each polygon is projected onto a plane and the 2D projected points are considered using a range of heuristics. One aim is to avoid long, skinny triangles, by trying to maximize the angles within the triangles. The presence of heuristics means that it is sometimes very difficult to know how a polygon will be divided.

The example is figure 10 shows a single quad with two equally likely triangulations. A division along PR will give point P a height of 0, but if the SQ line is used then the same point will have a height of 5 units.

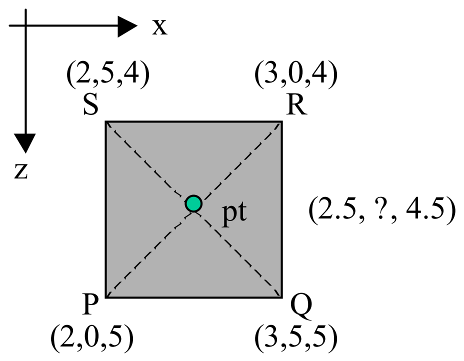


Figure 10. A Triangulation Problem

One way of dealing with this issue is to move away from quadrilaterals and use triangles as the tiling units for the terrain. An equation for the surface delineated by a triangle’s three vertices (the *plane equation*) is easily obtained. Consider the points P, Q, R in the triangle of figure 11.

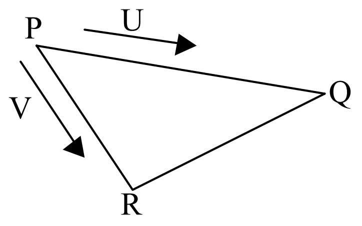


Figure 11. A Triangle

The plane equation is defined as:

Ax + By + Cz = D

where A, B, C, and D are constants.

The normal vector N is calculated with the cross product of the vectors U and V:

N = U x V

U and V are obtained by subtracting P from Q and R respectively.

When is N is normalized (made to have unit length), then A, B, and C are its coefficients.

The distance from the origin to the point on the plane nearest to the origin is equivalent to the plane equation’s D constant. D can be calculated as the dot product of the normal and any point on the triangle (e.g. the point P):

D = N . P

Once we have the plane equation coefficients, the height of any point lying on the triangle is:

y = (D - Ax - Cz) / B

where (x,z) is the known XZ position of the point.

The Java 3D code for doing this is very simple:

```
private void vecsToHeight(Vector3d u, Vector3d v, Vector3d pt)
{
  Vector3d normal = new Vector3d();
  normal.cross(u, v);
  normal.normalize();
  double dist = normal.dot(pt);

  System.out.println("A: "+ df.format(normal.x) + ", B: " +
    df.format(normal.y) + ", C: " + df.format(normal.z) +
    ", D: " + df.format(dist) );    // Ax + By + Cz = D

  double height = (dist - (normal.x * pt.x) -
                    (normal.z * pt.z)) / normal.y;
  System.out.println("Height for pt: " + df.format(height) );
}
```

A drawback with working with triangles as the terrain’s tiling unit is the distortion apparent in textures laid over the triangles.

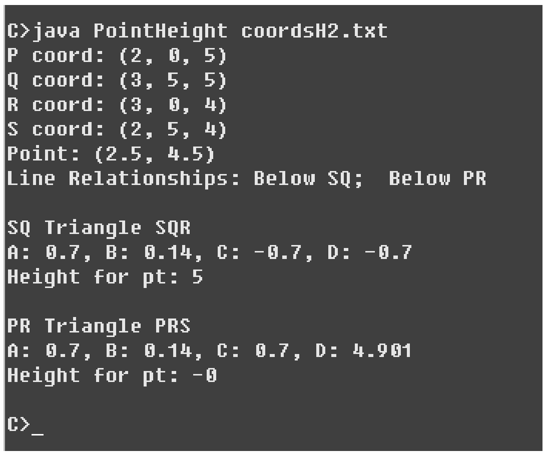
13.1. PointHeight.java

PointHeight.java reads in a file of four coordinates defining a 1 by 1 square in the XZ plane, but with various y-axis heights. The coordinates are labeled P, Q, R, S, starting at the bottom-left corner and going counter-clockwise, when viewed from above (i.e. the same as in Figure 10).

PointHeight also reads in a (x,z) pair that is located somewhere within the quad defined by PQRS.

It then calculates the two possible heights for the coordinate, depending on whether the quad is triangulated around the lines PR or SQ.

Figure 12 shows the output when PointHeight is supplied with the coordinates and point used in Figure 10.



```
C>java PointHeight coordsH2.txt
P coord: (2, 0, 5)
Q coord: (3, 5, 5)
R coord: (3, 0, 4)
S coord: (2, 5, 4)
Point: (2.5, 4.5)
Line Relationships: Below SQ;  Below PR

SQ Triangle SQR
A: 0.7, B: 0.14, C: -0.7, D: -0.7
Height for pt: 5

PR Triangle PRS
A: 0.7, B: 0.14, C: 0.7, D: 4.901
Height for pt: -0

C>_
```

Figure 12. PointHeight output.

The output shows the two possibilities. Now the user must choose which one to use.

PointHeight is not used in FractalLand3D since each invocation causes a new mesh to be generated. However, this utility would be useful if fractal generation was a separate element from FractalLand3D, and the mesh was read in at start-time and scenery added at that time.

The code for PointHeight can be found in the subdirectory of "FractalLand3D/Plane Equation".