Chapter N2. Get a Life (in 3D)

This chapter introduces a number of programming techniques that I'll reuse frequently in subsequent chapters, including the integration of Java 3D's Canvas3D class (where a scene is rendered) with a Swing-based GUI, and displaying the scene in a full-screen window.

The scene is lit with ambient and directional lights, starts with a blue-sky background (but that can be changed), and is filled with multi-colored spheres (made with the Java 3D Sphere class). The spheres gradually rotate, change color, and fade in and out of view. These dynamic elements are driven by a simple subclass of Java 3D's Behavior acting as a timer, which triggers updates to the scene every 50 milliseconds.

The user's viewpoint (the camera) can be zoomed in and out, panned, and rotated with the mouse and control keys (courtesy of Java 3D's OrbitBehavior class).

1. The Game of Life

The application is a 3D version of Conway's Game of Life, a well-known cellular automaton. The original game consists of an infinite 2D grid of cells, each of which is either alive or dead. At every time 'tick', a cell evaluates rules involving the current state of its immediate neighbors to decide whether to continue living, to die, to stay dead, or be born. The rules in Conway's game are:

- Any living cell with fewer than two neighbors, or more than three, dies.
- Any living cell with two or three neighbors continues living.
- Any dead cell with exactly three neighbors comes to life.

All the cells in the grid are updated simultaneously in each time 'tick', so the entire grid moves to a new state (or *generation*) at once.

The infinite grid is usually implemented as a finite 2D array, with cells at an edge using the cells at the opposite edge as neighbors.

An important factor is the initial configuration of the grid (i.e. which cells start alive). Another variable is the shape of the grid: 1D and 3D (my interest) are possible, as are special shapes.

Although Conway's Game is only a game, cellular automata have found application in more serious areas, such as computability theory and theoretical biology. I include links to more information at the end of the chapter.

2. Running Life3D

My 3D version of the game, called Life3D, is shown in action in Figures 1 and 2, the screenshots were taken several generations apart.

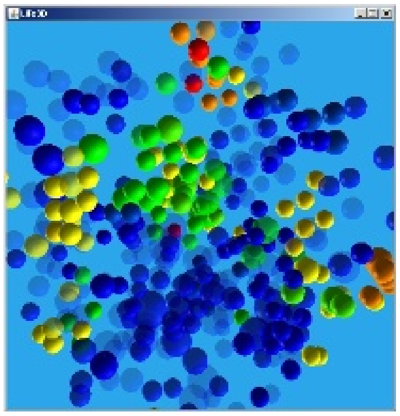


Figure 1. Life3D in Action.

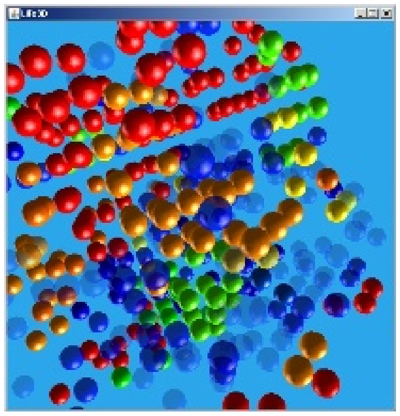


Figure 2. Life3D Still in Action.

Each cell is represented by a sphere, and the grid is a 10x10x10 lattice. When a cell is first born (or reborn) it's painted blue, then gradually changes color as it ages, through green, yellow, orange, and finally to red. When a cell die, it fades away, and when one is born, it gradually appears.

The grid is rotating randomly during all of this, changing its direction at regular intervals.

2.1. Configuring Life3D

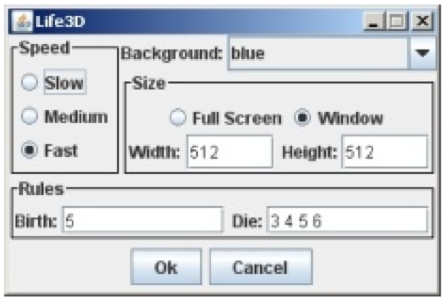
Life3D is configured using a properties file, life3DProps.txt. A properties file is a series of key=value pairs which can be read and updated easily with Java's Properties class. life3DProps.txt stores information on seven attributes: whether the application is full-screen or not, the window's width and height (if it's not full-screen), the rotation speed of the grid, the scene's background color, and the game's birth and die ranges. For example, my current version of life3DProps.txt contains:

```
fullscreen=false
width=512
height=512
speed=fast
bgColour=blue
birth=5
die=3 4 5 6
```

I'll explain how the birth and die ranges are used to initialize the game rules when I get to the CellsGrid class.

Since the properties file is text, it can be edited directly. However, Life3D includes a configuration screen (see Figure 3), which is displayed when the application is started with the -edit option:

```
$ java Life3D -edit
```



Speed: Slow / Medium / [✓ Fast]

Background: blue

Size: Full Screen / [✓ Window]

Width: 512 Height: 512

Rules

Birth: 5 Die: 3 4 5 6

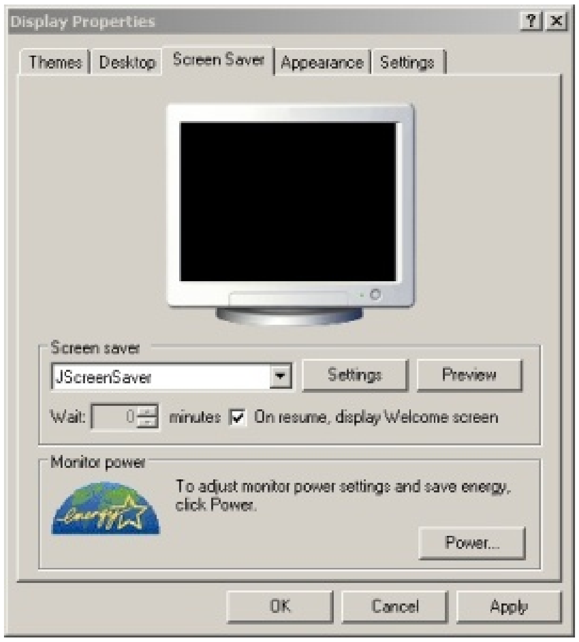
Figure 3. The Configuration Screen.

The adjusted properties are saved back to life3DProps.txt when the "Ok" button is pressed. Next time that Life3D is started, it will use the new settings.

2.2. A Life3D Screensaver

The main reason for integrating the configuration screen into the Life3D application is to make it fit the requirements of JScreenSaver, a Java-based Windows screensaver loader, written by Yoshinori Watanabe, and available at http://homepage2.nifty.com/igat/igapyon/soft/jssaver.html.

Figure 4 shows Window XP's "Display Properties" window with the screensaver tab ready to start Life3D.



Screen saver: JScreenSaver

Wait: 0 minutes [✓] On resume, display Welcome screen

Monitor power

To adjust monitor power settings and save energy, click Power.

Figure 4. The Life3D Screensaver.

To be honest, there's not much to see in Figure 4, since JScreenSaver doesn't support drawing to the preview mini-screen in the middle of the tab. However, JScreenSaver is shown as the currently selected saver.

Clicking on the "Settings" button brings up the configuration screen shown in Figure 3, while the "Preview" button starts Life3D proper (e.g. as in Figures 1 and 2).

I'll talk more about how to convert Life3D into a screensaver module at the end of this chapter, and discuss some other ways of making screensavers.

3. An Overview of the Life3D Classes

Figure 5 shows the class diagrams for the Life3D application. Only the public methods are shown, and I've left out superclasses and listeners.

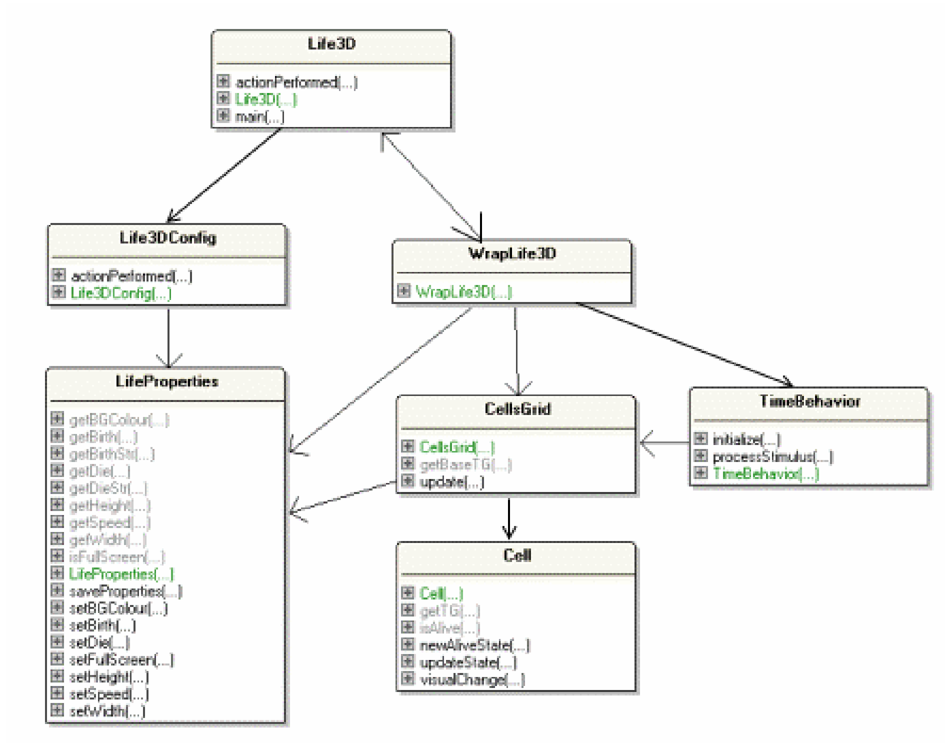


Figure 5. Class Diagrams for Life3D.

Life3D is a subclass of JFrame and manages the creation of the configuration screen (handled by Life3DConfig) or the 3D scene (in WrapLife3D) depending on if the command line include "-edit". Both Life3DConfig and Wrap3DLife are subclasses of JPanel.

Life3DConfig is a conventional mix of Swing controls for building the interface shown in Figure 3. It uses LifeProperties as an interface to the properties file, calling various get and set methods for the seven properties. I won't explain the Life3DConfig or LifeProperties classes in this chapter, since they're quite standard examples of using Swing and properties; both classes are fully documented.

WrapLife3D is the home for Java 3D's rendering of the scene, which is displayed in a Canvas3D object surrounded by WrapLife3D's JPanel. WrapLife3D handles many elements of the scene, including the background, lighting, and moving the camera. WrapLife3D delegates the creation of the cells grid to the CellsGrid class, which represents each cell (sphere) with a Cell object.

The TimeBehavior class (a subclass of Java 3D's Behavior) periodically calls the update() method in CellsGrid, triggering generational change in the grid.

In the rest of this chapter, I'll look at each of these classes in more detail.

4. Deciding How to Start

Life3D can progress in three ways:

1) if "-edit" is supplied on the command line then a configuration screen, managed by Life3DConfig, is slotted into Life3D's JFrame;
2) if the Java 3D API is found on the machine then the 3D application is started;
3) if Java 3D isn't found then Life3D reports the problem, and terminates.

The three-way branch is located in Life3D's constructor.

```
public Life3D(String[] args)
{
  super("Life3D");

  LifeProperties lifeProps = new LifeProperties();

  Container c = getContentPane();
  c.setLayout( new BorderLayout() );

  if (args.length > 0 && args[0].equals("-edit")) {
    // view/change application properties
    Life3DConfig l3Ctrls = new Life3DConfig(lifeProps);
    c.add(l3Ctrls, BorderLayout.CENTER);
  }
  else if (hasJ3D()) {
    // start the Life3D application
    WrapLife3D w3d = new WrapLife3D(this, lifeProps);
    c.add(w3d, BorderLayout.CENTER);
    if (lifeProps.isFullScreen())
      setUndecorated(true);   // no menubar, borders
  }
  else
    reportProb(c);

  setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  pack();
  setResizable(false);    // fixed size display

  // center this window
  Dimension screenDim = Toolkit.getDefaultToolkit().getScreenSize();
  Dimension winDim = this.getSize();
  this.setLocation( (screenDim.width-winDim.width)/2,
                    (screenDim.height-winDim.height)/2);
  setVisible(true);
} // end of Life3D()
```

The constructor does a few other things, the most notable being the creation of a LifeProperties object, which acts as the interface to the properties file. A reference to this object is passed to both JPanels (Life3DConfig and Wrap3DLife), and is also used inside Life3D.

If the properties specify full-screen rendering then the JFrame decorations (its menubar and borders) are switched off. The resizing of the window is done indirectly by the resizing of Wrap3DLife's JPanel, as you'll see when I get to that class.

If full-screen mode is employed, then the code for centering the window is superfluous, but it is utilized by the configuration screen, and when the 3D scene is drawn in a normal-sized pane, and when Life3D reports Java 3D's absence.

4.1. When Java 3D isn't Around

Since the Java 3D API isn't part of the standard Java 6 distribution, then it's a good idea to check for it, and try to 'die' gracefully when the API isn't found.

hasJ3D() checks for the presence of a key Java 3D class, SimpleUniverse, although any Java 3D class would do:

```
private boolean hasJ3D()
// check if Java 3D is available
{
  try {   // test for a Java 3D class
    Class.forName("com.sun.j3d.utils.universe.SimpleUniverse");
    return true;
  }
  catch(ClassNotFoundException e) {
    System.err.println("Java 3D not installed");
    return false;
  }
} // end of hasJ3D()
```

reportProb() reports on Java 3D using a JPanel filled with a label and a non-functioning button showing the Java 3D home URL (see Figure 6). I'll make the button start a browser, using Java 6's new Desktop API, in the next chapter.

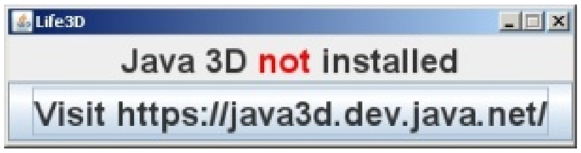


Figure 6. The Error Reporting Screen.

5. Displaying the 3D Game

If Java 3D is present, then WrapLife3D's constructor is called. It's main tasks are to integrate Java 3D's Canvas3D class into its JPanel, and to create the scene graph that's rendered inside the Canvas3D. Two lesser jobs are to fix the size of the JPanel (full-screen or a specific size), and to set up keyboard processing.

5.1. Integrating Java 3D and Swing

The Canvas3D view onto the 3D scene is created using:

```
// inside the WrapLife3D constructor
setLayout( new BorderLayout() );

GraphicsConfiguration config =
             SimpleUniverse.getPreferredConfiguration();
Canvas3D canvas3D = new Canvas3D(config);
add("Center", canvas3D);
```

Some care must be taken when using Canvas3D, since it's a heavyweight GUI element (a thin layer over an OS-generated window). Heavyweight components aren’t easily combined with Swing controls, which are lightweight (the controls are mostly generated by Java itself). Problems are avoided if the Canvas3D object is embedded in a JPanel (as here), then the panel can be safely integrated with the rest of the Swing-built application.

There's a detailed discussion of the issues related to combining Canvas3D and Swing at j3d.org (http://www.j3d.org/tutorials/quick_fix/swing.html).

One of the planned features of Java 3D 1.5, which should be available by the time you read this (after November 2006) is a JCanvas3D class, a lightweight version of Canvas3D. The JOGL rendering pipeline (another 1.5 innovation) will allow JCanvas3D to be accelerated, hopefully giving it better performance than Canvas3D.

The Canvas3D object is initialized with a configuration obtained from getPreferredConfiguration(); this method queries the hardware for rendering information. Some older Java 3D programs don't bother initializing a GraphicsConfiguration object, using null as the argument to the Canvas3D constructor instead; this isn't good programming practice.

5.2. Window Sizing

WrapLife3D checks the properties file (via the LifeProperties object) to see whether the application should be full-screen or not, and sets the size of the JPanel accordingly.

```
// global
private LifeProperties lifeProps;

// in the WrapLife3D constructor
if (lifeProps.isFullScreen())
  setPreferredSize( Toolkit.getDefaultToolkit().getScreenSize() );
else {  // not full-screen
  int width = lifeProps.getWidth();
  int height = lifeProps.getHeight();
  setPreferredSize( new Dimension(width, height));
}
```

The window's width and height are obtained from the properties file.

The changes to the JPanel's dimensions will affect the parent JFrame since it was tied to the size of the JPanel by its call to Window.pack().

5.3. Processing Key Presses

WrapLife3D gives focus to the Canvas3D object, canvas3D, so that keyboard events will be visible to behaviors in the scene graph. This is a good general strategy since most applications require user input to be directed to the code controlling the scene graph. However, the scene dynamics in Life3D are driven by a time-triggered behavior, so it's not really necessary to set the focus. I've left these lines in since they're needed in almost every other program we'll consider.

Keyboard input is utilized in Life3D, but only as a quick way to terminate the application. This feature becomes essential when Life3D is shown in full-screen mode since there isn't a menubar with a close box.

A key listener is attached to the canvas, and calls Window.dispose() in the top-level JFrame.

```
// global
private Life3D topLevel;  // the JFrame


// in the WrapLife3D constructor
canvas3D.setFocusable(true);
canvas3D.requestFocus();
   // the canvas now has focus, so receives key events

canvas3D.addKeyListener( new KeyAdapter() {
// listen for esc, q, end, ctrl-c on the canvas to
// allow a convenient exit from the full-screen configuration
  public void keyPressed(KeyEvent e)
  { int keyCode = e.getKeyCode();
    if ((keyCode == KeyEvent.VK_ESCAPE) ||
        (keyCode == KeyEvent.VK_Q) ||
        (keyCode == KeyEvent.VK_END) ||
        ((keyCode == KeyEvent.VK_C) && e.isControlDown()) ) {
       topLevel.dispose();
       System.exit(0);
       // exit() alone isn't sufficient most of the time
     }
  }
});
```

The call to dispose() ensures that all the screen resources are released, which doesn't seem to be the case when the key listener only calls System.exit().

6. Scene Graph Creation

The scene graph is created by the WrapLife3D constructor after instantiating the Canvas3D object:

```
// globals
private SimpleUniverse su;
private BranchGroup sceneBG;


// inside the WrapLife3D constructor
su = new SimpleUniverse(canvas3D);

createSceneGraph();
initUserPosition();        // set user's viewpoint
orbitControls(canvas3D);   // controls for moving the viewpoint

su.addBranchGraph( sceneBG );
```

The su SimpleUniverse object very kindly generates a standard view branch graph and the VirtualUniverse and Locale nodes of the scene graph for me, but I have to do the rest. createSceneGraph() sets up the lighting, the sky background, the floor, and the cells grid, while initUserPosition() and orbitControls() handle viewer issues. The three method calls in bold will be explained in more detail below.

createSceneGraph() builds the application scene graph below a sceneBG BranchGroup, which is connected to the graph made by SimpleUniverse by calling addBranchGroup().

```
// globals
private static final int BOUNDSIZE = 100;  // larger than world

private BranchGroup sceneBG;
private BoundingSphere bounds;   // for environment nodes


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

  lightScene();       // add the lights
  addBackground();    // add the sky
  addGrid();          // add cells grid

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

Various methods add subgraphs to sceneBG to build up the content branch graph. sceneBG is compiled once the graph has been finalized, to allow Java 3D to optimize it. The optimizations may involve reordering the graph, and regrouping and combining nodes. For example, a chain of TransformGroup nodes containing different translations may be combined into a single node.

bounds is a global BoundingSphere used to specify the influence of environment nodes for lighting, background, and the OrbitBehavior object. The bounding sphere is placed at the center of the scene, and affects everything within a BOUNDSIZE units radius. Bounding boxes and polytopes are also available.

The scene graph created by WrapLife3D is shown in Figure 7.

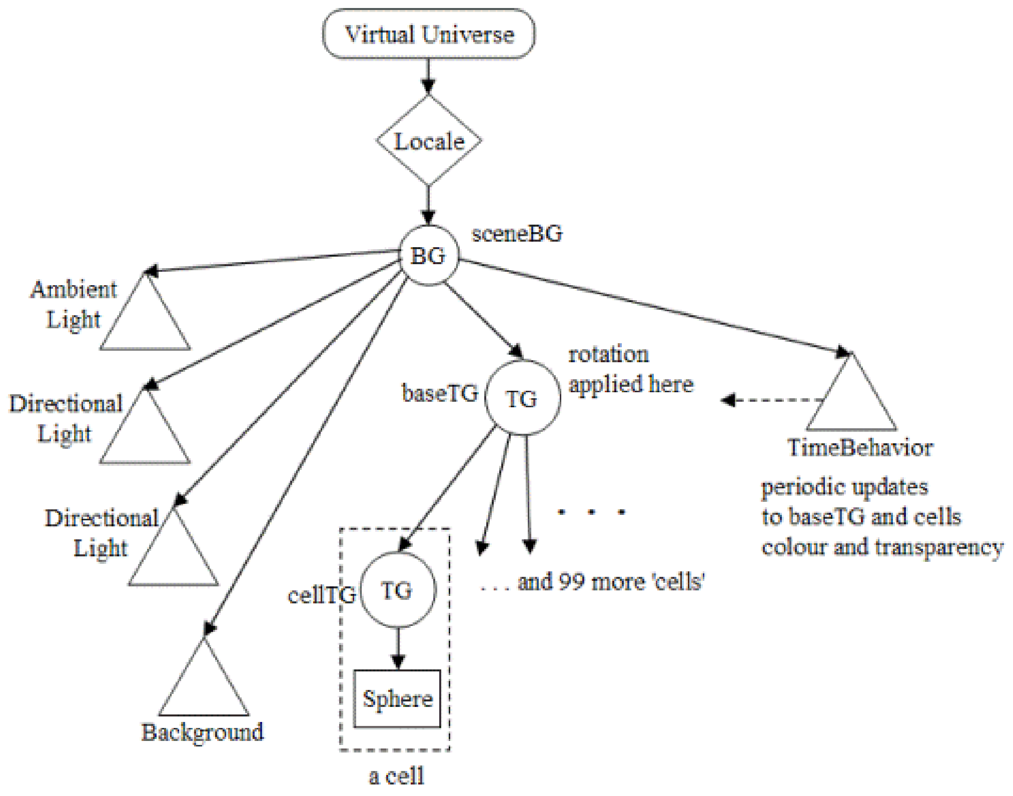


Figure 7. Scene graph for Life3D.

The three lights are created by lightScene(), the Background node by addBackGround(), and the subgraph starting at the baseTG TransformGroup by addGrid(). The TimeBehavior object is also instantiated in addGrid().

The view branch part of the scene graph is missing from Figure 7, since it's created by the SimpleUniverse object in WrapLife3D.

6.1. Lighting the Scene

One ambient and two directional lights are added to the scene by lightScene(). An ambient light reaches every corner of the world, illuminating everything equally.

```
private void lightScene()
{
  Color3f white = new Color3f(1.0f, 1.0f, 1.0f);

  // Set up the ambient light
  AmbientLight ambientLightNode = new AmbientLight(white);
  ambientLightNode.setInfluencingBounds(bounds);
  sceneBG.addChild(ambientLightNode);
  // Set up the directional lights
  Vector3f light1Direction  = new Vector3f(-1.0f, 1.0f, -1.0f);
        // light coming from left, up, and back quadrant
  Vector3f light2Direction  = new Vector3f(1.0f, 1.0f, 1.0f);
        // light coming from right, up, and front quadrant

  DirectionalLight light1 =
          new DirectionalLight(white, light1Direction);
  light1.setInfluencingBounds(bounds);
  sceneBG.addChild(light1);

  DirectionalLight light2 =
      new DirectionalLight(white, light2Direction);
  light2.setInfluencingBounds(bounds);
  sceneBG.addChild(light2);
}  // end of lightScene()
```

The color of the light is set, the ambient source is created along with bounds, and added to the scene. The Color3f() constructor takes Red/Green/Blue values between 0.0f and 1.0f (1.0f being ‘full on’).

A directional light mimics a light from a distant source, hitting the surfaces of objects from a specified direction. The main difference from an ambient light is the requirement for a direction, such as:

```
Vector3f light1Direction  = new Vector3f(-1.0f, 1.0f, -1.0f);
   // light coming from left, up, and back quadrant
```

The direction is the vector starting at the specified coordinate, pointing towards (0, 0, 0); the light can be imagined to be multiple parallel lines with that direction, originating at infinity.

Point and spot lights are the other forms of Java 3D lighting. Point lights position the light in space, emitting in all directions. Spot lights are focused point lights, aimed in a particular direction.

6.2. The Scene's Background

A background for a scene can be specified as a constant color (as here), a static image, or a texture-mapped geometry such as a sphere.

```
private void addBackground()
/* The choice of background color is obtained from the
   properties file (blue, green, white, or black). */
{
  Background back = new Background();
  back.setApplicationBounds( bounds );

  int bgColour = lifeProps.getBGColour();
  if (bgColour == LifeProperties.BLUE)
    back.setColor(0.17f, 0.65f, 0.92f);    // sky blue color
  else if (bgColour == LifeProperties.GREEN)
    back.setColor(0.5f, 1.0f, 0.5f);       // grass color
  else if (bgColour == LifeProperties.WHITE)
    back.setColor(1.0f, 1.0f, 0.8f);       // off-white
  // else black by default
  sceneBG.addChild( back );
}  // end of addBackground()
```

The code is complicated by the need to check with the LifeProperties object to determine the currently specified color, as represented by the constants LifeProperties.BLUE, LifeProperties.GREEN, LifeProperties.WHITE, and LifeProperties.BLACK. addBackground() maps these to RGB values.

6.3. Building the Cells Grid, and Making it Behave

The work required to create the scene graph for the grid is left to a CellsGrid object. addGrid() creates the object, and a TimeBehavior instance for triggering grid changes.

```
// time delay (in ms) to regulate update speed
private static final int TIME_DELAY = 50;


private void addGrid()
/*  Create the cells grid and a time behavior to update
    it at TIME_DELAY intervals. */
{
  CellsGrid cellsGrid = new CellsGrid(lifeProps);
  sceneBG.addChild( cellsGrid.getBaseTG() );

  TimeBehavior tb = new TimeBehavior(TIME_DELAY, cellsGrid);
  tb.setSchedulingBounds(bounds);
  sceneBG.addChild(tb);
}  // end of addGrid()
```

The LifeProperties object, lifeProps, is passed to the CellsGrid object since the properties include the rotation speed for the grid and the birth and die ranges, which CellsGrid handles.

6.4. Viewer Positioning

The scene graph in Figure 7 doesn't include the view branch graph; that branch is shown in Figure 8.

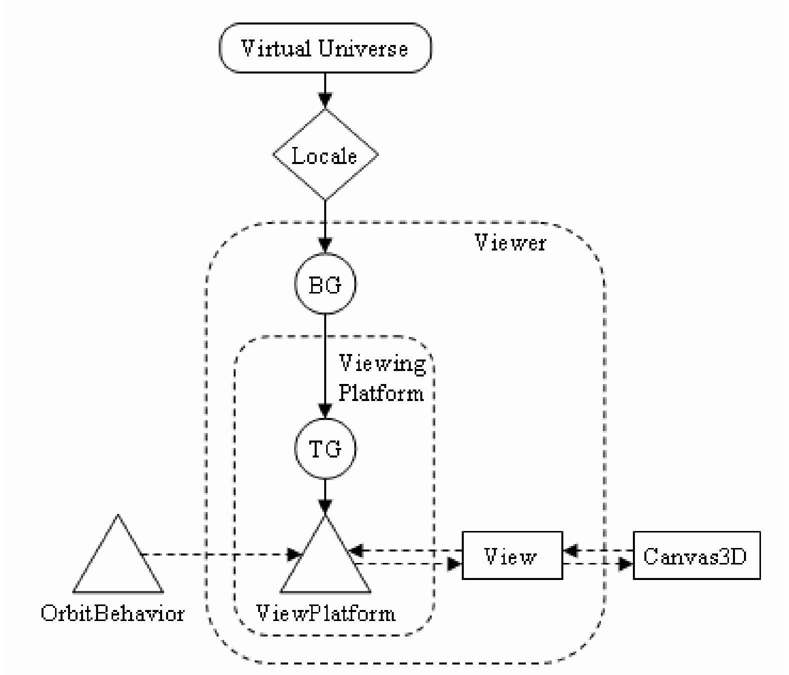


Figure 8. The View Branch Graph.

Most of the branch is generated by a call to the SimpleUniverse constructor in the WrapLife3D() constructor:

```
su = new SimpleUniverse(canvas3D);
```

SimpleUniverse offers simplified access to the view branch graph via the ViewingPlatform and Viewer classes, which are mapped to the graph (shown as dotted rectangles in Figure 8).

ViewingPlatform is used in initUserPosition() to access the TransformGroup above the ViewPlatform node:

```
// global
private static final Point3d USERPOSN = new Point3d(-2,5,10);
  // initial user position


private void initUserPosition()
/* Set the user's initial viewpoint using lookAt()  */
{
  ViewingPlatform vp = su.getViewingPlatform();
  TransformGroup steerTG = vp.getViewPlatformTransform();

  Transform3D t3d = new Transform3D( );
  steerTG.getTransform( t3d );
  t3d.lookAt( USERPOSN, new Point3d(0,0,0), new Vector3d(0,1,0));
  // args are: viewer posn, where looking, up direction
  t3d.invert();

  steerTG.setTransform(t3d);
}  // end of initUserPosition()
```

lookAt() is a convenient way to set the viewer's position (i.e. the camera position) in the virtual world. The method requires the viewer's intended position, the point that she is looking at, and a vector specifying the upward direction. In this application, the viewer's position is USERPOSN (the (-2, 5, 10) coordinate); she is looking towards the origin (0, 0, 0), and 'up' is along the positive y-axis. This is illustrated by Figure 9.

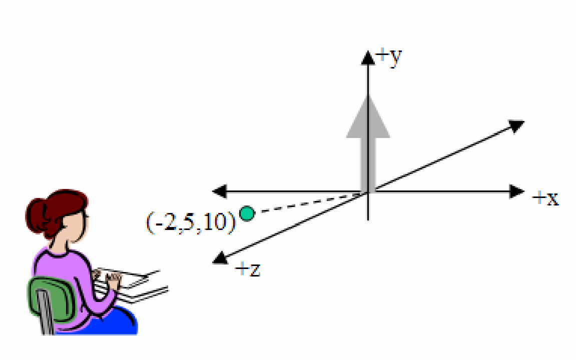


Figure 9. lookAt() Depicted Graphically.

invert() is required since the position is relative to the viewer rather than an object in the scene.

6.5. Viewer Movement

The user is able to move through the scene by connecting a Java 3D OrbitBehavior object to the view graph (the triangle in Figure 8). It offers a combination of control keys and mouse button presses to pan, zoom, and rotate the viewer's position.

The behavior is set up in orbitControls() in WrapCheckers3D:

```
private void orbitControls(Canvas3D c)
{
  OrbitBehavior orbit =
         new OrbitBehavior(c, OrbitBehavior.REVERSE_ALL);
  orbit.setSchedulingBounds(bounds);

  ViewingPlatform vp = su.getViewingPlatform();
  vp.setViewPlatformBehavior(orbit);
}  // end of orbitControls()
```

The REVERSE_ALL flag ensures that the viewpoint moves in the same direction as the mouse.

There are numerous other flags and methods for affecting the rotation, translation, and zooming characteristics, explained in the OrbitBehavior class documentation.

The Java 3D classes, MouseRotate, MouseTranslate, and MouseZoom, are similar behavior classes that appear in many examples; their principal difference from OrbitBehavior is that they affect the objects in the scene rather than the viewer.

Most games, such as first person shooters, require greater control over the viewer's movements than these utility behaviors can offer, so I'll be implementing my own behaviors in later chapters.

7. Behaviors in Java 3D

A Behavior object is used to monitor and respond to events occurring in a Java 3D application, such as key presses, the rendering of frames, the passage of time, the movement of the user's viewpoint, Transform3D changes, and collisions. These events, called *wakeup criteria*, activate the Behavior object so it can carry out specified tasks.

A typical Behavior subclass has the format:

```
public class FooBehavior extends Behavior
{
  private WakeupCondition wc;   // what will wake the object
  // other global variables

  public FooBehavior(…)
  { // initialise globals
    wc = new ...  //  create the wakeup condition
  }

  public void initialize()
  // register interest in the wakeup condition
  {  wakeupOn(wc);  }


  public void processStimulus(Enumeration criteria)
  {
     WakeupCriterion wakeup;
     while (criteria.hasMoreElements() ) {
       wakeup = (WakeupCriterion) criteria.nextElement();
       // determine the type of criterion assigned to wakeup;
       // carry out the relevant task;
     }
     wakeupOn(wc);  // re-register interest
   } // end of processStimulus()

}  // end of FooBehavior class
```

A subclass of Behavior must implement initialize() and processStimulus(). initialize() should register the behavior's wakeup condition, but other initialization code can be placed in the constructor for the class. processStimulus() is called by Java 3D when an event (or events) of interest to the behavior is received. Often, the simple matter of processStimulus() being called is enough to decide what task should be carried out (e.g. as in TimeBehavior below). In more complex classes, the events passed to the object must be analyzed. For example, a key press may be the wakeup criterion, but the code will also need to determine which key was pressed.

A common error when implementing processStimulus() is to forget to re-register the wakeup condition at the end of the method:

```
wakeupOn(wc);  // re-register interest
```

If this is skipped, the behavior won't be triggered again.

A WakeupCondition object can be a combination of one or more WakeupCriterion. There are many subclasses of WakeupCriterion, including:

- WakeupOnAWTEvent
  For AWT events such as key presses and mouse movements.
- WakeupOnElapsedFrames
  An event can be generated after a specified number of renderings. This criterion should be used with care since it may result in the object being triggered many times per second.
- WakeupOnElapsedTime
  An event can be generated after a specified time interval. WakeupOnElapsedTime is used in TimeBehavior below.

Another common mistake when using Behaviors is to forget to specify a scheduling volume (or region) with Behavior.setSchedulingBounds(). A Behavior node is only active (and able to receive events) when the user's viewpoint intersects a Behavior object's scheduling volume. If no volume is set, then the Behavior will never be triggered.

The volume for TimeBehavior is set in addGrids() when the behavior object is created.

7.1. A Time-based Behavior

The TimeBehavior class is pleasantly short since processStimulus() doesn't need to examine its wakeup criteria.

```
public class TimeBehavior extends Behavior
{
  private WakeupCondition timeOut;
  private int timeDelay;
  private CellsGrid cellsGrid;


  public TimeBehavior(int td, CellsGrid cg)
  { timeDelay = td;
    cellsGrid = cg;
    timeOut = new WakeupOnElapsedTime(timeDelay);
  }

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

  public void processStimulus(Enumeration criteria)
  { cellsGrid.update();   // ignore criteria
    wakeupOn(timeOut);
  }
}  // end of TimeBehavior class
```

The wakeup condition is an instance of WakeupOnElapsedTime, making TimeBehavior the Java 3D equivalent of a timer which fires every timeDelay milliseconds. It calls the update() method in CellsGrid to change the grid.

8. Managing the Grid

CellsGrid creates and controls a 10*10*10 grid of Cell objects centered at (0,0,0) in the scene.

TimeBehavior periodically calls CellsGrid's update() method to update the grid. Since I'm interested in using Life3D as a screensaver later on, an update can trigger either a state change *or* a visual change. Figure 10 shows the general idea: a state change is followed by a series of visual changes (MAX_TRANS+1 of them), and then the sequence repeats.

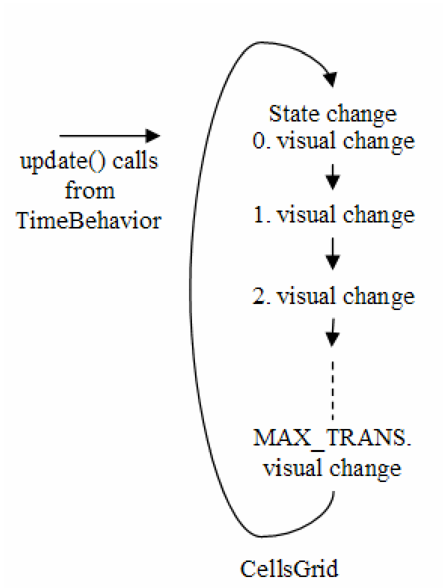


Figure 10. Updates Causing State and Visual Changes.

A state change changes the grid's cells state by applying game rules using birth and die ranges. I'll describe what these ranges are later in this section.

Most updates trigger visual changes to the cells, which affect their visibility or color. A visual transition is spread out over several updates, so a sphere gradually appears, disappears, or changes color based on its current age. The choice of transition depends on the current visual state of the cell, which I'll explain when I describe the Cell class.

Every update also causes the grid to rotate, and the rotation axis is periodically changed so the grid moves in a random manner.

8.1. Accessing Properties

The birth and die ranges, and the grid rotation speed, are specified by properties obtained a LifeProperties object. The code is located in the CellsGrid constructor:

```
// globals
private LifeProperties lifeProps;

// birth and die ranges used in the life rules
boolean[] birthRange, dieRange;

public CellsGrid(LifeProperties lps)
{
  lifeProps = lps;

  // load birth and die ranges
  birthRange = lifeProps.getBirth();
  dieRange = lifeProps.getDie();

  setTurnAngle();

  // scene graph creation code
}  // end of CellsGrid() constructor
```

setTurnAngle reads a speed constant from the properties file (it may be SLOW, MEDIUM or FAST), and converts it into a rotation angle.

```
// globals
// grid rotation amount
private static final double ROTATE_AMT = Math.toRadians(4);
                                          // 4 degrees
private double turnAngle;

private void setTurnAngle()
{
  int speed = lifeProps.getSpeed();

  if (speed == LifeProperties.SLOW)
    turnAngle = ROTATE_AMT/4;
  else if (speed == LifeProperties.MEDIUM)
    turnAngle = ROTATE_AMT/2;
  else  // fast --> large rotation
    turnAngle = ROTATE_AMT;
}  // end of setTurnAngle()
```

The grid's turning angle is larger for faster speeds.

8.2. Creating the Grid Scene Graph

The CellsGrid constructor is also responsible for building the scene graph branch for the cells. This corresponds to the branch below the baseTG TransformGroup node in Figure 7, which is repeated here again as Figure 11 for convenience.

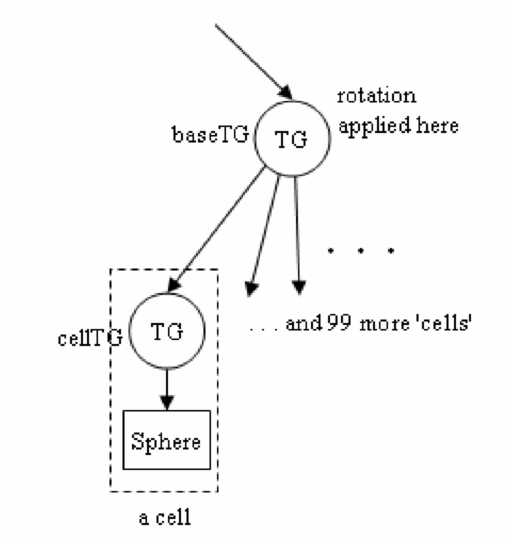


Figure 11. The Cells Grid Part of the Scene Graph.

The code for creating the cells is:

```
// globals
// number of cells along the x-, y-, and z- axes
private final static int GRID_LEN = 10;

private Cell[][][] cells;        // cells storage
private TransformGroup baseTG;   // used to rotate the grid


// in the CellsGrid constructor
/* Allow baseTG to be read and changed at run time (so
   it can be rotated). */
baseTG = new TransformGroup();
baseTG.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
baseTG.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);

// initialize the grid with Cell objects
cells = new Cell[GRID_LEN][GRID_LEN][GRID_LEN];
for (int i=0; i < GRID_LEN; i++)
  for (int j=0; j < GRID_LEN; j++)
    for (int k=0; k < GRID_LEN; k++) {
      cells[i][j][k] =
           new Cell(i-GRID_LEN/2, j-GRID_LEN/2, k-GRID_LEN/2);
           // subtract GRID_LEN/2 so grid is centered
      baseTG.addChild( cells[i][j][k].getTG() );  //connect to baseTG
    }
```

The runtime manipulation of scene graph nodes is only possible if those nodes have the desired capabilities switched on with setCapability() calls. The orientation of the baseTG TransformGroup will be read and changed at runtime, necessitating two setCapability() calls.

Each cell is managed by a Cell object which is passed a coordinate derived from its position in the cells[][][] array. The subtraction of GRID_LEN/2 from i, j, and k means that the 10x10x10 grid is centered on the origin.

To understand this, it helps to recall how the viewer is orientated with respect to the axes (see Figure 12).

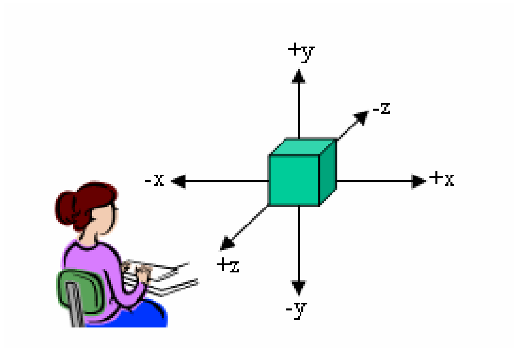


Figure 12. Axes in Java 3D.

If a cell is positioned using only the i, j, and k values then it be located too close to the viewer, too far to the right and too far up. The subtraction of GRID_LEN/2 moves the cell back towards a central region around the origin.

8.3. Updating the Cells States

As shown in Figure 10, an update() call either triggers a state change or a visual change. The cycling through the changes is done using a counter, transCounter, which is incremented from 0 to MAX_TRANS, then repeats. When transCounter is 0, the state of the grid's cells is updated, and for other values, the cells' visuals are changed. The grid is rotated at every update, irrespective of the transCounter value.

```
// globals
// number of updates used to complete a visual transition
public static final int MAX_TRANS = 8;

// transition (transparency/color change) step counter
private int transCounter = 0;
private Random rand = new Random();


public void update()
{
  if (transCounter == 0) {   // time for grid state change
    stateChange();
    turnAxis = rand.nextInt(3);  // change rotation axis
    transCounter = 1;
  }
  else {   // make a visual change
    for (int i=0; i < GRID_LEN; i++)
      for (int j=0; j < GRID_LEN; j++)
        for (int k=0; k < GRID_LEN; k++)
```

```
          cells[i][j][k].visualChange(transCounter);

    transCounter++;
    if (transCounter > MAX_TRANS)
      transCounter = 0;   // reset counter
  }

  doRotate();   // rotate in every update() call
}  // end of update()
```

A visual change is handled by iterating over the cells[][][] array, and calling Cell.visualChange() for each cell. The current value of transCounter is supplied, and Cell has access to the MAX_TRANS constant as well, since it's declared public in CellsGrid.

The state change performed by stateChange() is a two-stage affair – first it determines the state of each cell in the next generation. Only when every cell has been examined, are they then updated. The state update includes the first visual change to a cell (see Figure 10).

```
private void stateChange()
{
  boolean willLive;

  // calculate next state for each cell
  for (int i=0; i < GRID_LEN; i++)
    for (int j=0; j < GRID_LEN; j++)
      for (int k=0; k < GRID_LEN; k++) {
        willLive = aliveNextState(i, j, k);
        cells[i][j][k].newAliveState(willLive);
      }

  // update each cell
  for (int i=0; i < GRID_LEN; i++)
    for (int j=0; j < GRID_LEN; j++)
      for (int k=0; k < GRID_LEN; k++) {
        cells[i][j][k].updateState();
        cells[i][j][k].visualChange(0);
      }
}  // end of stateChange()
```

The next state for a cell is determined by calling aliveNextState(), and the value is stored in the cell by calling Cell.newAliveState(). However, the cell doesn't actually use the state until Cell.updateState() is called inside the second for-loop block.

8.4. Will the Cell Live or Die?

There are many ways of specifying how a cell's state may change. The state diagram in Figure 13 shows the four possible transitions between the 'alive' and 'dead' states.

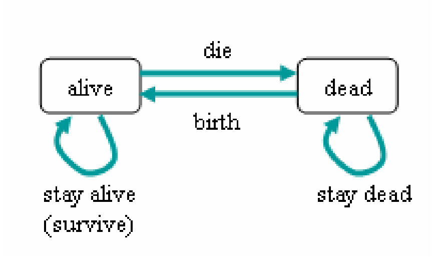


Figure 13. Cell State Changes.

CellsGrid utilizes rules which specify reasons to 'die' (i.e. move from the 'alive' state to 'dead'), and reasons for 'birth' (i.e. from 'dead' to 'alive'). If these rules can't be applied to a particular cell, then it stays in its current state.

The coding uses birth and die ranges, which specify the *number* of cell neighbors that can cause a 'birth' or 'die' transition.

For example, the birth and die ranges might be:

birth range = {5, 10}   die range = {3, 4, 5, 6}

These ranges state that a 'dead' cell is born if it has 5 or 10 living neighbors, and an 'alive' cell dies if it has 3, 4, 5, or 6 living neighbors.

On the Life3D configuration screen (see Figure 3), these ranges can be changed by typing in new numbers separated by spaces. Figure 3 shows that the birth range has been changed to only use 5 neighbors.

The birth and die ranges are encoded as boolean arrays in the CellsGrid code. For the birthRange[] array, if birthRange[i] is true then i living neighbors are needed to bring a cell to life. Each array has 27 elements, large enough for the 26 neighbors of a 3D cell, with the 0th entry employed when having *no* living neighbors will trigger the transition.

For the example above, birthRange[5] and birthRange[10] will be true, and its other elements false. The dieRange[] array will contain true at index positions 3, 4, 5, and 6.

aliveNextState() uses the birth and die ranges by first collecting two pieces of information for the cell at position (i,j,k) in the cells[][][] array: the number of its living neighbors, and its current state (i.e. alive or dead).

```
private boolean aliveNextState(int i, int j, int k)
{
  // count all the living neighbors, but not the cell itself
  int numberLiving = 0;
  for(int r=i-1; r <= i+1; r++)  // range i-1 to i+1
    for(int s=j-1; s <= j+1; s++)  // range j-1 to j+1
      for(int t=k-1; t <= k+1; t++) {  // range k-1 to k+1
        if ((r==i) && (s==j) && (t==k))
          continue;   // skip self
        else if (isAlive(r,s,t))
```

```
          numberLiving++;
      }

  // get the cell's current life state
  boolean currAliveState = isAlive(i,j,k);

  // ** Life Rules **: calculate the cell's next life state

  if (birthRange[numberLiving] && !currAliveState)
    return true;   // to be born && dead now --> make alive
  else if (dieRange[numberLiving]  && currAliveState)
    return false;  // to die && alive now --> kill off
  else
    return currAliveState;  // no change
}  // end of aliveNextState()
```

aliveNextState() returns true or false representing 'alive' or 'dead', and this value becomes the cell's new state when it is updated.

isAlive() gets the cell's current state by calling Cell.isAlive(), and also deals with grid edge cases, when the neighbor is on the opposite side of the grid.

```
private boolean isAlive(int i, int j, int k)
{
  // deal with edge cases for cells array
  i = rangeCorrect(i);
  j = rangeCorrect(j);
  k = rangeCorrect(k);
  return  cells[i][j][k].isAlive();
}  // end of isAlive()


private int rangeCorrect(int index)
/* if the cell index is out of range then use the index of
   the opposite edge */
{
  if (index < 0)
    return (GRID_LEN + index);
  else if (index > GRID_LEN-1)
    return (index - GRID_LEN);
  else // make no change
    return index;
}  // end of rangeCorrect()
```

8.5. Rotating the Grid

The baseTG TransformGroup stores its position, rotation, and scaling information in a 4x4 matrix. Thankfully, we can manipulate it using Java 3D's Transform3D class, which offers numerous methods for translating, rotating, and scaling.

The programming strategy, is to copy the matrix from a TransformGroup node as a Transform3D object, apply an operation to it (e.g. a rotation), then write the changed Transform3D back into the TransformGroup. When the scene is next rendered, the node will be changed accordingly.

baseTG highlights an important advantage of the scene graph hierarchy: since all the cells are children nodes of baseTG (see Figure 11), they'll be affected by the transformation applied to baseTG. This means that only baseTG needs to be rotated in order to turn the entire grid.

doRotate() is called at the end of the update() method:

```
// globals
// reusable Transform3D object
private Transform3D t3d = new Transform3D();
private Transform3D rotT3d = new Transform3D();

private int turnAxis = 0;

private void doRotate()
// rotate the object turnAngle radians around an axis
{
  baseTG.getTransform(t3d);  // get current rotation
  rotT3d.setIdentity();      // reset the rotation transform object

  switch (turnAxis) {    // set rotation based on the current axis
    case 0: rotT3d.rotX(turnAngle); break;
    case 1: rotT3d.rotY(turnAngle); break;
    case 2: rotT3d.rotZ(turnAngle); break;
    default: System.out.println("Unknown axis of rotation"); break;
  }

  t3d.mul(rotT3d);            // 'add' new rotation to current one
  baseTG.setTransform(t3d);   // update the TG
}  // end of doRotate()
```

The transformation matrix is copied into t3d using TransformGroup.getTransform(), and written back with TransformGroup.setTransform() at the end of the method. t3d is utilized each time that doRotate() is called, rather than repeatedly creating a temporary object, to reduce garbage collection, and so improve the application's efficiency.

doRotate() applies a rotation around either the x-, y-, or z- axes using the turnAngle value (which was set in getTurnAngle()). The rotation direction can be determined using *the right hand rule*: point the thumb of your right hand along the positive axis being used for the rotation, and the turning direction will be the direction of your closed fingers (as in Figure 14).

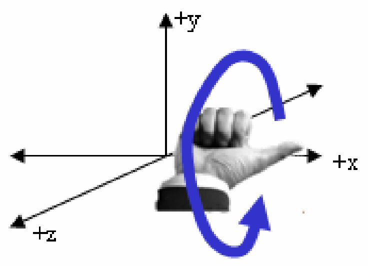


Figure 14. The Right Hand Rule for Rotation.

Figure 14 illustrates the direction of a positive rotation around the x-axis.

The choice of axis depends on the turnAxis value, which is randomly changed whenever there's a state change (the code is in update()). The rotation is stored in rotT3d, another reusable Transform3D object, using Transform3D.rotX(), rotY(), or rotZ() to store a x-, y-, or z- axis rotation.

If you run Life3D, it seems that the rotations are much more varied than just turns around the three axes. The apparent variety is because a rotation around one axis affects the position of the other axes. This makes subsequent rotations using those axes appear different.

For instance, when the rotation in Figure 14 is carried out on baseTG, it turns its y- and z- axes as well: the +y-axis rotates forwards, and the +z axis downwards. If a subsequent rotation is applied around those axes, it will be relative to their new orientations.

The rotT3d rotation is applied to the existing transformation in t3d by multiplying the objects together:

```
t3d.mul(rotT3d);
```

This is t3d = t3d * rotT3d, which multiplies the object's matrices together. This has the effect of *adding* the rotT3d rotation to the rotation in t3d.

baseTG's initial orientation is specified when it's created in the CellsGrid constructor:

```
baseTG = new TransformGroup();
```

The Transform3D component is set to be the identity matrix, which means that the node is positioned at the origin, and is unrotated and unscaled.

9. The Cell

A good way to understand how a cell works is to consider its possible states. In terms of the Life game, a cell can be either alive or dead, as shown in Figure 13. However, the picture becomes more complex when I add the possible visual changes:

- gradually fading out when the cell dies;
- gradually fading in as the cell is born;
- color changes as the cell gets older.

The state chart in Figure 15 represents these visual states as internal structure for the 'alive' and 'dead' states.

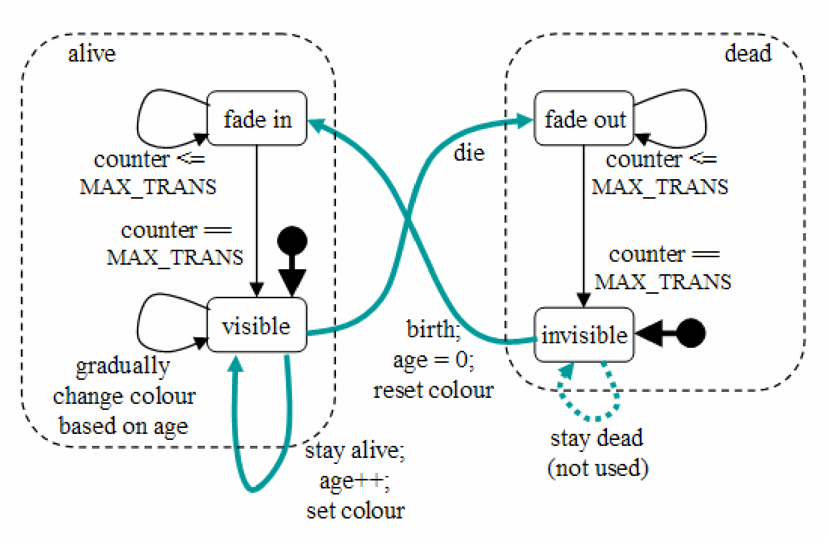


Figure 15. Life and Visual States for a Cell.

The transitions between the life states are drawn as thicker blue arrows in Figure 15, while the visual state transitions are normal thickness and black. The 'stay dead' transition is dotted to indicate that it's not used in my code.

When a cell is created it will be 'alive' or 'dead'. If 'alive', it'll start in the 'visible' visual state, if 'dead' then in the 'invisible' state. These two start states are indicated by thick arrows with black circles in Figure 15.

When the cell's life state changes from 'alive' to 'dead', the cell's visual state switches from 'visible' to 'fade out'. Subsequent visual change requests by CellsGrid trigger a fading away of the cell over CellsGrid.MAX_TRANS updates followed by a transition from 'fade out' to 'invisible'.

When the cell's state changes from 'dead' to 'alive', it switches it's visual state from 'invisible' to 'fade in', and its age is reset to 0. Subsequent visual change requests by CellsGrid trigger a gradually fade in of the cell over CellsGrid.MAX_TRANS updates until it switches from 'fade in' to 'visible'.

If a state change keeps the cell alive (i.e. it survives), then it's age is incremented, and at certain ages assigned a new color. This will trigger a series of visual transitions which gradually change the cell's color from its old value to the new one, spread over CellsGrid.MAX_TRANS updates.

Another aspect of the cell is its scene graph branch: the "cell" box in Figure 11 can be expanded to become Figure 16.

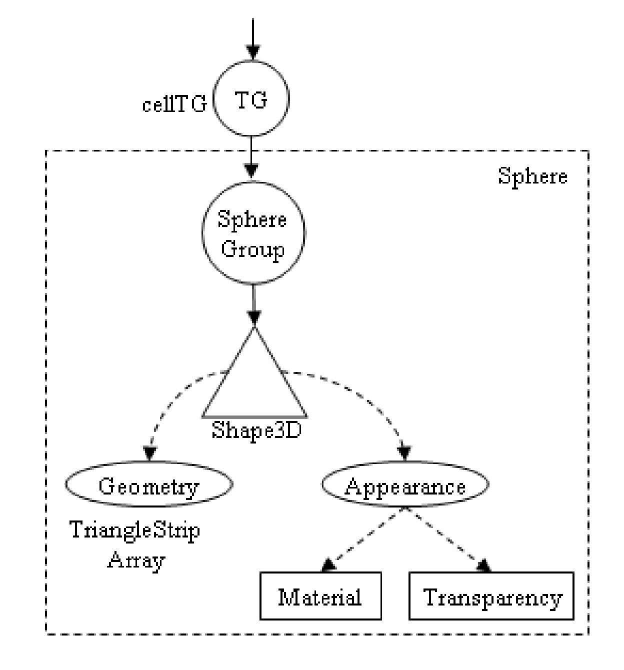


Figure 16. The Scene Graph for a Cell.

Much of the sphere's scene graph is created automatically by Java 3D's Sphere class, but the Material and Transparency node components need to be added by me.

9.1. Building the Cell's Scene Graph

The primary job of the Cell constructor is to build the scene graph shown in Figure 16.

```
// globals
// length of cell side (== diameter when the cell is a ball)
private final static float CELL_LEN = 0.5f;

// space between cells : a factor multiplied to CELL_LEN
private final static float SPACING = 1.5f;


private boolean isAlive;    // cell state information
private Appearance cellApp;
private TransformGroup cellTG;


public Cell(int x, int y, int z)
{
  isAlive = (Math.random() < 0.1) ? true : false;
    // it's more likely that a cell is initially dead (invisible)

  // create appearance
  cellApp = new Appearance();
  makeMaterial();
  setVisibility();

  // the cell shape as a sphere
  Sphere cellShape = new Sphere(CELL_LEN/2,
                          Sphere.GENERATE_NORMALS, cellApp);

  // fix cell's position
  Transform3D t3d = new Transform3D();
  double xPosn = x * CELL_LEN * SPACING;
  double yPosn = y * CELL_LEN * SPACING;
  double zPosn = z * CELL_LEN * SPACING;
  t3d.setTranslation( new Vector3d(xPosn, yPosn, zPosn) );

  // build scene branch
  cellTG = new TransformGroup();
  cellTG.setTransform(t3d);
  cellTG.addChild(cellShape);
}  // end of Cell()
```

The arguments of the Sphere constructor are the sphere's radius, the GENERATE_NORMALS flag to have Sphere add normals to the shape, and its appearance. Normals are needed so the sphere can reflect light. The other requirements for light to affect the shape's color are a Material component in the Appearance object, and that light is enabled in the Material object. Both of these are handled by setMaterial() described below.

The Sphere's geometry is stored in a Java 3D TriangleStripArray, which specifies the sphere as an array of connected triangles. Fortunately, the intricacies of building this mesh are dealt with by Sphere.

The Appearance node is a container for a variety of information, including geometry coloring, line, point, polygon, rendering, transparency, and texture attributes. The attributes needed here are added in setMaterial() and setVisibility().

The positioning of the sphere is done with the cellTG TransformGroup, using a Transform3D object containing a position. The position is based on the (x,y,z) coordinate passed in to the constructor, but scaled up to take account of the sphere's diameter (CELL_LEN) and the space between the cells (SPACING).

It's quite easy to change the cell shape by employing one of the other shape utility classes in Java 3D's com.sun.j3d.utils.geometry package (it contains Sphere, Box, Cone, and Cylinder). For example, cellShape can be defined as a cube with:

```
Box cellShape = new Box( CELL_LEN/2, CELL_LEN/2, CELL_LEN/2,
                  Box.GENERATE_NORMALS, cellApp);
```

Each box will have CELL_LEN long sides, include normals for lighting, and have the same appearance characteristics as the spherical cell. Figure 17 shows Life3D with boxes instead of spheres.

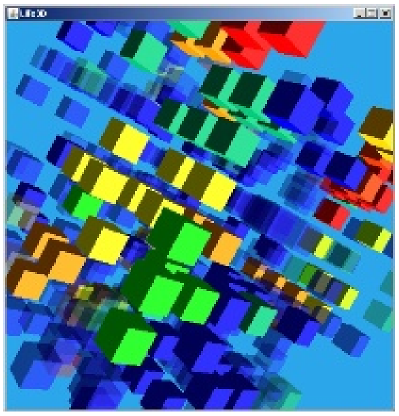


Figure 17. Life3D with Boxes.

9.2. Coloring the Cells

The coloring of a cell is complicated by the need to change it as the cell ages, and to reset it to blue when the cell is reborn.

```
// globals
private final static Color3f BLACK = new Color3f(0.0f, 0.0f, 0.0f);
private final static Color3f WHITE = new Color3f(0.9f, 0.9f, 0.9f);

// appearance elements
private Material material;
private Color3f cellCol, oldCol, newCol;


private void makeMaterial()
{
  cellCol = new Color3f();
  oldCol = new Color3f();
  newCol = new Color3f();

  // set material
  material = new Material(WHITE, BLACK, WHITE, WHITE, 100.f);
             // sets ambient, emissive, diffuse, specular, shininess

  material.setCapability(Material.ALLOW_COMPONENT_WRITE);
  material.setLightingEnable(true);
  resetColours();
  cellApp.setMaterial(material);
}  // end of makeMaterial()
```

The three Color3f objects are required for the color changes which gradually modify the cell's color (stored in cellCol) from the old color (stored in oldCol) to a new color (stored in newCol).

The Java 3D Material constructor controls what color a shape exhibits when lit by different kinds of lights:

```
Material mat = new Material(ambientColour, emissiveColour,
                      diffuseColour, specularColour, shininess);
```

The ambient color argument specifies the shape’s color when lit by ambient light: this gives the object a uniform color. The emissive color contributes the color that the shape produces itself (as for a light bulb); frequently, this argument is set to black (equivalent to off). The diffuse color is the color of the object when lit, with its intensity depending on the angle the light beams make with the shape’s surface.

The intensity of the specular color parameter is related to how much the shape reflects from its "shiny" areas. This is combined with the shininess argument, which controls the size of the reflective highlights.

The specular color is often set to white, matching the specular color produced by most objects in the real world.

In Life3D there are two directional lights, which create two shiny patches on each sphere.

The ambient and diffuse values in makeMaterial() are 'dummies': their real values are set in resetColours() and setMatColours():

```
// globals
private final static Color3f BLUE = new Color3f(0.0f, 0.0f, 1.0f);


private void resetColours()
// intialization of the material's color to blue
{
  cellCol.set(BLUE);
  oldCol.set(cellCol);   // old and new cols are blue as well
  newCol.set(cellCol);

  setMatColours(cellCol);
}  // end of resetColours()


private void setMatColours(Color3f col)
// the ambient color is a darker shade of the diffuse color
{
  material.setAmbientColor(col.x/3.0f, col.y/3.0f, col.z/3.0f);
  material.setDiffuseColor(col);
}  // end of setMatColours()
```

This unusual separation of tasks is due to the need to call resetColours() whenever the cell is reborn, to make it blue again. setMatColours() is called whenever the cell changes color. The darker ambient color helps to highlight the curvature of the spheres.

9.3. Setting the Cell's Visibility

A cell's visibility is controlled by a TransparencyAttributes instance. If the cell starts in the 'alive' state, that means it will be in the 'visible' visual state, while if it starts as 'dead' then it must be invisible. The capabilities of the TransparencyAttributes object must also be set to allow its value to change at runtime.

```
// globals
// possible visual states for a cell
private final static int INVISIBLE = 0;
private final static int FADE_IN = 1;
private final static int FADE_OUT = 2;
private final static int VISIBLE = 3;

private int visualState;


private void setVisibility()
{
  // let transparency value change at run time
  transAtt = new TransparencyAttributes();
  transAtt.setTransparencyMode(TransparencyAttributes.BLENDED);
  transAtt.setCapability(TransparencyAttributes.ALLOW_VALUE_WRITE);

  if (isAlive) {
    visualState = VISIBLE;
    transAtt.setTransparency(0.0f);     // opaque
  }
  else  { // dead so invisible
    visualState = INVISIBLE;
    transAtt.setTransparency(1.0f);   // totally transparent
  }

  cellApp.setTransparencyAttributes(transAtt);
}  // end of setVisibility()
```

The visual states of Figure 15 are encoded as values for the visualState integer.

9.4. Changing a Cell's Life State

CellsGrid updates the grid in two stages: first it evaluates the rules for all its cells, and stores a new state in each one. When all the cells have been examined, then the grid is updated in a second pass.

The Cell class helps with the initial evaluation stage by offering methods to get and set the cell's state:

```
// globals
private boolean isAlive, newAliveState;

public boolean isAlive()
{  return isAlive;  }

public void newAliveState(boolean b)
```

```
{  newAliveState = b;  }
```

The 'trick' is that the cell's isAlive boolean is *not* updated by a newAliveState() call, instead the value is stored in a second boolean, newAliveState. The updating of isAlive occurs during CellsGrid's second pass when it calls Cell.updateState().

Figure 15 shows the three transitions that change the cell's life state: 'die', 'birth', and 'stay alive' (they're the thick blue arrows). The code representing these transitions is located in updateState().

```
// globals
private int age = 0;

public void updateState()
{
  if (isAlive != newAliveState) {  // there's a state change
    if (isAlive && !newAliveState)  // alive --> dead (die)
      visualState = FADE_OUT;   // from VISIBLE
    else {  // dead --> alive (birth)
      visualState = FADE_IN;    // from INVISIBLE
      age = 0;    // reset age since born again
      resetColours();
    }
  }
  else { // current and new states are the same
    if (isAlive) {   // cell stays alive (survives)
      age++;   // get older
      ageSetColour();
    }
  }
}  // end of updateState()
```

The current cell state (in isAlive) and the new state (in newAliveState) are examined to determine which transition to apply.

ageSetColour() changes the newCol object if the age has reached certain hardwired values.

```
// global material colours
private final static Color3f RED = new Color3f(1.0f, 0.0f, 0.0f);
private final static Color3f ORANGE = new Color3f(1.0f, 0.5f, 0.0f);
private final static Color3f YELLOW = new Color3f(1.0f, 1.0f, 0.0f);
private final static Color3f GREEN = new Color3f(0.0f, 1.0f, 0.0f);
private final static Color3f BLUE = new Color3f(0.0f, 0.0f, 1.0f);


private void ageSetColour()
{
  if (age > 16)
    newCol.set(RED);
  else if (age > 8)
    newCol.set(ORANGE);
  else if (age > 4)
    newCol.set(YELLOW);
  else if (age > 2)
    newCol.set(GREEN);
  else
    newCol.set(BLUE);
} // end of ageSetColour()
```

9.5. Visual Changes to a Cell

Figure 10 shows that a state change by CellsGrid is followed by MAX_TRANS+1 visual changes; these are carried out by CellsGrid repeatedly calling Cell.visualChange(). There are five possible visual changes, represented by five thin black arrows in Figure 15, which correspond to the possible transitions between the visual states, 'fade in', 'visible', 'fade out', and 'invisible'.

The three looping transitions, which return to the same state they leave, are handled by an if-test inside visualChange().

```
public void visualChange(int transCounter)
{
  float transFrac = ((float)transCounter)/CellsGrid.MAX_TRANS;

  if(visualState == FADE_OUT)
    transAtt.setTransparency(transFrac);  // 1.0f is transparent
  else if (visualState == FADE_IN)
    transAtt.setTransparency(1.0f-transFrac);
  else if (visualState == VISIBLE)
    changeColour(transFrac);
  else if (visualState == INVISIBLE) {}
    // do nothing
  else
    System.out.println("Error in visualState");

  if (transCounter == CellsGrid.MAX_TRANS)
    endVisualTransition();
}  // end of visualChange()
```

transFrac is assigned a number between 0 and 1 based on the current transCounter value, which increases from 0 to CellsGrid.MAX_TRANS.

changeColour() sets the current cell's color to be a mix of its old and new colours (if the two are different).

```
private void changeColour(float transFrac)
{
  if (!oldCol.equals(newCol)) {  // if colours are different
    float redFrac = oldCol.x*(1.0f-transFrac) + newCol.x*transFrac;
    float greenFrac = oldCol.y*(1.0f-transFrac) + newCol.y*transFrac;
    float blueFrac = oldCol.z*(1.0f-transFrac) + newCol.z*transFrac;

    cellCol.set(redFrac, greenFrac, blueFrac);
    setMatColours(cellCol);
  }
}  // end of changeColour()
```

The mix uses the transFrac value so that as transCounter increases, the current color migrates towards the new color.

visualChange() ends with a test:

```
if (transCounter == CellsGrid.MAX_TRANS)
  endVisualTransition();
```

endVisualTransition() handles the two visual transitions not dealt with by the if-test in visualChange().

```
private void endVisualTransition()
{
  // store current color as both the old and new colours;
  // used when fading in and when visible
  oldCol.set(cellCol);
  newCol.set(cellCol);

  isAlive = newAliveState;   // update alive state

  if (visualState == FADE_IN)
    visualState = VISIBLE;
  else if (visualState == FADE_OUT)
    visualState = INVISIBLE;
}  // end of endVisualTransition()
```

The visual states are changed, and the new life state is finally stored as the cell's current life state in isAlive. The old and new colours are also updated to be the current color.

10. Time for Screensavers

JScreenSaver is a Windows screensaver loader which can execute Java programs. It was written by Yoshinori Watanabe, and is available from http://homepage2.nifty.com/igat/igapyon/soft/jssaver.html and http://sourceforge.net/projects/jssaver/.

JScreenSaver has three components:

- jssaver.scr, a SCR executable which Windows calls when it wants the JScreenSaver screensaver to start;
- jssaver.cfg, a text-based configuration file which specifies the JAR file that jssaver.scr will invoke;
- jssaver.jar, an example JAR file.

Windows XP looks for SCR executables in c:\windows\system32, so the three files should be moved there (XP also looks in c:\windows as well, so you actually have a choice).

The configuration file is very simple, it must include the name of the JAR and the class where main() is found. For example:

```
-classpath
jssaver.jar
SimpleRssSaver
```

The "-classpath" argument, the JAR name, and the class name must be on separate lines.

The configuration can be understood, by considering its command line equivalent:

```
$ java -classpath jssaver.jar SimpleRssSaver
```

This command is executed when the screensaver starts, or if the "Preview" button is pressed in Window's screensaver tab (see Figure 4). If the "Settings" button is pressed, then jssaver.scr calls the equivalent of the command line:

```
$ java -classpath jssaver.jar SimpleRssSaver -edit
```

The simplicity of Watanabe's interface means that jssaver.scr can invoke any JAR file, including ones using non-standard APIs such as Java 3D. The only requirement is that the application understands the "-edit" command line argument. Life3D illustrates one approach to this: the "-edit" option causes a configuration screen to be displayed, which edits properties in the life3DProps.txt file; this file is also utilized by the main application

10.1. Changing Life3D into a Screensaver

The application needs to be packaged up as a JAR:

```
$ javac *.java
$ jar cvfm Life3D.jar mainClass.txt *.class
```

mainClass.txt contains the name of the class containing main(), information that's added to the JAR's manifest:

```
Main-Class: Life3D
```

A nice feature of Watanabe's approach is that the JAR, the configuration screen, and main application can be tested separately from the SCR application, by executing their equivalent java command lines directly. So I tested the Life3D configuration screen with:

```
$ java -classpath Life3D.jar Life3D -edit
```

and the main application with:

```
$ java -classpath Life3D.jar Life3D
```

The jssaver.cfg file must be modified to refer to Life3D:

```
-classpath
Life3D.jar
Life3D
```

Also, Life3D.jar *and* life3DProps.txt must be copied to the same system directory as jssaver.scr and jssaver.cfg.

10.2. Problems with Screensavers

Many people have reported problems with screensavers in Windows XP; a good (but somewhat poorly organized) list of tips for getting things to work can be found at http://www.softwaretipsandtricks.com/windowsxp/articles/573/1/Windows-XP-FAQ-S, starting under the heading "Screen Savers".

10.3. The SaverBeans SDK

One drawback of JScreenSaver is that it only works with Windows. Another is that it doesn't support all of the Windows SCR functionality; most notable is that a preview of the screensaver isn't shown in the little computer screen at the top of the screensaver tab (see Figure 3).

A possible solution is the excellent SaverBeans Screensaver SDK (https://jdic.dev.java.net/documentation/incubator/screensaver/), a Java screensaver development kit for creating cross-platform screensavers. Currently Windows, Linux, and Solaris are supported, but not the Mac.

The programmer writes Java classes, and an XML description of the screensaver settings, and uses the SDK tools to produce a screensaver.

The main reason why I didn't choose SaverBeans is because it's aimed at writing 2D screensavers or ones using JOGL. Currently, there's no support for Java 3D applications.

Joshua Marinacci wrote an informative introduction to SaverBeans at the end of 2004, found at http://today.java.net/pub/a/today/2004/11/01/jdic2.html. There's also a SaverBeans forum at https://screensavers.dev.java.net/servlets/ForumMessageList?forumID=698. A collection of SaverBeans screensavers is available from http://screensavers.dev.java.net/.

SaverBeans is a part of the larger JDesktop Integration Components (JDIC) project (https://jdic.dev.java.net/), which aims to integrate native OS applications with Java. JDIC components include access to the OSes Web browser, system tray, file and system utilities, and a floating dock. Some elements of JDIC functionality have been added to Java 6, which I'll be discussing in the next chapter.

11. More Life Required?

Conway's Game of Life is deceptively simple *and* additive. Paul Calahan has written a non-technical introduction to the 2D game at math.com (http://www.math.com/students/wonders/life/life.html), with pointers to information on John Conway, and some great math books.

The Wikeipedia entry (http://en.wikipedia.org/wiki/Conway's_Game_of_Life) has a good explanation, fun animations, and lots of links (as you might expect). It employs a birth/survive ranges notation for the rules, which is a fine alternative to my birth and die ranges. One link worth following is to more information on cellular automata, http://en.wikipedia.org/wiki/Cellular_automata.

The "Conway's Game of Life FAQ" site (http://cafaq.com/lifefaq/index.php) explains matters well, and has a link to Martin Gardner's *Scientific American* article which started the Life craze.

There's a lot of free software available for running Conway's Game, such as Life32 by Johan Bontes (http://psoup.math.wisc.edu/Life32.html), and MCell by Mirek

Wojtowicz (http://www.mirwoj.opus.chelm.pl/ca/index.html), a general-purpose 1D and 2D cellular automata package.

The 3D aspects of Life have been explored in several papers by Carter Bays (see his list at http://www.cse.sc.edu/~bays/articles.html). His team developed a 3D Life applet with many controls, at http://www.cse.sc.edu/~bays/d4d4d4/guide.html.

Robert Trujillo has a comprehensive list of 3D cellular automata links at http://www.geocities.com/robisais/3dca.html. Two 3D versions of Life that I've tried are: Life3D by Michael Shelley (http://wwwcsif.cs.ucdavis.edu/~shelley/projects/), which is fun to play with, and the "Kaleidoscope of 3D Life" applet at http://www.people.nnov.ru/fractal/Life/Game.htm, which has very nice controls and includes examples. I got the idea of using birth and die ranges for my game rules from that site.