Chapter 16. A 3D Maze

We continue the First Person Shooter (FPS) theme of the previous chapter, but without the shooting! Instead the emphasis is on navigation through a complex scene (a 3D maze). The two main topics will be how to generate a realistic looking scene (realistic in the "Doom" or "Quake" sense), and how to use multiple views to aid navigation.

Figure 1 shows a screenshot of the application, Maze3D, which consists of three JPanels. The left hand side shows the user's viewpoint facing forwards (we sometimes call this the *main camera*), the top right hand panel is the view behind the user (the *back facing camera*), and the bottom right hand panel is a schematic overview of the entire maze with the user represented by an arrow pointing in his/her current forward direction. We call this the *bird's eye view*.

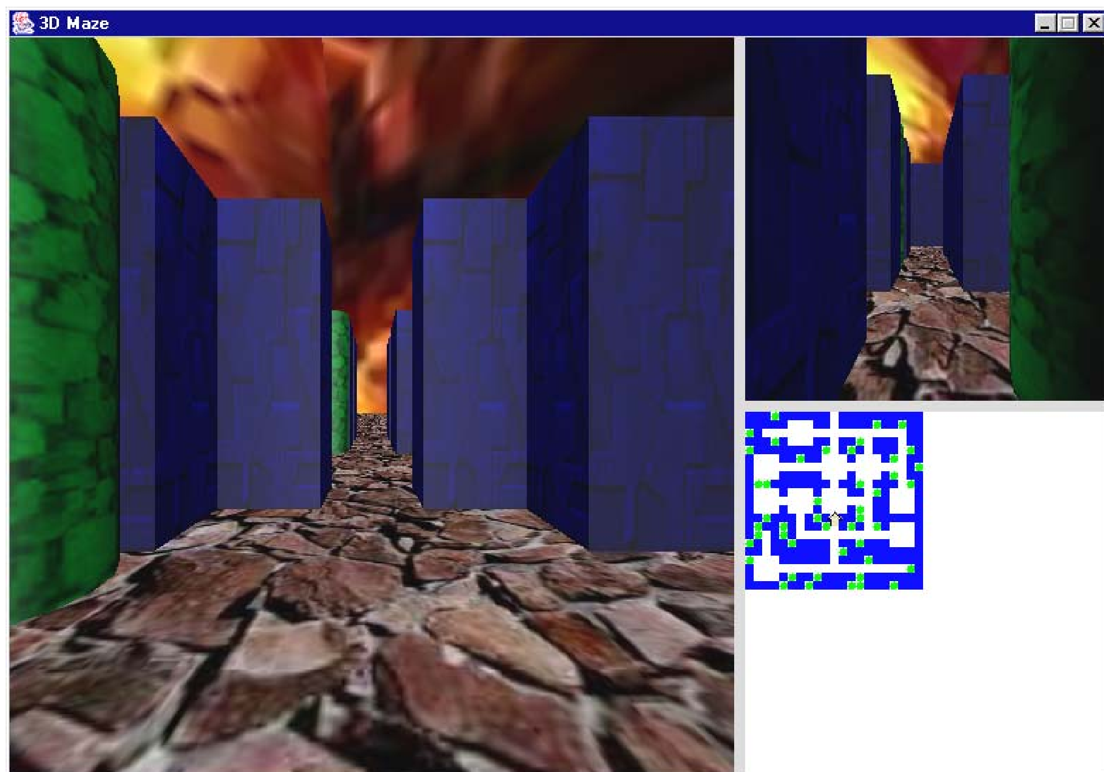

Figure 1. Navigating the 3D Maze.

As in the last chapter, navigation is controlled from the keyboard: the user can move forwards, backwards, left, right, up, or down, but can not move through walls or the floor. Rotation is limited to 90 degree turns to the left or right.

The maze layout (or plan) is generated by a separate application, which stores it as an ASCII file, and read in by Maze3D at start-time.

Other features:

- the floor, the maze's blocks, and cylinders all have textured and coloured surfaces;

 © Andrew Davison. 2003

- the background is a textured sphere;
- the user's viewpoint includes a forward-facing spotlight since there is no ambient light in the scene;
- certain attributes of the viewpoint are changed, including its field of view (FOV) and the forward and back clip distances;
- the back-facing camera is implemented as an additional view branch graph in the scene graph;
- the code includes an example method that adds a forward-facing cone to the viewpoint to act as an avatar, although we do not need to use it here.

The code can be found in /Maze3D.

1. UML Diagrams for Maze3D

Figure 2 gives the UML diagrams for all the classes in the Maze3D application, including the class names and public methods.

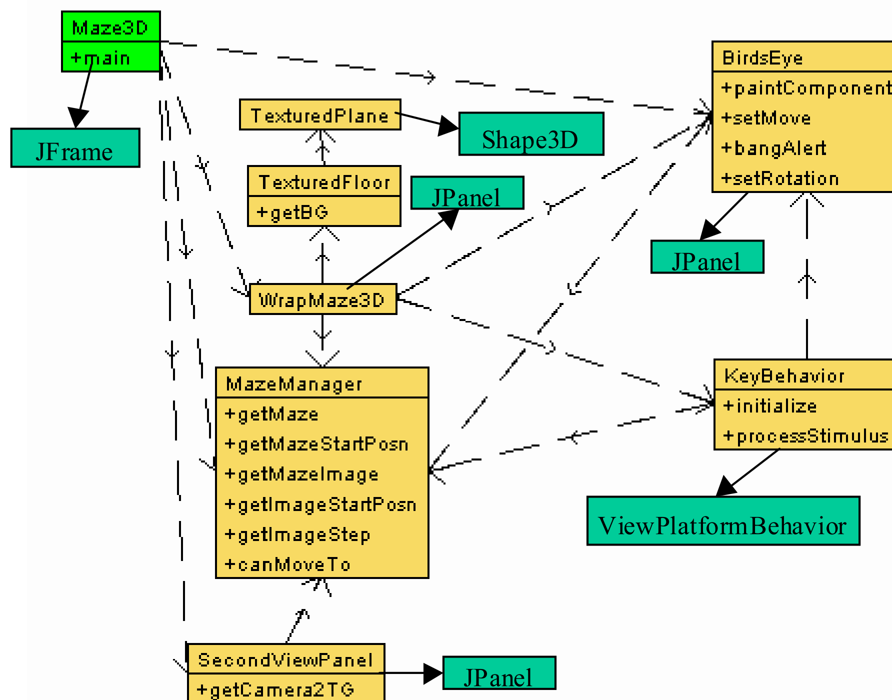

Figure 2. UML Diagram for Maze3D.

Maze3D is the top-level JFrame, and does a bit more than in previous examples: it creates several objects and builds the GUI interface.

WrapMaze3D is a JPanel that creates the scene, including the background, lighting, and the main camera viewpoint. It utilises maze and floor subgraphs made by other objects, and invokes a KeyBehavior object.

The TexturedFloor and TexturedPlane classes are used to create the floor.

 © Andrew Davison. 2003

MazeManager reads in the maze plan (a text file) prepared outside of Maze3D, and generates two representations: a Java 3D subgraph which is added to the scene, and a Java 2D image of the maze which is passed to the BirdsEye object.

BirdsEye draws an overview of the maze by moving and rotating an arrow over the top of the maze image. This bird's eye view is displayed in the bottom right hand JPanel in the GUI.

SecondViewPanel creates a second view branch subgraph which shows the view behind the user's current position (the back facing camera). The subgraph is added to the main scene, and its Canvas3D object linked to the top right hand JPanel in the GUI.

KeyBehavior converts key presses into moves and rotations of the two cameras and updates to the bird's eye view.

2. Making a Maze Plan

The 3D and 2D maze representations employed in Maze3D are created by the MazeManager object after reading in a maze plan from a text file. The plan has a simple format, as shown in Figure 3 which shows the maze plan in maze1.txt.

```
                s
bbbbb bbbbb bbbbb bbbbb
b                     b
b                     b
bbbb  ccccc ccccc  bbbb
   b
   b
   b
```

Figure 3. The maze1.txt Maze Plan.

 © Andrew Davison. 2003

Figure 4 shows that plan utilised by Maze3D (via the call `java Maze3D maze1.txt`):

Figure 4. Maze3D using maze1.txt

The 's' character in the plan specifies the user's starting position. By default, the viewpoint is set to point along the positive z-axis in the scene which corresponds to straight down in the bird's eye view.

The 'b' characters in the maze plan become blue textured blocks in the scene, and the 'c' characters are drawn as green textured cylinders.

The maze plan can be prepared in a variety of ways. One approach is to use the MazeMaker application described in chapter ??. As an alternative, one of my students, Mr. Nawapoom Lohajarernvanich, and I wrote a new maze generation application, called MazeGen.java. It utilises recursive, depth first search with backtracking to create the maze.

 © Andrew Davison. 2003

The algorithm generates a maze inside a 2D dimensional character array. It assumes the array has an even number of rows and columns, and creates the outer walls of the maze offset by one cell from the left, right, top, and bottom (see Figure 5). This means that the maze boundaries are in the odd rows and columns of the array.

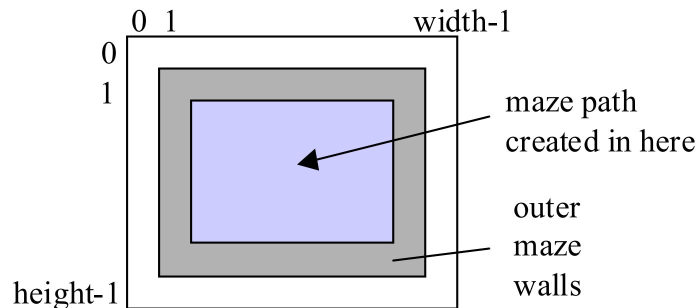

Figure 5. The Maze's Outer Walls.

A path is made up of connected array cells. The first cell is randomly selected at an even coordinate in the char array (e.g. (4,4), (6,8)), and the next cell is randomly chosen from the four cells that are 2 units away in the x- or y- directions. For example, if the current cell is (4,4), then the next cell could be (2,4), (6,4), (4,6), or (4,2). The path is made by connecting the cells. This process is illustrated in Figure 6.

The path-making process is repeated at the new cell, except that a path cannot be made to a cell that has already been used, or to a position on or outside the maze's outer walls. For example, from (6,4) the algorithm can use (8,4), (6,6), and (6,2), but not (4,4).

The algorithm continues until it reaches a cell which cannot be connected to any further cells. This occurs when all the possible next cells have already been utilised or are on or outside the outer walls. At this stage, the algorithm backtracks to the previous cell and looks at it again. The backtracking will continue until a cell has a possible next cell or there is no previous cell to go to (i.e. there is nothing before the starting cell (4,4)).

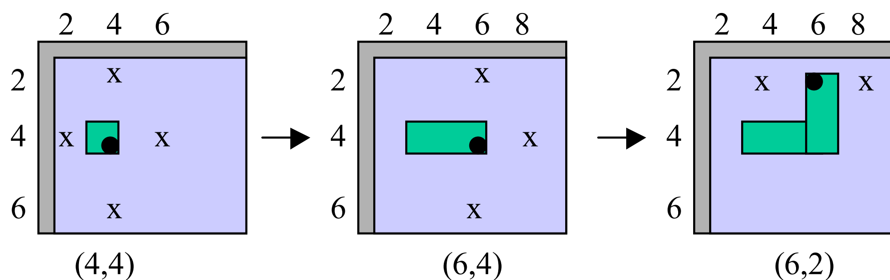

Figure 6. Path Generation in MazeGen.

The relevant code inside MazeGen is contained in createPath():

```
private void createPath(Point cell)
{ RandomInRange rr = new RandomInRange(NUM_DIRS);
  int dir;
  while((dir = rr.getNumber()) != -1){
```


© Andrew Davison. 2003

```
      Point nextCell = nextPos(cell, dir);
      if( !beenVisited(nextCell) ){
        visit(nextCell, dir);
        createPath(nextCell);  // recursive creation
      }
    }
  }
```

The RandomInRange object performs the same role as the one in MazeMaker from chapter ??. Its constructor creates a sequence between 0 and the supplied value-1, and each call to getNumber() randomly selects a number from that sequence.

createPath() passes the number to nextPos() to select the next cell. If the next cell has not been visited then visit() extends the path and the recursive call to createPath() continues path generation. Backtracking from the recursive call will eventually return to this loop, which will try another direction. getNumber() returns -1 when its sequence is exhausted, causing createPath() to return, perhaps to an earlier createPath () call.

The path is remembered by visit() side-effecting a global maze[][] array, so that backtracking will not ‘undo’ path creation.

A typical call to MazeGen is given in Figure 7. The output shows that the maze uses the entire area and doesn’t suffer from closed-off sub-areas which may occur with MazeMaker. The program randomly changes about 20% of the ‘b’s to ‘c’s and adds an entrance to the top row of the maze. The starting point used for the path creation is assigned an ‘s’ to become the starting point for the user.

```
D>java MazeGen 21 13
Width: 23; Height: 15 (2 added for borders)
Saving maze to file: maze.txt
Starting point: (4, 2)

 bbbbbcbbcbc cccbbbbcb
 c       b b         b
 c bbbbb c b b bbbbb c
 bsc   c b   b   c b b
 bbb c b b bbbcb c b b
 b   b b b c     b b b
 b cbc c bcb bbbbb b b
 b c   b b   b     b b
 c bbbbb b bbbcb bbb b
 b b   b   b     c   b
 b c b bbbbb bcbbb bcb
 b   b               b
 bbbbbbbbcbbbbccbbbbcc

Maze written to file maze.txt

D>
```

Figure 7. Sample Output from MazeGen.

 © Andrew Davison. 2003

3. Maze3D

Maze3D invokes MazeManager, BirdsEye, SecondViewPanel, and WrapMaze3D objects. The latter three are subclasses of JPanel, and are then organised into a GUI using the layout shown in Figure 8.

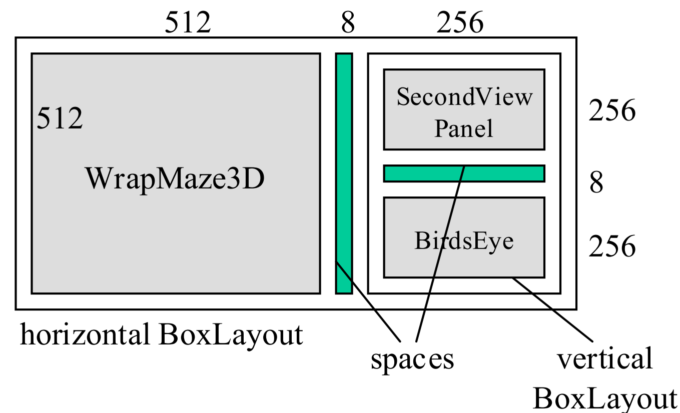

Figure 8. Maze3D GUI Layout.

The fragment of code below contains the invocation of the four objects, and shows that MazeManager is required by all the JPanel objects.

```
MazeManager mm = new MazeManager(fnm);   //fnm holds maze plan
BirdsEye be = new BirdsEye(mm);
SecondViewPanel secondVP = new SecondViewPanel(mm);
WrapMaze3D w3d = new WrapMaze3D(mm, be, secondVP.getCamera2TG() );
```

4. The MazeManager

MazeManager reads in a maze plan and generates two maze representations: a 3D subgraph added to the scene by WrapMaze3D, and a 2D image employed by BirdsEye as the background for its moving arrow.

The primary aim of MazeManager is to hide the processing of the maze plan from the rest of the application. MazeManager supplies the two maze representations, the coordinates of the user's starting point, and deals with collision detection.

MazeManager can be seen as a more complicated version of the Obstacles class in Tour3D in chapter 10. Obstacles creates 3D obstacles for the scene (cylinders) and checks whether the sprites would collide with them as they move.

MazeManager's readFile() method fills in a maze[][] array, and buildMazeReps() creates the maze representations.

When buildMazeReps() creates the 3D maze, it assumes that the rows of maze[][] are positive z-axis values and the columns positive x-axis values. For example, maze[3][5] refers to the coordinate (5,0,3). This means that, when the resulting 3D maze is rendered in the scene, the top-left corner of the maze plan is located at the origin and the maze stretches to the right and out of the scene, through the positive XZ quadrant.

 © Andrew Davison. 2003

When buildMazeReps() creates the 2D image, it continues to treat the columns of maze[][] as positive x-axis numbers, but the rows are viewed as *positive y-axis* values, as in Java 2D. In other words, the origin of the y-axis starts at the top-left of the image and the y values increase *down* the image. For instance, maze[3][5] refers to the coordinate (5,3) in the image.

The 3D scene is a single BranchGroup with TransformGroups hanging off it. There is one TransformGroup for each block (Box node) and cylinder (Cylinder node), to place the shape at its given coordinate. A single TransformGroup is created with a call to makeObs():

```
private TransformGroup makeObs(char ch,int x,int z,Appearance app)
// place an obstacle (block/cylinder) at (x,z) with appearance app
{
  Primitive obs;
  if (ch == 'b')  // blue textured block
    obs = new Box(RADIUS, HEIGHT/2, RADIUS,
                      Primitive.GENERATE_TEXTURE_COORDS |
                      Primitive.GENERATE_NORMALS, app );
  else   // green textured cylinder
    obs = new Cylinder(RADIUS, HEIGHT,
                      Primitive.GENERATE_TEXTURE_COORDS |
                      Primitive.GENERATE_NORMALS, app );

  // position obstacle so its base is resting on the floor at (x,z)
  TransformGroup posnTG = new TransformGroup();
  Transform3D trans = new Transform3D();
  trans.setTranslation( new Vector3d(x, HEIGHT/2, z) );  // move up
  posnTG.setTransform(trans);
  posnTG.addChild(obs);
  return posnTG;
}
```

Care must be taken to reduce the overhead of using textured surfaces. This is achieved by reusing two pre-calculated Appearance nodes: one for the blocks, one for the cylinders, created with calls to makeApp():

```
private Appearance makeApp(Color3f colObs, String texFnm)
{
  Appearance app = new Appearance();

  // mix the texture and the material colour
  TextureAttributes ta = new TextureAttributes();
  ta.setTextureMode(TextureAttributes.MODULATE);
  app.setTextureAttributes(ta);

  // load and set the texture
  System.out.println("Loading obstacle texture from " + texFnm);
  TextureLoader loader = new TextureLoader(texFnm, null);
  Texture2D texture = (Texture2D) loader.getTexture();
  app.setTexture(texture);      // set the texture

  // add a coloured material
  Material mat = new Material(colObs,black,colObs,specular,20.f);
  mat.setLightingEnable(true);
  app.setMaterial(mat);
  return app;
```

 © Andrew Davison. 2003

```
  }
```

The Appearance modulates the texture and material, and switches on lighting effects.

The intention is that the scene will be poorly lit except for a spotlight 'held' by the user. The efficacy of the spot depends on its parameters, such as its attenuation and concentration, but also on the shininess and specular colour of the objects, which are set in the Material node component.

The 2D image is initialised as a BufferedImage:

```
BufferedImage  mazeImg = new BufferedImage(IMAGE_LEN, IMAGE_LEN,
                         BufferedImage.TYPE_INT_ARGB);
```

The drawing operations are applied via a graphics context:

```
    Graphics g = (Graphics) mazeImg.createGraphics();
        :
    drawBlock(g, x, z);       // for a block at (x,y)
        :
    drawCylinder(g, x, z);    // for a cylinder at (x,y)
        :
    g.dispose();   // when drawing is completed
```

drawBlock() and drawCylinder():

```
  private void drawBlock(Graphics g, int i, int j)
  // draw a blue box in the 2D image
  { g.setColor(Color.blue);
    g.fillRect(i*IMAGE_STEP, j*IMAGE_STEP, IMAGE_STEP, IMAGE_STEP);
  }

  private void drawCylinder(Graphics g, int i, int j)
  // draw a green circle in the 2D image
  { g.setColor(Color.green);
    g.fillOval(i*IMAGE_STEP, j*IMAGE_STEP, IMAGE_STEP, IMAGE_STEP);
  }
```

IMAGE_STEP is the number of pixels corresponding to the size of a cell in the maze.

4.1. Collision Detection

Collision detection is a matter of testing a supplied (x,z) pair against the maze[z][x] cell to see if it contains a 'b' or 'c'. Coordinates beyond the maze's extent must also be dealt with.

```
  public boolean canMoveTo(double xWorld, double zWorld)
  // is (xWorld, zWorld) free of obstacles?
  // Called by the KeyBehavior object to test a possible move.
  {
    int x = (int) Math.round(xWorld);
    int z = (int) Math.round(zWorld);

    if ((x < 0) || (x >= LEN) || (z < 0) || (z >= LEN))
```

 © Andrew Davison. 2003

```
      return true;    // since outside the possible maze dimensions

    if ((maze[z][x] == 'b') || (maze[z][x] == 'c'))
      return false;    // since loc occupied by block or cylinder

    return true;
  }  // end of canMoveTo()
```

The supplied coordinates should be integer values since the user only moves in steps of 1 unit and rotates by 90 degrees. However, Java 3D transforms utilise floats or doubles, and so the coordinates will never be exactly whole, which means they must be rounded before use.

5. Scenery Creation in WrapMaze3D

One way of summarizing WrapMaze3D is to say that it carries out two main tasks: the creation of scene objects (e.g. the floor, the maze, lighting, background), and the initialisation of the viewpoint (e.g. its position, orientation, and geometries linked to the viewpoint). In this section, we consider scenery creation.

createSceneGraph() builds the scene:

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

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

    // add the textured floor
    TexturedFloor floor = new TexturedFloor();
    sceneBG.addChild( floor.getBG() );

    sceneBG.addChild( mazeMan.getMaze() );
                        // add 3D maze, using MazeManager
    sceneBG.addChild( camera2TG );      // add second camera

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

lightScene() is very similar to previous examples in that it switches on two downward facing lights. However, no ambient light is created, causing the maze's internal walls to be cast into darkness.

5.1. Making a Background

WrapMaze3D contains two versions of addBackground(), both of which lay a texture over the inside face of a Sphere. The first version makes the Sphere a child of a Background node:

 © Andrew Davison. 2003

```
private void addBackground()
// add a geometric background using a Background node
{
  System.out.println("Loading sky texture: " + SKY_TEX);
  TextureLoader tex = new TextureLoader(SKY_TEX, null);

  Sphere sphere = new Sphere(1.0f, Sphere.GENERATE_NORMALS |
          Sphere.GENERATE_NORMALS_INWARD |
          Sphere.GENERATE_TEXTURE_COORDS, 4);   // default = 15
  Appearance backApp = sphere.getAppearance();
  backApp.setTexture( tex.getTexture() );

  BranchGroup backBG = new BranchGroup();
  backBG.addChild(sphere);

  Background bg = new Background();
  bg.setApplicationBounds(bounds);
  bg.setGeometry(backBG);

  sceneBG.addChild(bg);
}
```

The initialisation of the Sphere node has some unusual elements: it's radius is set to 1.0f, but since it becomes a Background element, it will always appear behind the other scenery. The Sphere is set to create inward normals which will force the texture to appear on its inside face -- the one visible from within the scene.

The number of divisions, which controls the number of surfaces that make up the sphere, is reduced from 15 (the default) to 4. The number of surfaces is equal to the square of the number of divisions. This reduction makes little difference to the quality of the background, but greatly reduces the cost of generating the sphere.

Another example using this technique can be found in /Background in the Java 3D demos directory.

Unfortunately, there are some bugs connected with geometric Backgrounds in the current version of Java 3D (v.1.3.1). Also some (older) graphics cards find it hard to display them. An alternative is to render an inward facing sphere without using Background:

```
private void addBackground()
{
  System.out.println("Loading sky texture: " + SKY_TEX);
  TextureLoader tex = new TextureLoader(SKY_TEX, null);

  // create an appearance and assign the texture
  Appearance app =  new Appearance();
  app.setTexture( tex.getTexture() );

  Sphere sphere = new Sphere(100.0f,    // radius to edge of scene
          Sphere.GENERATE_NORMALS_INWARD |
          Sphere.GENERATE_TEXTURE_COORDS, 4, app);

  sceneBG.addChild( sphere );
}
```

 © Andrew Davison. 2003

The Sphere is made as large as the intended radius of the scene, and added to the scene directly.

6. The Floor

The floor is a QuadArray made up of a series of quads which taken together cover the entire floor. Each quad is assigned a texture and an upward facing normal. The floor has sides FLOOR_LEN, and each quad has sides STEP, as shown in Figure 9.

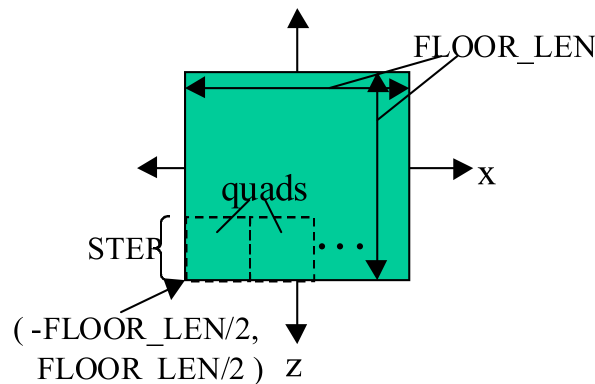

Figure 9. The Floor and its QuadArray (from above).

The paving of the floor starts at the front, left-most point (`-FLOOR_LEN/2`, `FLOOR_LEN/2`) and continues left-to-right, forward-to-back. This is done by TexturedFloor's constructor and createCoords():

```
public TexturedFloor()
// create quad coords, make TexturedPlane, add to floorBG
{
  ArrayList coords = new ArrayList();
  floorBG = new BranchGroup();

  // create coords for the quad
  for(int z=FLOOR_LEN/2; z >= (-FLOOR_LEN/2)+STEP; z-=STEP) {
    for(int x=-FLOOR_LEN/2; x <= (FLOOR_LEN/2)-STEP; x+=STEP)
        createCoords(x, z, coords);
  }

  Vector3f upNormal = new Vector3f(0.0f, 1.0f, 0.0f);   // upwards
  floorBG.addChild( new TexturedPlane(coords,FLOOR_IMG,upNormal) );
}


private void createCoords(int x, int z, ArrayList coords)
{
  // points created in counter-clockwise order, from front left
  // of length STEP
  Point3f p1 = new Point3f(x, 0.0f, z);
  Point3f p2 = new Point3f(x+STEP, 0.0f, z);
  Point3f p3 = new Point3f(x+STEP, 0.0f, z-STEP);
  Point3f p4 = new Point3f(x, 0.0f, z-STEP);
  coords.add(p1); coords.add(p2);
  coords.add(p3); coords.add(p4);
```

 © Andrew Davison. 2003

```
  }  // end of createCoords()
```

createCoords() takes a single point and creates the four points for a quad, making sure they are in counter-clockwise order to place the front face upwards on the XZ plane.

Once all the coordinates have been calculated, they are passed to a TexturedPlane object, along with the name of a texture file (TEXTURE_IMG) and a vector that will become each quad's normal.

6.1. TexturedPlane

The QuadArray created by TexturedPlane uses textures and normals:

```
QuadArray plane  = new QuadArray(numPoints,
                        GeometryArray.COORDINATES |
                        GeometryArray.TEXTURE_COORDINATE_2 |
                        GeometryArray.NORMALS  );
```

The hardest part of the geometry-related code is setting the texture coordinates. The coordinates in the QuadArray are ordered in groups of four, each specifying a quad in counter-clockwise order. The texture coords are created in the same way: in groups of four in counter-clockwise order:

```
    // assign texture coords to each quad
    // counter-clockwise, from bottom left
    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);
    }
    plane.setTextureCoordinates(0, 0, tcoords);
```

The Appearance part of the shape creates a modulated texture with a Material node component so that lighting can be enabled. The code is very similar to makeApp() in MazeManager.

The end result is that the texture will be tiled across the floor, with each tile having sides of length STEP. Although there appears to be many textures, there is only one, but referenced many times.

By varying the STEP constant, the number and size of the tiles can be varied. One point to remember is that STEP should be divisible into the floor length value, FLOOR_LEN.

7. ViewPoint Creation in WrapMaze3D

Various viewpoint manipulations are packaged together in prepareViewPoint():

- the field of view (FOV) is widened;
- the front and back clip distances are adjusted;

 © Andrew Davison. 2003

- the viewpoint position is specified using the maze's starting point;
- the viewpoint is rotated to face along the positive z-axis;
- a forward-pointing spotlight is added to the viewpoint;
- a KeyBehavior object is invoked; it controls the movement of the viewpoint during execution.

In addition, there is an unused method that adds a cone to the viewpoint, to show how to give the viewpoint a visible presence (an avatar).

7.1 Field of View (FOV)

The FOV is the range of angles into the scene that can be seen at any time. The default is 45 degrees, 22.5 degrees on either side of the perpendicular into the scene (see Figure 10).

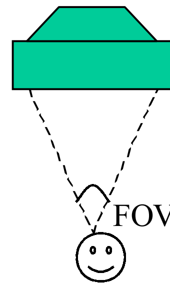

Figure 10. Field of View of a User.

Increasing the FOV permits the user to see more of the sides of the scene, which is very useful in this application when left or right hand corridors need to be spotted. The downside is that a fish-eye effect gradually increases as the FOV is made larger, which distorts the view. Code which changes the FOV to 90 degrees:

```
View userView = su.getViewer().getView();
userView.setFieldOfView( Math.toRadians(90.0)); // wider FOV
```

7.2. Clip Distances

Clip distances specify the closest and farthest objects that can be seen by the viewer. Java 3D's defaults are 0.1 meters for the front clip plane and 10 meters for the back clip, which roughly corresponds to the limitations of human eyesight. Note that these distance are in real world units, although it is possible to adjust the values by employing virtual world units.

The back clip value may often be too small, which can be detected by objects disappearing when the viewpoint moves sufficiently far away from them. The front clip plane may be too large, especially in a FPS, since the viewpoint often gets very close to objects in the scene, which will then be clipped.

The back clip distance can be increased easily, but care must be taken with the front clip value: the temptation is to set it to 0, which would seem to mean that no clipping will occur, no matter how close the user got to a scene node. What actually happens is that the depth ordering of objects in the scene breaks down, with far and near objects partially overlapping each other in arbitrary ways.

 © Andrew Davison. 2003

The reason is to do with the ratio of the back clip to front clip distances. When the back/front value becomes sufficiently large, the machine's hardware (specifically its depth buffer) is being asked to squeeze too large a range of z-values into too few bits per pixel on screen. The critical ratio depends on the bits per pixel used in the depth buffer: older machines may start to 'sweat' at ratios close to 100, but modern cards using 32 bits will still be happy with ratios close to 100,000.

After some experimentation, I went with a back clip distance of 20 meters and a front clip of 0.05m, creating a fairly safe ratio of 400:

```
userView.setBackClipDistance(20);
userView.setFrontClipDistance(0.05);
```

If the front clip distance is left at 0.1m, then when the viewpoint 'hits' a wall, the view is of the block in front of the one hit. This is caused by the back face culling of all the blocks, and the clipping of the front face of the hit block.

7.3. Adding a Spotlight

A spotlight is a light node, and so can be added to the ViewingPlatform via a PlatformGeometry (see Figure 11 for the components of a typical view branch graph).

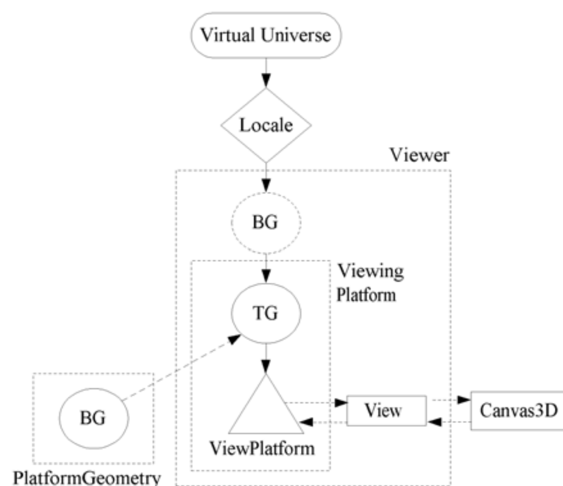

Figure 11. The View Branch Graph.

The construction of the spotlight is done by makeSpot(), embedded in the following code:

```
ViewingPlatform vp = su.getViewingPlatform();
     :
PlatformGeometry pg = new PlatformGeometry();
pg.addChild( makeSpot() );
vp.setPlatformGeometry( pg );
```

The Spotlight node has a position, a direction in which it is pointing, and controls for focussing the light beam called the spread angle and concentration. The spread angle controls the width of the beam – no light is generated outside of the angle. An

 © Andrew Davison. 2003

increased concentration value causes the beam to be focussed more into a narrow beam, although some light will appear beyond the beam's bounds.

The default spread angle is 180 degrees; the default concentration is 0.0 which provides uniform light distribution.

Since Spotlight is a subclass of PointLight, it inherits other useful attributes, such as the ability to adjust its attenuation (how quickly the light fades away for objects further away).

A spotlight contributes to diffuse and specular reflections of objects, which depends on the orientation and position of their surfaces, and on their material node values.

In practice, some trial-and-error is required to get a suitable effect, which in our case is faint lighting, quickly crowded out by darkness. The makeSpot() method:

```
private SpotLight makeSpot()
{
  SpotLight spot = new SpotLight();
  spot.setPosition(0.0f, 0.5f, 0.0f);      // a bit above the user
  spot.setAttenuation(0.0f, 1.2f, 0.0f);   // linear attenuation
  spot.setSpreadAngle( (float)Math.toRadians(30.0));  // smaller
  spot.setConcentration(5.0f);       // reduce strength quicker
  spot.setInfluencingBounds(bounds);
  return spot;
}
```

The 30 degree value supplied to setSpreadAngle() corresponds to a spread angle of 60 degrees, 30 degrees on each side of the forward direction. The increased concentration focuses the beam, making the sides of the scene somewhat darker.

The setPosition() value applies only to the SpotLight node, moving it above the viewpoint (and any other geometries added using PlatformGeometry).

7.4. Adding an Avatar

This application does not include an on-screen representation for the user, although an approach using an image laid over a QuadArray was discussed in the last chapter. A more common solution is to add a 3D model (e.g. a hand, a gun). To illustrate the idea, makeAvatar() adds a cone at the viewpoint, rotated by 90 degrees about the x-axis so its apex is pointing forwards.

```
private TransformGroup makeAvatar()
{
  Transform3D t3d = new Transform3D();
  t3d.rotX(Math.PI/2);    // rotate so top of cone is facing front
  TransformGroup userTG = new TransformGroup(t3d);

  userTG.addChild( new Cone(0.35f, 1.0f) );  // a thin cone
  return userTG;
}
```

The TransformGroup is linked to the PlatformGeometry like so:

```
pg.addChild( makeAvatar() );
```

 © Andrew Davison. 2003

This line could appear in the code fragment on the previous page where the spotlight is added to the PlatformGeometry.

7.5. Positioning the Viewpoint

Setting up the position and orientation of the viewpoint requires the TransformGroup above the ViewPlatform to be accessed and changed:

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

steerTG is the TransformGroup above the ViewPlatform node in Figure 11. By default, the initial viewpoint is facing into the scene, along the -z axis. initViewPosition() rotates it by 180 degrees and moves it to the maze's start position.

```
private void initViewPosition(TransformGroup steerTG)
{
  Transform3D t3d = new Transform3D();
  steerTG.getTransform(t3d);
  Transform3D toRot = new Transform3D();
  toRot.rotY(-Math.PI);    // so facing along positive z-axis

  t3d.mul(toRot);
  t3d.setTranslation( mazeMan.getMazeStartPosn() );
  steerTG.setTransform(t3d);
}
```

Since the rotation orientates the viewpoint in the positive z-axis direction, the translation value does not need to be adjusted. More generally, a translation will be affected by an earlier rotation since the coordinate system is affected.

In most code, viewpoint positioning is more easily achieved with lookAt(), as illustrated below in SecondViewPanel.

7.6. Keyboard Controls

The keyboard behaviour is set up inside prepareViewPoint() in WrapMaze3D:

```
ViewingPlatform vp = su.getViewingPlatform();
  :
KeyBehavior keybeh = new KeyBehavior(mazeMan, be, camera2TG);
keybeh.setSchedulingBounds(bounds);
vp.setViewPlatformBehavior(keybeh);
```

The KeyBehavior class is a subclass of ViewPlatformBehavior and so has access to the TransformGroup above the ViewPlatform (targetTG) without the programmer having to pass it to the constructor. The KeyBehavior class is explained below.

 © Andrew Davison. 2003

8. The Back Facing Camera

The back facing camera is a second ViewPlatform node, which requires its own Canvas3D object so that the view can be displayed. The minimal subgraph which does all of this is shown in Figure 12.

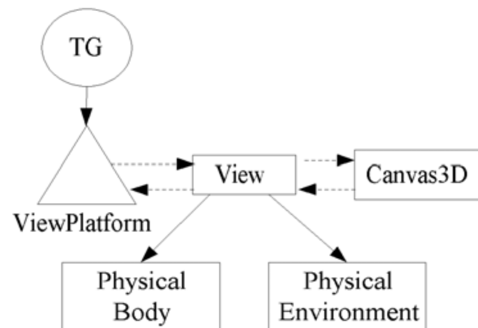

Figure 12. The Minimal View Branch Graph.

Figure 12 should be compared with the usual view graph shown in Figure 11.

Since the Figure 11 graph is constructed by the SimpleUniverse utility, it is often manipulated with support classes such as Viewer, ViewingPlatform, and PlatformGeometry. However, when building a new camera, it is easier to directly construct the subgraph.

The Figure 11 subgraph also contains PhysicalBody and PhysicalEnvironment nodes attached to its View, like those in Figure 12, but were not included in order to simplify the diagram.

Attributes related to the user's head are defined in PhysicalBody, such as the position of his/her eyes relative to the scene. PhysicalEnvironment specifies the environment in which the view will be generated, such as whether head-tracking, wall displays, or unusual audio output devices are being utilised. We are using a standard PC configuration, and so these objects can be created with their default settings.

The TransformGroup above the ViewPlatform in Figure 12 positions the viewpoint, *and* is also the connection point between the camera and the top-level BranchGroup in the main scene.

 © Andrew Davison. 2003

The SecondViewPanel class creates a subgraph like the one in Figure 12, and makes its TransformGroup, called canvas2TG, visible via the method getCamera2TG(). The class' other role is to embed its Canvas3D object inside a JPanel so that it can be easily inserted into a Swing-based GUI.

```
public class SecondViewPanel extends JPanel
{ private static final int PWIDTH = 256;   // size of panel
  private static final int PHEIGHT = 256;

  private MazeManager mazeMan;
  private TransformGroup camera2TG;
     // TG for the camera; will be linked to the main scene

  public SecondViewPanel(MazeManager mm)
  { mazeMan = mm;
    setLayout(new BorderLayout());
    setOpaque(false);
    setPreferredSize( new Dimension(PWIDTH, PHEIGHT));
    GraphicsConfiguration config =
            SimpleUniverse.getPreferredConfiguration();
    Canvas3D canvas3D = new Canvas3D(config);
    add("Center", canvas3D);    // JPanel contains its own Canvas3D

    initView(canvas3D);
  }

      :
} // end of SecondViewPanel class
```

The JPanel is 256 by 256 pixels large and hold a single Canvas3D object.

initView() constructs the subgraph:

```
private void initView(Canvas3D canvas3D)
{
  ViewPlatform vp = new ViewPlatform();

  // create a View node for the ViewPlatform
  // it has the same attributes as the main camera View
  View view = new View();
  view.setPhysicalBody( new PhysicalBody() );
  view.setPhysicalEnvironment( new PhysicalEnvironment() );
  view.addCanvas3D(canvas3D);
  view.attachViewPlatform(vp);   // attach the ViewPlatform

  view.setFieldOfView( Math.toRadians(90.0));
  view.setBackClipDistance(20);
  view.setFrontClipDistance(0.05);

  camera2TG = setCameraPosition();
  camera2TG.addChild(vp);   // add ViewPlatform to camera TG
}
```

The View attributes (FOV, clip distances) are set to be the same as the main camera.

No spotlight is added to the subgraph, causing the view behind the user to be substantially darker than the view ahead.

 © Andrew Davison. 2003

setCameraPosition() sets up the camera position by creating the top-level TransformGroup, and positioning it with lookAt(). This approach should be compared with the initViewPosition() method for the main camera, which uses rotation and translation.

```
private TransformGroup setCameraPosition()
{
  Vector3d startVec = mazeMan.getMazeStartPosn();

  Transform3D t3d = new Transform3D( );
  // args are: viewer posn, where looking, up direction
  t3d.lookAt( new Point3d(startVec.x, startVec.y, startVec.z),
          new Point3d(startVec.x, startVec.y, -10),
               // any -z value will do
          new Vector3d(0,1,0));
  t3d.invert();

  TransformGroup tg = new TransformGroup(t3d);
  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_READ); //moveable
  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
  return tg;
}
```

lookAt() specifies that the camera should be positioned at the starting position in the maze, and point along the negative z-axis. An implicit assumption is that the start point is somewhere in the positive quadrant of the XZ plane so that startVec.z cannot be less than 0.

Since the camera's default viewpoint faces into the scene, the transform created with lookAt() must be inverted to have the necessary effect.

This TransformGroup will be modified by the KeyBehavior object to move along with the main camera, so the relevant capability bits are set.

The creation of the SecondViewPanel and its addition to the GUI is carried out in Maze3D. Maze3D also passes a reference to the camera's top-level TransformGroup into WrapMaze3D:

```
     :
SecondViewPanel secondVP = new SecondViewPanel(mm);
WrapMaze3D w3d = new WrapMaze3D(mm, be,
                         secondVP.getCamera2TG() );
     :
Container c = getContentPane();
     :
Box vertBox = Box.createVerticalBox();
vertBox.add( secondVP );   // add back-facing camera pane
     :
c.add(vertBox);
     :
```

The TransformGroup, called camera2TG in WrapMaze3D, is added to the scene by the createSceneGraph() method:

```
sceneBG.addChild( camera2TG );
```


© Andrew Davison. 2003

9. The KeyBehavior Class

This chapter's KeyBehavior class is very similar to the one described in chapter 15, which is unsurprising since both control the user's viewpoint. They both extend ViewPlatformBehavior, so can access the ViewPlatform's TransformGroup, called targetTG. They both use the same kind of key presses to trigger movement and rotation.

Movement can be forwards, backwards, left, right, up, or down, by a single unit step, excluding moves through walls or the floor. Rotations are by 90 degrees left or right. This design was chosen to simplify the implementation of the BirdsEye class (described below) which must calculate the new position and orientation of its arrow when the viewpoint moves or rotates.

KeyBehavior is complicated by being involved in the manipulation of two cameras and the BirdsEye view. The movement/rotation of the cameras are done inside KeyBehavior by affecting their TransformGroups, but the BirdsEye object carries out its own changes. These additional responsibilities can be seen in standardMove() which calls the moveBy() and doRotateY() methods with extra arguments related to the extra views:

```
private void standardMove(int keycode)
{
  if(keycode == forwardKey)
    moveBy(VFWD, FORWARD, VBACK);
  else if(keycode == backKey)
    moveBy(VBACK, BACK, VFWD);
  else if(keycode == leftKey)
    doRotateY(ROT_AMT, LEFT);
  else if(keycode == rightKey)
    doRotateY(-ROT_AMT, RIGHT);
}
```

The arguments of moveBy() are constants related to the main camera, the bird's eye view, and the back facing camera. When the main camera moves in a given direction (e.g. VFWD, meaning forwards), then the back facing camera must move in the opposite direction (VBACK, meaning backwards). However, both cameras always rotate in the same direction, so no distinction is made between them when doRotateY () is called.

moveBy() implements collision detection by first calculating the new position after carrying out the requested move. If that position is occupied by an obstacle then the move is not executed, and a warning is reported. This 'try-it-and-see' approach is also employed in the Tour3D application of chapter 10.

```
private void moveBy(Vector3d theMove, int dir, Vector3d theMoveC2)
{
  Point3d nextLoc = possibleMove(theMove);
  if (mm.canMoveTo(nextLoc.x, nextLoc.z)) {   // no obstacle there?
    targetTG.setTransform(t3d);  // nasty!
    doMoveC2(theMoveC2);
    be.setMove(dir);
  }
  else  // there is an obstacle
    be.bangAlert();  // tell BirdsEye, so a warning can be shown
```

 © Andrew Davison. 2003

```
}
```

possibleMove() retrieves the current transform (into the global t3d) and makes the move, but doesn't update the TransformGroup.

```
private Point3d possibleMove(Vector3d theMove)
{
  targetTG.getTransform(t3d);   // targetTG is ViewPlatform's TG
  toMove.setTranslation(theMove);
  t3d.mul(toMove);
  t3d.get(trans);
  return new Point3d( trans.x, trans.y, trans.z);
}
```

The new location is returned to moveBy() which checks it by calling canMoveTo() in MazeManager. If all is well then targetTG is updated with t3d (see the line commented with 'nasty!'; nasty because the change is achieved with a global that was set in a different method).

moveBy() also has to change the back facing camera and the bird's eye view. doMoveC2() deals with the camera by applying a move to its TransformGroup.

```
private void doMoveC2(Vector3d theMoveC2)
{
  camera2TG.getTransform(t3d);
  toMove.setTranslation(theMoveC2);
  t3d.mul(toMove);
  camera2TG.setTransform(t3d);
}
```

Rotations to the cameras and bird's eye view can be carried out immediately without any collision testing, and are all done inside doRotateY().

```
private void doRotateY(double radians, int dir)
{
  targetTG.getTransform(t3d);   // rotate main camera
  toRot.rotY(radians);
  t3d.mul(toRot);
  targetTG.setTransform(t3d);

  camera2TG.getTransform(t3d);  // rotate back-facing camera
  t3d.mul(toRot);  // reuse toRot value
  camera2TG.setTransform(t3d);

  be.setRotation(dir);  // rotate bird's eye view
}
```

 © Andrew Davison. 2003

10. The Bird's Eye View

The BirdsEye object displays a static image representing the maze as seen from above, and draws an arrow on top of it to show the user's current position. As the user moves and turns, so does the arrow. If the user hits a wall, then the message "BANG!" appears (see Figure 13).

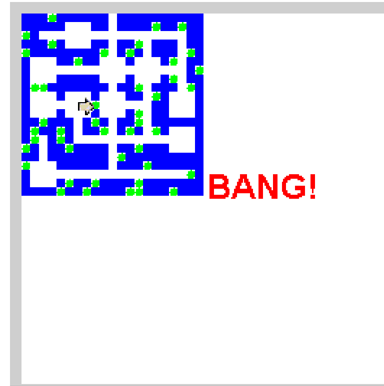

Figure 13. The Bird's Eye View Pane.

The arrow can switch between different images, shown in Figure 14 arranged around a clock.

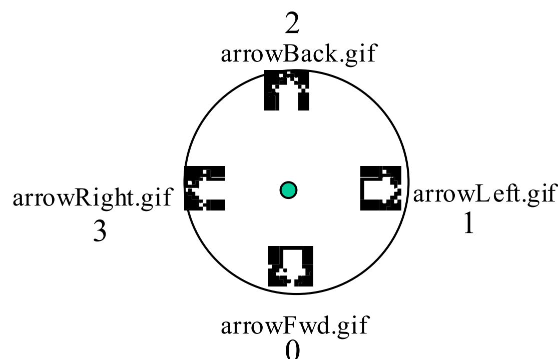

Figure 14. Arrow Images as a Clock.

These images are loaded into the arrowIms[] array at start-time, indexed with the numbers shown in the clock diagram. For instance, arrowIms[0] contains the image from arrowFwd.gif.

```
private static final int NUM_DIRS = 4;

private static final int FORWARD = 0;
private static final int LEFT = 1;
private static final int BACK = 2;
private static final int RIGHT = 3;

Image[] arrowIms = new Image[NUM_DIRS];

arrowIms[FORWARD] =new ImageIcon("images/arrowFwd.gif").getImage();
arrowIms[LEFT] = new ImageIcon("images/arrowLeft.gif").getImage();
arrowIms[BACK] = new ImageIcon("images/arrowBack.gif").getImage();
```


© Andrew Davison. 2003

```
arrowIms[RIGHT]=new ImageIcon("images/arrowRight.gif").getImage();
```

The images are ordered in the array so that moves and rotations can be calculated using clock arithmetic, as explained below.

moves[] is the other important data structure -- it contains the distance offset when the user moves forwards, left, backwards, or right:

```
private Point moves[];
 :
private void initMoves()
{ moves = new Point[NUM_DIRS];
  step = mm.getImageStep();
  moves[FORWARD] = new Point(0, step);  // move downwards on-screen
  moves[LEFT] = new Point(step, 0);     // right on-screen
  moves[BACK] = new Point(0, -step);    // up on-screen
  moves[RIGHT] = new Point(-step, 0);   // left on-screen
}
```

moves[] stores the offsets in the same order as arrowIms[].

The user's arrow starts at the maze's starting point. The starting direction is along the positive z-axis, which is down the screen when viewed from above in BirdsEye. This information is stored in initPosition():

```
private void initPosition()
{ currPosn = mm.getImageStartPosn();
  compass = FORWARD;
  userIm = arrowIms[FORWARD];
      // the user's arrow starts by facing down the screen
  showBang = false;
}
```

currPosn is a Point object holding the current (x,y) position of the arrow. compass is the current heading for the user *and* the index into the arrowIms[] to get the current arrow image for the user.

10.1. Moving the Arrow

KeyBehavior calls setMove() in BirdsEye in order to move the arrow, and supplies a direction which matches the FORWARD, LEFT, BACK or RIGHT constants. Before the move is made, the actual heading is calculated as the current compass value plus the direction, modulo 4.

```
public void setMove(int dir)
{
  int actualHd = (compass + dir) % NUM_DIRS;
  Point move = moves[actualHd];
  currPosn.x += move.x;  // update user position
  currPosn.y += move.y;
  repaint();
}
```

 © Andrew Davison. 2003

For example, if the compass value is LEFT (1) and the direction is BACK (2), then the actual heading will be RIGHT (1+2=3). This means that when the arrow is pointing left, and the user moves backwards, then the actual heading is to move right.

After the current position (currPosn) is updated, a repaint is requested which causes paintComponent() to be called. It draws the maze image first, then the arrow at the current position, and final the "BANG!" graphic if necessary.

10.2. Rotating the Arrow

KeyBehavior calls setRotation() in BirdsEye in order to rotate the arrow, and supplies a LEFT or RIGHT value. As with moves, the actual heading must be calculated, and this becomes the new compass value and changes the user's arrow.

```
public void setRotation(int dir)
{
  compass = (compass + dir) % NUM_DIRS;
  userIm = arrowIms[compass];  // update user's arrow
  repaint();
}
```

For example, if the compass value is LEFT (1) and the rotation direction is RIGHT (3), then the compass will change to FORWARD (1+3=0). This means that when the arrow is pointing left, and the user turns right, then the new heading is forward.

10.3. Why not use a Bird's Eye *Camera*?

The code in BirdsEye is quite tricky, but would be worse if the user could rotate through a wider choice of angles. The additional complexity would arise from the need to calculate more distance offsets, which would translate into a larger moves[] array and more images in arrowIms[].

An obvious alternative to BirdsEye is to create a third camera in a similar way to SecondViewPanel, but looking down on the maze from above. In fact, this was the original approach used in Maze3D but it proved unsatisfactory. The main reason was that it proved almost impossible to see the user's avatar (the cone) due to the darkness of the scene, the texturing on the floor, and the camera's height above the scene that made the cone very small. An obvious solution is to move the camera nearer but this loses most of the maze overview.

In general, it is very useful to present an *abstracted* picture of the entire scene which leaves out unnecessary detail such as textures and parts of the scenery. This helps the player maintain a general idea of the game without overloading him/her with information. Invariably, this abstraction requires a different kind of modeling than just another Java 3D pane.

Having both detailed views and overviews in a game suggests that key scene information (e.g. the maze plan) should be managed by an object which can supply alternative representations for these views. This is the role played by MazeManager in Maze3D.

 © Andrew Davison. 2003

11. Related Approaches to Scene Generation

The idea of using an ASCII plan to generate a 3D maze comes from the excellent "You Build It Virtual Reality" application by Chris Heistad and Steve Pietrowicz, developed while they were in the NCSA Java 3D Group.

The scene is defined using a series of ASCII 'maps' which specify the location of objects at different heights in the world. This layering approach allows the positioning of things along the y-axis. The notation used in the maps is extensible, so new kinds of objects can be included in the scene, including sounds and animations.

The world is networked (using multicasting), with each user represented by an avatar. Communication is via a chat window.

Unfortunately, this application is currently unavailable.

Daniel Selman has a DOOM-style application in his "Java 3D Programming" book, in section 11.6.3. The map is a GIF image, and the colours of its pixels are compared with colour values assigned to scenery objects to decide how the scene should be constructed.

Interesting elements include animated flaming torches, and guards that move through walls.

 © Andrew Davison. 2003