Chapter N3. Get a Life (the Java 6 Way)

The Life3D example is utilized again in this chapter, this time as a way to discuss four new Java 6 features useful for gaming:

- splashscreens;
- the system tray API;
- the desktop API;
- scripting integration.

This chapter's Life3D still displays a rotating 3D grid of cells which obey rules inspired by Conway's Game of Life (see Figure 1). The graphics code is mostly unchanged, so I won't need to explain it again.

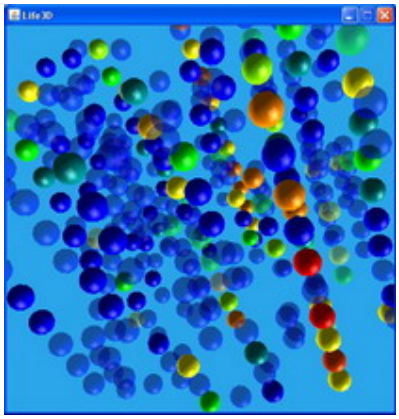

Figure 1. Life3D Once Again.

The two most significant things that I've removed from the application are the configuration window (and its properties configuration file), and full-screen rendering. The application starts with hardwired values for its window size, background color, grid rotation speed, and birth and die ranges.

The most visible new elements are a splashscreen which includes an animated clock picture (see Figure 2), and a popup menu accessible from a system tray icon (Figure 3).

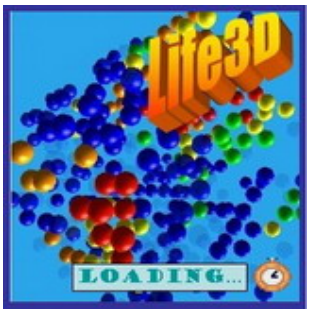

Figure 2. Life3D Splashscreen.

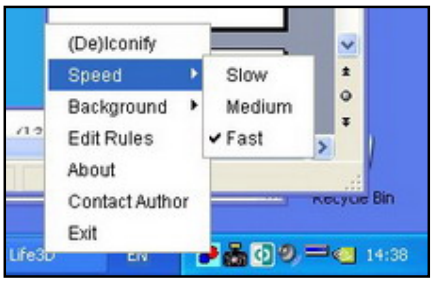

Figure 3. Life3D Popup Menu

The popup menu allows the user to modify Life3D in various ways at runtime. This includes the grid's rotation speed and the scene's background color. The application can also be iconified/deiconfied, and closed. The menu utilizes Java's new desktop API to let the user access the system's default web browser, a text editor, and e-mail client.

One of the exciting elements of Java 6 is its ability to communicate with scripting languages. In Life3D, JavaScript if-then rules to control how cells change over time, and the user can edit them at runtime to dynamically change the grid's behavior.

1. An Overview of the Life3D Classes

Figure 4 shows the class diagrams for the new version of Life3D. Only the public methods are shown, and I've left out the listeners used by the Life3D and Life3DPopup classes.

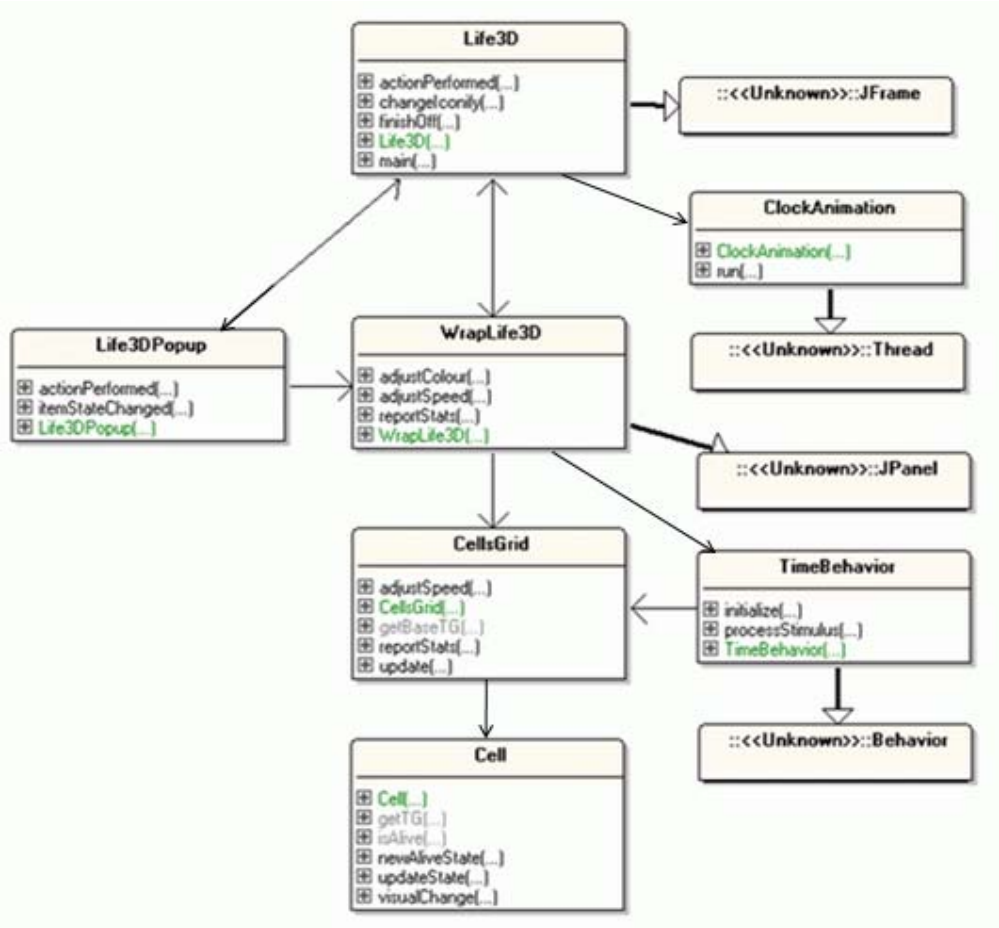

Figure 5. Class Diagrams for Life3D.

A quick comparison with the class diagrams for Life3D in chapter 2 reveals most of the high-level changes: the configuration classes are gone (Life3DConfig and LifeProperties), and Life3DPopup and ClockAnimator are new. Many classes appear in both versions: Cell and TimeBehavior are unchanged, but Life3D, WrapLife3D, and CellsGrid are modified.

Life3D starts by creating an instance of ClockAnimator, which displays the animated clock in the bottom right of the splashscreen (see Figure 2). It then creates the popup menu in the system tray using Life3DPopup (Figure 3), and renders the 3D scene with WrapLife3D.

The state and visual updates to the grid are controlled by periodic calls to CellsGrid.update() by TimeBehavior, and each cell in the grid is managed by a Cell object.

The user can modify Life3D's execution in various ways by selecting items from the popup menu in the system tray. Life3DPopup responds by calling methods in Life3D (changeIconify()) and WrapLife3D (adjustColour() and adjustSpeed()).

The user can edit a rules script file (rules.js), which is automatically reloaded by CellsGrid when it's been changed. rules.js contains JavaScript if-then rules, such as:

```
// rules.js
var beBorn = false;
var toDie = false;

if (states[4])
  beBorn = true;

if (states[5] && states[6] && states[17] && states[14])
  toDie = true;

if (numberLiving > 5)
  toDie = true;
```

I'll explain the meaning of these rules when I consider CellsGrid.

2. Making a Splash

Adding a splashscreen to an application is very easy – just add "-splash <image filename>" to the command line. For instance:

```
java -splash:lifeSplash.jpg LifeSplashTest
```

You can use GIF, JPEG, and PNG files for the splash image. Animated GIFs and transparency (in GIF or PNG files) are supported. The splash is positioned at the center of the screen, and displayed until the application's JFrame is made visible.

The splash effect can be made a little fancier by using the new Java 6 SplashScreen class to treat the splash as a drawing surface. The class also has methods to close the splashscreen, change the image, and get the image's position and size.

The LifeSplashTest class listed below is a test-rig for the SplashScreen functionality in my ClockAnimation class, which is utilized by Life3D.

```
public class LifeSplashTest
{
  public static void main(String args[])
  {
    // add a ticking clock to the splashscreen
    ClockAnimation ca = new ClockAnimation(7,7);
    ca.start();

    try {
      Thread.sleep(5000);   // 5 secs asleep
    }
    catch (InterruptedException ignored) {}

    // make a window
```

```
    JFrame frame = new JFrame("Life3D");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    JLabel label = new JLabel("Life3D has started", JLabel.CENTER);

    frame.add(label, BorderLayout.CENTER);
    frame.setSize(300, 95);

    // center the window
    Dimension screenDim =
              Toolkit.getDefaultToolkit().getScreenSize();
    Dimension winDim = frame.getSize();
    frame.setLocation( (screenDim.width-winDim.width)/2,
                      (screenDim.height-winDim.height)/2);

    frame.setVisible(true);  // splashscreen will close now
  } // end of main()

} // end of LifeSplashTest class
```

LifeSplashTest calls ClockAnimation to draw a ticking clock at the bottom right of the splashscreen. The clock's offset from the bottom right is determined by the x- and y- values passed to the ClockAnimation constructor (two 7's).

LifeSplashTest sleeps for 5 seconds before creating a JFrame holding a label. The call to Window.setVisible() at the end of main() causes the splashscreen to become invisible.

2.1. Animating a Clock

The animated clock effect is created using a series of eight GIFs with transparent backgrounds, stored in clock0.gif to clock7.gif; they're shown in Figure 6.

Figure 6. The Clock Images.

The main task of the ClockAnimation constructor is to load these images from the clocks/ subdirectory.

```
// globals
private static final int NUM_CLOCKS = 8;

private ImageIcon clockImages[];  // stores the clock images
private int currClock = 0;
private int clockWidth, clockHeight;
private int offsetX, offsetY;
                // offset of images from bottom right corner


public ClockAnimation(int oX, int oY)
```



```
{
  offsetX = oX; offsetY = oY;

  // load the clock images
  clockImages = new ImageIcon[NUM_CLOCKS];
  for (int i = 0; i < NUM_CLOCKS; i++)
    clockImages[i] = new ImageIcon(
         getClass().getResource("clocks/clock" + i + ".gif"));

  // get clock dimensions; assume all images are same size
  clockWidth = clockImages[0].getIconWidth();
  clockHeight = clockImages[0].getIconHeight();
}  // end of ClockAnimation()
```

The images are stored in an ImageIcon array. The call to the ImageIcon constructor uses getClass().getResource() to identify the full path to the images. This is only required if the class and images are going to be packaged up inside a JAR.

2.2. Drawing onto a Splash

Drawing onto the splash is a two-stage process, first a reference is obtained to the splashscreen using the SplashScreen.getSplashScreen() static method:

```
SplashScreen splash = SplashScreen.getSplashScreen();
```

Then a graphics context is obtained, which refers to the splashscreen's drawing surface:

```
Graphics2D g = splash.createGraphics();
```

Drawing can now commence.

It's also useful to know the size of the splash, so the drawing area can be constrained.

```
Dimension splashDim = splash.getSize();
```

Animation effects are coded using a loop, which should terminate when the splash disappears from the screen after the JFrame is made visible. This change can be detected by calling SplashScreen.isVisible().

2.3. Drawing the Clocks

The ClockAnimation run() method gets the splash's graphic context, then loops through the clock images, drawing each one in turn. When the end of the images array is reached, the drawing starts back at the beginning. There's a short pause between each drawing while run() sleeps.

```
// globals
private static final int MAX_REPEATS = 100;
private static final int TIME_DELAY = 250;   // ms


public void run()
{
  // get a reference to the splash
  SplashScreen splash = SplashScreen.getSplashScreen();
  if (splash == null) {
    System.out.println("No splashscreen found");
```

```
    return;
  }

  // get a reference to the splash's drawing surface
  Graphics2D g = splash.createGraphics();
  if (g == null) {
    System.out.println("No graphics context for splash");
    return;
  }

  /* calculate a (x,y) position for the clock images near the
     the bottom right of the splash image. */
  Dimension splashDim = splash.getSize();
  int xPosn = splashDim.width - clockWidth - offsetX;
  int yPosn = splashDim.height - clockHeight - offsetY;

  // start cycling through the images
  boolean splashVisible = true;
  for (int i = 0; ((i < MAX_REPEATS) && splashVisible); i++) {
    clockImages[currClock].paintIcon(null, g, xPosn, yPosn);
    currClock = (currClock + 1) % NUM_CLOCKS;

    // only update the splash if it's visible
    if (splash.isVisible())   // turns invisible when appl. starts
      splash.update();
    else
      splashVisible = false;

    try {
      Thread.sleep(TIME_DELAY);
    }
    catch (InterruptedException e) {}
   }
}  // end of run()
```

The two-step access to the graphic context at the start of run() detects problems, such as there not being a splash image.

The drawing loop can terminate in two ways: either when the splash becomes invisible (splashVisible is set to false) or after the loop has repeated MAX_REPEATS times. The second test ensures that the ClockAnimation thread will definitely stop after a certain period. Relying only on the application's visibility is a bit dangerous since the JFrame might never be made visible is there are initialization problems.

2.4. JAR Packaging

The command line specification of the splash image isn't possible if the application is wrapped up inside a JAR. Instead, the image's filename is added to the JAR's manifest.

The JAR is created in the standard way:

```
jar -cvfm LifeSplashTest.jar mainClass.txt *.class
                                          clocks lifeSplash.jpg
```

Note that the list of files include the clocks/ subdirectory and the splash image file, lifeSplash.jpg. mainClass.txt contains the extra manifest information:

```
Main-Class: LifeSplashTest
SplashScreen-Image: lifeSplash.jpg
```

The resulting JAR file, LifeSplashTest.jar, can be executed by double clicking it's icon, or by typing:

```
java -jar LifeSplashTest.jar
```

2.5. Adding ClockAnimation to Life3D

Life3D is called with the "-splash" command line argument:

```
java -splash:lifeSplash.jpg Life3D
```

The ClockAnimation thread is created at the beginning of the Life3D constructor:

```
public Life3D()
{
  super("Life3D");

  // start the clock animation that appears with the splashscreen
  ClockAnimation ca = new ClockAnimation(7,7);
  ca.start();

  // rest of the Life3D initialization
}
```

3. The Desktop API

The new java.awt.Desktop class allows a Java application to do the following:

- launch the host system's default browser with a specific Uniform Resource Identifier (URI);
- launch the host system's default e-mail client;
- launch applications to open, edit, or print files associated with those applications.

The Desktop API relies on the operating system's file-type associations for your machine to decide which applications are employed. On Windows, these associations map file extensions (e.g. ".doc") to programs (e.g. MS Word).

Unfortunately, the Desktop API doesn't support a mechanism for viewing or changing these associations, so the actual application started when a webpage is loaded, or e-mail created, or file opened, is outside of the programmer's control.

Life3D uses the Desktop API in two places: the Life3D class starts a browser to display the Java 3D home page if Java 3D isn't installed on the machine, and the popup menu lets the user open and edit text files, and send e-mail. I'll discuss the popup menu when I get to the Life3DPopup class.

3.1. Using the Desktop Browser

The Life3D constructor creates a Desktop instance, but only after checking if the functionality is supported.

```
// global
private Desktop desktop = null;

// in the Life3D constructor, get a desktop ref
if (Desktop.isDesktopSupported())
  desktop = Desktop.getDesktop();
```

This desktop reference is used to start a browser when the user clicks on the URL in the "Java 3D not installed" window (shown in Figure 7).

Figure 7. The "Java 3D not installed" Window.

The reportProb() method creates the window's label and button, just as in chapter 2. The difference is that an action listener is attached to the button if the desktop reference is capable of starting a browser.

```
// global
private JButton visitButton;


// inside reportProb()
String visitText = "<html><font size=+2>" +
                   "Visit https://java3d.dev.java.net/" +
                   "</font></html>";
visitButton = new JButton(visitText);
visitButton.setBorderPainted(false);

if ((desktop != null) &&
    (desktop.isSupported(Desktop.Action.BROWSE)) )
  visitButton.addActionListener(this);

reportPanel.add(visitButton);
```

The Desktop.isSupported() method tests whether a particular OS 'action' is available. The actions are defined as constants in the Desktop.Action class.

The action listener starts the system's browser with Desktop.browse().

```
// global
private static final String J3D_URL = "https://java3d.dev.java.net/";


public void actionPerformed(ActionEvent e)
{
  if (e.getSource() == visitButton) {
    try {   // launch browser
```



```
      URI uri = new URI(J3D_URL);
      desktop.browse(uri);
    }
    catch (Exception ex)
    { System.out.println(ex); }
  }
}  // end of actionPerformed
```

3.2. What Other Browser Capabilities are there?

Desktop.browse() is the only browser-related method in the Desktop class, which is somewhat disappointing when compared to the features in the JDesktop Integration Components (JDIC) project (http://jdic.dev.java.net/); JDIC was the inspiration for the Desktop class.

JDIC includes a browser listener class, which permits a page download to be monitored. The browser's navigation controls can be accessed to move backwards and forwards through the history, and for a page to be reloaded, or a download to be aborted. It's possible to access and change the HTML text of a page, and execute JavaScript against it. There's a WebBrowser class, which lets the rendering part of the native browser be embedded in the Java application as a Canvas; WebBrowser is intended to be a replacement for Java's aging JEditorPane.

JDIC can also manipulate OS file-type associations which means, for example, that it's possible to specify which browser is used to download a page.

A good JDIC overview by Michael Abernethy can be found at http://www-128.ibm.com/developerworks/java/library/j-jdic/index.html?ca=drs, or you can browse information at the JDIC site (https://jdic.dev.java.net/documentation/).

An informative article on JDIC's file-type associations by Jack Conradson is at http://java.sun.com/developer/technicalArticles/J2SE/Desktop/jdic_assoc/.

4. The System Tray

The system tray is the strip of icons on the right of the Windows task bar, and is sometimes called the notification area (see Figure 3). The tray is in a similar place in Linux (at least in KDE), and up on the menu bar in Mac OS X.

The new java.awt.SystemTray class represents the desktop's system tray. If tray functionality is offered by your OS (checkable with SystemTray.isSupported()), then it can be accessed with SystemTray.getSystemTray().

SystemTray holds one or more TrayIcon objects, which are added to the tray using the add() method, and removed with remove(). A TrayIcon object is represented by an icon in the tray, and may also have a text tooltip, an AWT popup menu, and assorted listeners.

When the tray icon is right-clicked it displays the popup menu. When the mouse hovers over the icon, the tooltip appears.

4.1. Creating Life3D's Popup Menu

Life3D delegates the hard work of managing the tray icon and popup menu to the Life3DPopup class, which is instantiated in Life3D's constructor:

```
// globals
private WrapLife3D w3d = null;
private Desktop desktop = null;


// in the Life3D constructor
Life3DPopup lpop = new Life3DPopup(this, w3d, desktop);
```

The constructor's references to Life3D (this) and the WrapLife3D object (w3d) are needed for method calls from the popup menu. It can call the Life3D method changeIconify() to iconify or deiconify the application, and finishOff() to terminate the application. It can call the WrapLife3D method adjustSpeed() to adjust the rotation speed of the grid, and adjustColour() to change the scene's background color.

The Desktop reference is required so the menu can start the OS'es default text editor and e-mail client.

4.2. The Menu Contents

Figure 3 shows the "Speed" submenu, while Figure 8 was snapped when the "Background" submenu was open.

Figure 8. The Popup Menu with "Background" Selected.

The differences in the menu's appearance between Figures 3 and 8 is due to my testing the application on differently configured WinXP machines.

The menu items are:

- '(De)Iconify'. Clicking on this iconifies or deiconfies the application. The user can also double click on the tray icon to do the same thing.
- 'Speed' changes the grid's rotation speed. There are three choices in the submenu: slow, medium, and fast (see Figure 3).
- 'Background' changes the scene's background color. Figure 8 shows the possible colors: blue, green, white, and black.
- 'Edit Rules' opens a text editor using the Desktop API so the user can modify rules.js. The rules are reloaded after the file has been saved.
- 'About' starts a text editor to display the contents of the about.txt file.

- 'Contact Author' opens a e-mail client with the Desktop API. The message will be sent to the address, adNOT@fivedots.coe.psu.ac.th.
- 'Exit' causes the Life3D application to exit.

4.3. Creating the TrayIcon

The TrayIcon object is created in makeTray() after a reference to the SystemTray has been obtained.

```
// globals
private Life3D applWindow;
private WrapLife3D w3d;
private Desktop desktop;    // for accessing desktop applications
private TrayIcon trayIcon;


public Life3DPopup(Life3D top, WrapLife3D w3d, Desktop d)
{
  applWindow = top;   // used to (de-)iconify, and closing the appl.
  this.w3d = w3d;     // used to change speed and background
  desktop = d;        // used for text editing & opening, and e-mail

  if (SystemTray.isSupported())
    makeTrayIcon();
  else
    System.err.println("System tray is currently not supported.");
}  // end of buildTray()


private void makeTrayIcon()
// the tray icon has an image, tooltip, and a popup menu
{
  SystemTray tray = SystemTray.getSystemTray();

  Image trayImage =
       Toolkit.getDefaultToolkit().getImage("balls.gif");
  PopupMenu trayPopup = makePopup();

  trayIcon = new TrayIcon(trayImage, "Life3D", trayPopup);
  trayIcon.setImageAutoSize(true);

  // double left clicking on the tray icon causes (de-)iconification
  ActionListener actionListener = new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {  applWindow.changeIconify();  }
  };
  trayIcon.addActionListener(actionListener);

  try {
    tray.add(trayIcon);
  }
  catch (AWTException e)
  { System.err.println("TrayIcon could not be added.");  }
}  // end of makeTrayIcon()
```

The TrayIcon object is initialized with an image (which appears in the system tray), a string for the tray icon's tooltip ("Life3D"), and a popup menu created with makePopup(). The balls.gif image is shown in Figure 9; it can be seen in the system tray in Figures 3 and 8.

Figure 9. The TrayIcon Image.

The TrayIcon action listener is triggered by a double-click of the left mouse button. In response, the changeIconify() method is called over in Life3D:

```
// in the Life3D class
public void changeIconify()
// iconify or deiconify the application
{
  if ((getState() & JFrame.ICONIFIED) == JFrame.ICONIFIED)
    setState(Frame.NORMAL);    // deiconify
  else
    setState(Frame.ICONIFIED);  // iconify
} // end of changeIconify()
```

It toggles the window between iconified and normal size.

4.4. Building the Popup Menu

A disappointing aspect of the TrayIcon class is that its popup menu must be built with the old AWT components: PopupMenu, MenuItem, and CheckboxMenuItem, rather than the modern Swing versions, JPopupMenu, JMenuItem, and JCheckBoxMenuItem. There's also no equivalent of JRadioButtonMenuItem, which means that I had to code up radio button group behavior in Life3DPopup.

By comparison, JDIC's system tray API does use Swing components (http://jdic.dev.java.net/).

The PopupMenu object consists of a series of MenuItems except for the "Speed" and "Background" submenus, which are represented by Menu objects containing CheckboxMenuItems.

```
// globals
private static final int DEFAULT_SPEED = 2;   // fast
private static final int DEFAULT_COLOUR = 0;  // blue

private static final String[] speedLabels =
                              {"Slow", "Medium", "Fast"};
private static final String[] colourLabels =
                     {"Blue", "Green", "White", "Black"};

private MenuItem iconifyItem, rulesItem, aboutItem,
                 authorItem, exitItem;
private CheckboxMenuItem[] speedItems;   // for speed values
private CheckboxMenuItem[] colourItems;  // for background colors
```

```
private PopupMenu makePopup()
{
  PopupMenu trayPopup = new PopupMenu();

  iconifyItem = new MenuItem("(De)Iconify");
  iconifyItem.addActionListener(this);
  trayPopup.add(iconifyItem);

  // --------------- speed items ----------------------

  Menu speedMenu = new Menu("Speed");
  trayPopup.add(speedMenu);

  speedItems = new CheckboxMenuItem[speedLabels.length];
  for(int i=0; i < speedLabels.length; i++) {
    speedItems[i] = new CheckboxMenuItem( speedLabels[i] );
    speedItems[i].addItemListener(this);
    speedMenu.add( speedItems[i] );
    if (w3d == null)   // if no 3D scene, then cannot change speed
      speedItems[i].setEnabled(false);
  }
  speedItems[DEFAULT_SPEED].setState(true);   // default is 'fast'


  // ------------ background color items ---------------

  Menu bgMenu = new Menu("Background");
  trayPopup.add(bgMenu);

  colourItems = new CheckboxMenuItem[colourLabels.length];
  for(int i=0; i < colourLabels.length; i++) {
    colourItems[i] = new CheckboxMenuItem( colourLabels[i] );
    colourItems[i].addItemListener(this);
    bgMenu.add( colourItems[i] );
    if (w3d == null)   // if no 3D, then cannot change background
      colourItems[i].setEnabled(false);
  }
  colourItems[DEFAULT_COLOUR].setState(true);   // default is 'blue'


  // -----------------------------

  rulesItem = new MenuItem("Edit Rules");
  if ((desktop != null)&&(desktop.isSupported(Desktop.Action.EDIT)) )
    rulesItem.addActionListener(this);  // if text editing possible
  else
    rulesItem.setEnabled(false);
  trayPopup.add(rulesItem);

  aboutItem = new MenuItem("About");
  if ((desktop != null)&&(desktop.isSupported(Desktop.Action.OPEN)) )
    aboutItem.addActionListener(this);  // if text opening possible
  else
    aboutItem.setEnabled(false);
  trayPopup.add(aboutItem);

  authorItem = new MenuItem("Contact Author");
  if ((desktop != null)&&(desktop.isSupported(Desktop.Action.MAIL)) )
    authorItem.addActionListener(this);   // if e-mail possible
  else
```

```
    authorItem.setEnabled(false);
  trayPopup.add(authorItem);

  MenuShortcut exitShortcut = new MenuShortcut(KeyEvent.VK_X,true);
    // ctrl-shift-x on windows
  exitItem = new MenuItem("Exit", exitShortcut);
  exitItem.addActionListener(this);
  trayPopup.add(exitItem);

  return trayPopup;
}  // end of makePopup()
```

The CheckboxMenuItems utilized in the "Speed" and "Background" submenus are meant to act as radio buttons, so that the selection of one will disable the previous selection. speedItems[] stores all the checkboxes for the "Speed" menu, and colourItems[] all the background color checkboxes. These arrays are manipulated by the itemStateChanged() method to simulate the behavior of radio buttons.

The "Edit Rules", "About", and "Contact Author" menu items require the Desktop API, so the GUI code checks for it's presence before attaching the action listener. It also uses Desktop.isSupported() to determine whether the API supports the action required by the menu item: "Edit Rules" needs text editing, "About" utilizes text viewing, and "Contact Author" requires an e-mail client. If the checking fails, then the menu item is disabled.

The "Exit" menu is assigned a key combination shortcut (ctrl-shift-x) as a MenuShortcut object. Unfortunately, it doesn't work in the version of Java I'm using, v.1.6.0-beta 2. (By the way, 'ctrl' is replaced by the 'command' key on the Mac.)

The five menu items ("(De)Iconify", "Edit Rules", "About", "Contact Author", and "Exit" use an action listener, while the "Speed" and "Background" checkboxes use an item listener.

4.5. Listening for Actions

The actionPerformed() method in Life3DPopup reacts differently depending on which of the five menu items activated it.

```
// globals
// rules file, author e-mail, and about file
private static final String SCRIPT_FNM = "rules.js";
private static final String AUTHOR_EMAIL =
                              "adNOT@fivedots.coe.psu.ac.th";
private static final String ABOUT_FNM = "about.txt";


public void actionPerformed(ActionEvent e)
{
  MenuItem item = (MenuItem) e.getSource();
            // all the actions come from MenuItems

  if (item == iconifyItem)
    applWindow.changeIconify();
  else if (item == rulesItem)
    launchFile(SCRIPT_FNM, Desktop.Action.EDIT);
  else if (item == aboutItem)
```

```
    launchFile(ABOUT_FNM, Desktop.Action.OPEN);
  else if (item == authorItem)
    launchMail(AUTHOR_EMAIL);
  else if (item == exitItem)
    applWindow.finishOff();
  else
    System.out.println("Unknown Action Event");
}  // end of actionPerformed()
```

The "(De)Iconify" menu item is processed by calling changeIconify() in Life3D, duplicating the action of double left clicking on the tray icon.

The "Edit Rules" and "About" menu items are processed by calling launchFile() which uses the Desktop API to start a text editor, either to edit or view the supplied file.

The "Contact Author" menu item triggers a call to launchMail() which starts the e-mail client with the address argument.

When the "Exit" item is pressed, finishOff() is called in Life3D:

```
// in Life3D
// global
private WrapLife3D w3d = null;

public void finishOff()
{ if (w3d != null)
    w3d.reportStats();
  System.exit(0);
}
```

The application is terminated, after printing statistics related to the 3D scene.

```
// in WrapLife3D
// global
private CellsGrid cellsGrid;

public void reportStats()
{  cellsGrid.reportStats(); }
```

The necessary information is located in CellsGrid, so the reportStats() call is passed along to that class:

```
// in CellsGrid
// globals used in the timing code
private long totalTime = 0;
private long numUpdates = 0;


public void reportStats()
{ int avgTime = (int) ((totalTime/numUpdates)/1000000);  // in ms
  System.out.println("Average update time: " + avgTime +
                     "ms;  no. updates: " + numUpdates);
}
```

reportStats() prints the average update time, which measures how long it takes for a new grid generation to be calculated in CellsGrid's update() method. This information is useful for adjusting the time delay used by TimeBehavior. For instance, if an update takes 100ms, then there's no point having TimeBehavior fire off update() calls every 50ms; it's interval should be 100ms, or longer

The reported average varies quite a lot on my slowish WinXP test machines, with values ranging between 50 and 100ms. One factor is the complexity of the script being executed.

4.6. Using a Text Editor

launchFile() opens up a text file in the OS'es default text editor, after checking that the file exists. Errors are reported as popup messages using TrayIcon.displayMessage().

```
private void launchFile(String fnm, Desktop.Action action)
{
  File f = null;
  try {
    f = new File(fnm);
  }
  catch (Exception e)
  { trayIcon.displayMessage("File Error",
       "Could not access " + fnm, TrayIcon.MessageType.ERROR);
    return;
  }

  if (!f.exists()) {
    trayIcon.displayMessage("File Error",
      "File " + fnm + " does not exist", TrayIcon.MessageType.ERROR);
    return;
  }

  if (action == Desktop.Action.OPEN)
    openFile(fnm, f);
  else if (action == Desktop.Action.EDIT)
    editFile(fnm, f);
}  // end of launchFile()
```


Figure 10 shows the popup error message when about.txt (displayed by the "About" menu item) cannot be found.

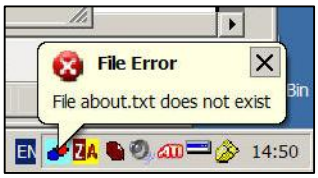

Figure 10. Popup Error Message.

openFile() and editFile() are quite similar, differing in the File tests they apply before calling the Desktop API. openFile() checks if the file can be read (with File.canRead()), while editFile() makes sure that the file can be edited (with File.canWrite()). The editFile() method:

```
private void editFile(String fnm, File f)
{
  if (!f.canWrite()) {
    trayIcon.displayMessage("File Error",
        "Cannot write to file " + fnm, TrayIcon.MessageType.ERROR);
    return;
  }
  else {  // can write
    try {
      desktop.edit(f.getAbsoluteFile());
    }
    catch (Exception e)
    {  trayIcon.displayMessage("File Error",
        "Cannot edit file " + fnm, TrayIcon.MessageType.ERROR);
    }
  }
}  // end of editFile()
```

editFile() employs Desktop.edit(), and requires an absolute filename. openFile() calls Desktop.open() in a similar manner.

4.7. Launching an E-mail Client

launchMail() converts the supplied e-mail address ("adNOT@fivedots.coe.psu.ac.th") into a "mailto" URI, then calls Desktop.mail().

```
private void launchMail(String addr)
{
  try {
    URI uriMail = new URI("mailto", addr +
                          "?SUBJECT=Life 3D Query", null);
    desktop.mail(uriMail);
  }
  catch (Exception e)
  {  trayIcon.displayMessage("E-mail Error",
        "Cannot send e-mail to " + addr, TrayIcon.MessageType.ERROR);
  }
}  // end of launchMail()
```

A "mailto" URI can include header lines after the e-mail address. I added a "SUBJECT" header which initializes the subject line in the e-mail client (see Figure 11).

Figure 11. The E-mail Client.

Details on the "mailto" URI can be found in its Internet Society specification at ftp://ftp.isi.edu/in-notes/rfc2368.txt.

4.8. Listening for the Checkboxes

The "Speed" and "Background" menus are populated with CheckboxItems, and Life3DPopup implements an itemStateChanged() method to listens to them.

```
// globals
private static final String[] speedLabels =
                          {"Slow", "Medium", "Fast"};
private static final String[] colourLabels =
                          {"Blue", "Green", "White", "Black"};

private CheckboxMenuItem[] speedItems;   // for speed values
private CheckboxMenuItem[] colourItems;  // for background colors

private WrapLife3D w3d;


public void itemStateChanged(ItemEvent e)
{
  CheckboxMenuItem item = (CheckboxMenuItem) e.getSource();
  int posn = -1;

  // speed checkbox items
  if ((posn = findItem(item, speedItems)) != -1) {
    switchOffItems(posn, speedItems);
    if (w3d != null)
      w3d.adjustSpeed( speedLabels[posn] );
  }
  // color checkbox items
  else if ((posn = findItem(item, colourItems)) != -1) {
    switchOffItems(posn, colourItems);
    if (w3d != null)
```

```
      w3d.adjustColour( colourLabels[posn] );
  }
  else
    System.out.println("Unknown Item Event");
}  // end of itemStateChanged()
```

findItem() searches for the selected CheckboxItem in the supplied array of CheckboxItems, returning its index position, or -1 if the item isn't found.

If the item is found then switchOffItems() iterates through the CheckboxItems array, setting the state of the other items to false (i.e. unchecking them). In this way the selection of an item causes any previously checked items to be unchecked.

If a "Speed" checkbox was selected, then adjustSpeed() is called in WrapLife3D, with a speed string argument ("Slow", "Medium", or "Fast"):

```
// in WrapLife3D
public void adjustSpeed(String speedStr)
{  cellsGrid.adjustSpeed(speedStr);  }
```

CellsGrid uses the speed to affect its rotation:

```
// in CellsGrid
// globals
private static final double ROTATE_AMT = Math.toRadians(4);
private double turnAngle = ROTATE_AMT;


public void adjustSpeed(String speedStr)
{
  if (speedStr.equals("Slow"))
    turnAngle = ROTATE_AMT/4;
  else if (speedStr.equals("Medium"))
    turnAngle = ROTATE_AMT/2;
  else  // fast --> large rotation
    turnAngle = ROTATE_AMT;
} // end of adjustSpeed()
```

A faster speed string is converted into a larger rotation angle, which makes the grid turn faster.

If a "Background" checkbox is selected in Life3DPopup, then adjustColour() is called in WrapLife3D with a color string argument ("Blue", "Green", "White", or "Black"):

```
// in WrapLife3D
public void adjustColour(String colourStr)
{
  if (colourStr.equals("Blue"))
    back.setColor(0.17f, 0.65f, 0.92f);    // sky blue color
  else if (colourStr.equals("Green"))
    back.setColor(0.5f, 1.0f, 0.5f);       // grass color
  else if (colourStr.equals("White"))
    back.setColor(1.0f, 1.0f, 0.8f);       // off-white
  else   // black by default
    back.setColor(0.0f, 0.0f, 0.0f);       // black
```

```
}  // end of adjustColour()
```

5. Scripting in Java 6

One of the most interesting (and powerful) new features of Java 6 is it's support for scripting languages such as JavaScript. Java 6 includes the Mozilla Rhino engine (http://www.mozilla.org/rhino/), an implementation of JavaScript. Other scripting languages can be easily added to the JRE, so long as they conform to the JSR-233 scripting specification (http://www.jcp.org/en/jsr/detail?id=223). The list of conformant engines at https://scripting.dev.java.net/ includes AWK, Python, Ruby, Scheme, and Tcl, and is growing rapidly.

There are several reasons for employing scripts in a Java application:

- scripts are more powerful and flexible than having the user interact with the application through configuration screens or property files;
- scripts can easily interact with the application's Java code, so can add or extend its functionality;
- writing application add-ons with scripting languages is generally less complex than using Java, especially if the coder is unfamiliar with Java;
- many scripting languages have large libraries, which can be useful for the Java application.

Adding scripting capabilities to an application isn't always necessary, but if the application is meant to be extensible, giving the user the ability to add or change substantial features, then scripting is an ideal approach.

A downside of using scripts is their speed, a problem that can be partly overcome by compilation.

Life3D utilizes Java 6's built-in JavaScript support (i.e. the Rhino engine) to allow the user to modify cell rules. The rules are simple if-then statements, which hide the complexities of the CellsGrid class. As soon as the rules file has been edited and saved (via the "Edit Rules" popup menu item), the modified rules start to be applied, and the behavior of the cells changes. To improve speed, the rules script is compiled.

Before I explain CellsGrid's scripting elements, I'll go through several smaller scripting examples.

5.1. Executing a Script

Executing a script is a three step process:

1. Create a ScriptEngineManager object.
2. Retrieve a scripting engine of your choice, as a ScriptEngine object.
3. Evaluate the script using the engine.

The ScriptingEx1 class shows these steps:

```
import java.io.*;
import javax.script.*;
```

```
public class ScriptingEx1
{
  public static void main(String[] args)
  {
    // create a script engine manager (step 1)
    ScriptEngineManager factory = new ScriptEngineManager();

    // get the JavaScript engine (step 2)
    ScriptEngine engine = factory.getEngineByName("js");

    try {
      // evaluate a JavaScript string (step 3)
      engine.eval("println('hello world')");
    }
    catch(ScriptException e)
    {  System.out.println(e);  }
  } // end of main()

} // end of ScriptingEx1 class
```

There's several ways of obtaining an engine at step 2, via it's name, file extension, or even MIME type.

The output from ScriptingEx1 is:

```
> java ScriptingEx1
hello world
```

It's not much more difficult to evaluate code loaded from a file, as ScriptingEx2 illustrates.

```
public class ScriptingEx2
{
  public static void main(String[] args)
  {
    // create a script engine manager
    ScriptEngineManager factory = new ScriptEngineManager();

    // create JavaScript engine
    ScriptEngine engine = factory.getEngineByName("js");

    // evaluate JavaScript code from a file
    evalScript(engine, "hello.js");
  } // end of main()


  static private void evalScript(ScriptEngine engine, String fnm)
  {
    try {
      FileReader fr = new FileReader(fnm);
      engine.eval(fr);
      fr.close();
    }
    catch(FileNotFoundException e)
    {  System.out.println(fnm + " not found");  }
    catch(IOException e)
    {  System.out.println("IO problem with " + fnm);  }
```

```
    catch(ScriptException e)
    {  System.out.println("Problem evaluating script in " + fnm);  }
    catch(NullPointerException e)
    {  System.out.println("Problem reading script in " + fnm);  }
  }  // end of evalScript()

} // end of ScriptingEx2 class
```

evalScript() uses a variant of the ScriptEngine.eval() method with a FileReader source. It's made longer by having to deal with multiple possible exceptions.

The hello.js script file contains one line:

```
println('hello world');
```

The execution of ScriptingEx2:

```
> java ScriptingEx2
hello world
```

5.2. Communicating with a Script

Java can pass data into a script, get a result back, and retrieve the changed data. The following example passes an integer and an array of integers into the sums.js script. The communication is achieved by storing variable bindings in the script engine.

```
public static void main(String[] args)
{
  // create a script engine manager
  ScriptEngineManager factory = new ScriptEngineManager();

  // create JavaScript engine
  ScriptEngine engine = factory.getEngineByName("js");

  int age = 40;
  int[] nums = { 1, 2, 3, 4, 5, 6, 7};   // sum is 28

  // pass values to JavaScript
  engine.put("age", age);
  engine.put("nums", nums);

  // evaluate script from a file
  evalSummer(engine, "sums.js");

  // more code, explained below
} // end of main()
```

The integer and array bindings are stored in the engine through calls to ScriptEngine.put():

```
engine.put("age", age);
engine.put("nums", nums);
```

The age integer is copied into an age variable inside the script, while a *reference* to the nums[] array (an object) is assigned to the nums[] array inside the script.

sums.js compares the input age with the sum of the numbers in nums[], and return true or false depending on if the age is bigger.

```
// sums.js
println("age = " + age);
println("nums[6] = " + nums[6]);

var sum = 0;
for(var i=0; i < nums.length; i++)
  sum += nums[i];

println("sum = " + sum);

age > sum;
```

The boolean result of the final expression (age > sum) becomes the return result for the script.

evalSummer() evaluates the sums.js script:

```
static private void evalSummer(ScriptEngine engine, String fnm)
// evaluate script from a file
{
  boolean isBigger = false;

  try {
    FileReader fr = new FileReader(fnm);
    isBigger = (Boolean) engine.eval(fr);
           // converts returned Object to a boolean
    fr.close();
  }
  catch(FileNotFoundException e)
  {  System.out.println(fnm + " not found");  }
  catch(IOException e)
  {  System.out.println("IO problem with " + fnm);  }
  catch(ScriptException e)
  {  System.out.println("Problem evaluating script in " + fnm);  }
  catch(NullPointerException e)
  {  System.out.println("Problem reading script in " + fnm);  }

  // javascript number mapped to Double
  double sum = (Double) engine.get("sum");
  System.out.println("(java) sum = " + sum);

  System.out.println("age is bigger = " + isBigger);
 }  // end of evalSummer()
```

This method is quite similar to evalScript() in ScriptingEx2, but shows two ways of getting data out of a script.

ScriptEngine.eval() returns an Object storing the value of the last executed expression. The last expression in sums.js is a boolean, so evalSummer() converts the Object to a Boolean, and then to a boolean assigned to isBigger.

The other way of obtaining data is with ScriptEngine.get() which copies the value of a specified script variable. The call to get() in evalSummer() retrieves the sum variable from sums.js as a Double. It's cast to a double, and assigned to sum.

The call to evalSummer() produces the following output:

```
age = 40
nums[6] = 7
sum = 28
(java) sum = 28.0
age is bigger = true
```

The output shows that the Java age value and the nums[] array are accessible to the script, and that the script's boolean result and its sum value are accessible back in the Java code.

The meaning of ScriptEngine.put() depends on if the value is a simple type (e.g. int, boolean) or an object (e.g. an array, String). The script gets a *copy* of the value of a simple type, such as age, but a *reference* to an object, such as num[]. This means that subsequent changes to the object will be visible to the Java code and the script without the need for put() or get() calls. This is illustrated by a second call to evalSummer():

```
// change age and nums[] values, and re-evaluate script
age = 50;       // increased by 10
nums[6] = 27;   // increased by 20, so sum should be 48
evalSummer(engine, "sums.js");
```

The resulting output from evalSummer():

```
age = 40
nums[6] = 27
sum = 48
(java) sum = 48.0
age is bigger = false
```

The script 'sees' the change to nums[], but not the new age value (which it thinks is still 40).

ScriptEngine.put() must be called again to update the age copy used by the script. This is shown by the third call to evalSummer():

```
// explicitly copy age value into engine
engine.put("age", age);
evalSummer(engine, "sums.js");
```

The output:

```
age = 50
nums[6] = 27
sum = 48
(java) sum = 48.0
age is bigger = true
```

The script uses the new age value (50).

5.3. Speeding Things Up

If a script is going to be used repeatedly, then it's execution speed can be improved by compilation. This capability is an optional part of JSR 223, but available in Java 6's Rhino engine. The next example executes a compiled version of sums.js.

```
public static void main(String[] args)
{
  // create a script engine manager
  ScriptEngineManager factory = new ScriptEngineManager();

  // create JavaScript engine
  ScriptEngine engine = factory.getEngineByName("js");

  // load and compile the script
  CompiledScript cs = loadCompile(engine, "sums.js");

  int age = 40;
  int[] nums = { 1, 2, 3, 4, 5, 6, 7};   // sum is 28

  // pass values to script
  engine.put("age", age);
  engine.put("nums", nums);

  // evaluate compiled script
  evalCSummer(cs, engine);
} // end of main()
```

Script compilation requires a Compilable version of the engine, and the compiled script is stored as a CompiledScript object.

```
static private CompiledScript loadCompile(ScriptEngine engine,
                                                    String fnm)
// load the script from a file, and compile it
{
  Compilable compEngine = (Compilable) engine;

  CompiledScript cs = null;
  try {
    FileReader fr = new FileReader(fnm);
    cs = compEngine.compile(fr);
    fr.close();
  }
  catch(FileNotFoundException e)
  {  System.out.println(fnm + " not found");  }
  catch(IOException e)
  {  System.out.println("Could not read " + fnm);  }
  catch(ScriptException e)
  {  System.out.println("Problem compiling script in " + fnm);  }
  catch(NullPointerException e)
  {  System.out.println("Problem reading script in " + fnm);  }

  return cs;
}  // end of loadCompile()
```

A compiled script is executed using CompiledScript.eval(), but data is passed to the script, and retrieved from it, with the same mechanisms employed in ordinary scripts.

```
static private void evalCSummer(CompiledScript cs,
                                ScriptEngine engine)
// evaluate compiled script
{
  boolean isBigger = false;
  try {
    isBigger = (Boolean) cs.eval();   // converts Object to boolean
  }
  catch(ScriptException e)
  {  System.out.println("Problem evaluating script");  }

  // js number mapped to Double
  double sum = (Double) engine.get("sum");
  System.out.println("(java) sum = " + sum);

  System.out.println("age is bigger = " + isBigger);
  System.out.println();
}  // end of evalCSummer()
```

5.4. Calling Script Functions

It's useful to organize longer, more complex scripts into functions, and then call specific script functions from the Java side. This is possible with the Invocable interface, another optional part of the JSR 223 specification, which is supported by the Rhino engine.

First, the script is stored in the engine, then its functions can be called after casting the engine to an Invocable object.

```
// store the script function
engine.eval("function sayHello(name) {" +
            " println('Hello ' + name);" +
            "}");

// invoke the function
Invocable invocableEngine = (Invocable) engine;
Object[] fnArgs = {"Andrew"};  // function argument
try {
  invocableEngine.invokeFunction("sayHello", fnArgs);
}
catch(NoSuchMethodException e)
{  System.out.println(e);  }
catch(ScriptException e)
{  System.out.println(e);  }
```

If a function requires input arguments, they must be supplied in an Object[] array. In the example, sayHello() takes a single string argument; I supplied "Andrew", and the function printed "Hello Andrew".

5.5. Letting a Script Use Java

I've concentrated on explaining how Java can call scripts, but it's possible for scripts to use Java. The Rhino engine has an importPackage() function, which imports Java packages so that Java objects can be created. The following code snippet shows the use of the Date class from the java.util package.

```
engine.eval("importPackage(java.util); " +
            "today = new Date(); " +
            "println('Today is ' + today);");
```

Also, if an object reference is passed to a script, then it's class methods can be called. The following script changes a String object to uppercase using String.toUpperCase(), and prints it.

```
String name = "Andrew Davison";
engine.put("name", name);

engine.eval("nm2 = name.toUpperCase();" +
            "println('Uppercase name: ' + nm2)");
```

5.6. More Scripting Information

Two informative articles on Java 6 scripting:

- *Scripting for the Java Platform*, John O'Conner, July 2006 http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/
- *The Mustang Meets the Rhino: Scripting in Java 6*, John Ferguson Smart, April 2006 http://www.onjava.com/pub/a/onjava/2006/04/26/mustang-meets-rhino-java-se-6-scripting.html

An older overview of scripting, presented at JavaOne 2005:

- *Scripting for the Java Platform*, Mike Grogan, A. Sundararajan, Joe Wang, June 2005 http://developers.sun.com/learning/javaoneonline/2005/coreplatform/TS-7706.pdf

It's possible to create new scripting languages for Java, as long as they follow the JSR 223 specification. The following article outlines how to create a boolean expressions language:

- *Build your own Scripting Language for Java*, Chaur Wu, April 2006 http://www.javaworld.com/javaworld/jw-04-2006/jw-0424-scripting.html

A good online introduction to JavaScript, by Simon Willison, can be found at http://simon.incutio.com/archive/2006/03/07/etech. The Wikipedia page on JavaScript has more overview and background, together with many links (http://en.wikipedia.org/wiki/JavaScript).

6. Scripting in Life3D

All the Life3D scripting code is located in CellsGrid, which performs the same tasks as the version in chapter 2. It manages a grid of Cell objects, and TimeBehavior periodically calls it's update() method to update that grid. An update either triggers a state change or a visual change.

The state of the grid's cells can be changed either by applying birth and die ranges (as in chapter 2) or by employing rules loaded (and compiled) from a script. This version of CellsGrid doesn't use a properties configuration file, so the birth and die ranges are fixed.

As you saw in Life3DPopup, the user can modify the rules at run time via the system tray popup menu. When the rules file is saved, it's reloaded and recompiled by CellsGrid.

Timings shows that the compiled script rules are about 10-16 times slower than using birth and die ranges: an update using rules takes about 60-100 ms on average, versus around 6 ms for the ranges.

I won't explain all of CellsGrid again, only the parts affecting by the addition of the script rules.

6.1. Initializing the Grid

The CellsGrid constructor has to decide whether to use birth and die ranges or script rules, and then initialize the required data structures.

```
// globals
private static final String SCRIPT_FNM = "rules.js";
                                // holds the life rules
private boolean usingScript;


public CellsGrid()
{
  // will the rules come from a script or be predefined ranges?
  usingScript = hasScriptFile(SCRIPT_FNM);
  if (usingScript)
    initScripting();
  else
    initRanges();

  // more initailization code
}  // end of CellsGrid
```

hasScriptFile() checks if the script file is available, and assigns a File object to the scriptFile global.

```
// global
private File scriptFile;


private boolean hasScriptFile(String fnm)
{
```

```
  scriptFile = null;
  try {
    scriptFile = new File(fnm);
  }
  catch (NullPointerException e) {
    System.out.println("Could not access " + fnm);
    return false;
  }
  if (!scriptFile.exists()) {
    System.out.println("No script file " + fnm);
    return false;
  }
  return true;
}  // end of hasScriptFile()
```

Scripting initialization consists of obtaining a compilable JavaScript engine, and adding a states[] array to the engine for later use by the rules.

```
// globals
private static final int NUM_NEIGHBOURS = 26;

private ScriptEngine engine;
private Compilable compEngine;
private CompiledScript lifeScript = null;

private boolean[] states;


private void initScripting()
{
  // states array used by the scripting rules
  states = new boolean[NUM_NEIGHBOURS+1];   // includes self state
  for (int i=0; i <= NUM_NEIGHBOURS; i++)
    states[i] = false;

  // create a script engine manager
  ScriptEngineManager factory = new ScriptEngineManager();
  if (factory == null) {
    System.out.println("Could not create script engine manager");
    usingScript = false;
    return;
  }

  // create a JavaScript engine
  engine = factory.getEngineByName("js");
  if (engine == null) {
    System.out.println("Could not create javascript engine");
    usingScript = false;
    return;
  }

  // create a compilable engine
  compEngine = (Compilable) engine;
  if (compEngine == null) {
    System.out.println(
           "Could not create a compilable javascript engine");
    usingScript = false;
    return;
```

```
  }

  // add states[] array reference to engine
  engine.put("states", states);
} // end of initScripting()
```

The states[] array has 27 elements, one for each cell neighbor, and one for the cell itself.

The states[] array is added to the engine at the end of initScripting(). Since it's an object, any future changes to the data in states[] will be seen by the script when it's executed.

6.2. Changing the Grid's State

As in the previous version of CellsGrid, state change is a two-stage operation: first the next state is calculated for every cell, and then all the cells are updated. The first stage may use rules or birth and die ranges.

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

// storage for the cells making up the grid
private Cell[][][] cells;


private void stateChange()
{
  boolean willLive;

  // calculate next state for each cell

  if (!usingScript) {   // no script, so using ranges
    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);
        }
  }
  else {  // using a script
    if (isScriptModified())
      loadCompileScript(SCRIPT_FNM);   // if it's been modified

    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 = aliveScript(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 birth and die ranges use aliveNextState() to calculate each cell's new state, a method unchanged from chapter 2. Also, the nested for- loops for updating the cells are the same at the end of stateChange().

When rules are available (i.e. when usingScript is true), they're applied to each cell with the aliveScript() method. However, if the rules file has been modified since it was last accessed then it's loaded and recompiled first.

Change is detected by comparing the file's current modification timestamp with the time stored when the file was last loaded:

```
// global
private long lastModified = 0;


private boolean isScriptModified()
{
  long modTime = scriptFile.lastModified();
  if (modTime > lastModified) {
    lastModified = modTime;
    return true;
  }
  return false;
}  // end of isScriptModified()
```

The compilation uses the compilable version of the engine:

```
// globals
private Compilable compEngine;
private CompiledScript lifeScript = null;


private void loadCompileScript(String fnm)
{
  System.out.println("Loading script from " + fnm);
  lifeScript = null;

  try {
    FileReader fr = new FileReader(fnm);
    lifeScript = compEngine.compile(fr);
    fr.close();
  }
  catch(FileNotFoundException e)
  {  System.out.println("Could not find " + fnm);  }
  catch(IOException e)
  {  System.out.println("Could not read " + fnm);  }
  catch(ScriptException e)
  {  System.out.println("Problem compiling script in " + fnm);  }
  catch(NullPointerException e)
  {  System.out.println("Problem reading script in " + fnm);  }
} // end of loadCompileScript()
```

Any compilation problems will leave lifeScript with a null value.

This code has a slight chance of being affected by a threaded execution issue: the Compilable.compile() call may occur at the same time that the user saves the script via Life3D's popup menu. This might cause the compilation to be inconsistent due to changes to the file while FileReader is reading it in. Actually, that's quite unlikely bearing in mind that the file is read and compiled in a few milliseconds. For that reason, I've not bothered including any file locking code.

Once the compiled script has been generated, the file is only consulted again when stateChange() detects that the file's modification time has changed.

6.3. Executing the Script Rules

The rules script is given two inputs:

- the states[] array, which holds the states for all the cell's neighbors and the cell itself;
- numberLiving – the number of 'alive' states in states[]. (A cell can be alive or dead.)

The script calculates boolean values for beBorn and toDie, and their values are copied from the script at the end of its execution. Those booleans, and the cell's current state, are used to calculate the cell's next state.

```
private boolean aliveScript(int i, int j, int k)
{
  if (lifeScript == null)   // no script so just return current state
    return isAlive(i,j,k);

  /* collect states and number of living cells for all
     the neighbors, and the cell */
  int w = 0;
  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
        states[w] = isAlive(r,s,t);
        if (states[w])
          numberLiving++;
        w++;
      }

  // store input values in the engine
  // engine.put("states", states);    // no need to update object
  engine.put("numberLiving", numberLiving);

  // execute the script and get the beBorn and toDie results
  boolean beBorn = false;   // default values
  boolean toDie = false;
  try {
    lifeScript.eval();
    beBorn = (Boolean) engine.get("beBorn");
    toDie = (Boolean) engine.get("toDie");
  }
  catch(ScriptException e)
  {  System.out.println("Error in script execution of " +
                                                  SCRIPT_FNM);
     lifeScript  = null;  //stops this error appearing multiple times
```

```
  }


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

  // Life Rules: adjust the cell's state
  if (beBorn && !currAliveState)   // to be born && dead now
    return true;   // make alive
  else if (toDie  && currAliveState)  // to die && alive now
    return false;  // kill off
  else
    return currAliveState;  // no change
}  // end of aliveScript()
```

The isAlive() function, which returns a cell's state, is unchanged from chapter 2. It's used to fill the states[] array, and increment numberLiving.

The ordering of the nested for-loops in aliveScript() means that the ordering of the cells in states[] is as shown in Figure 12.

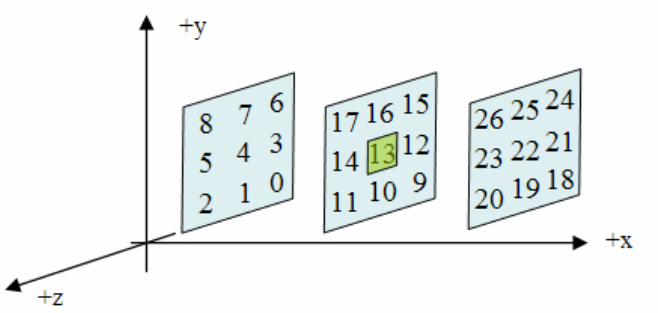

Figure 12. The Ordering of the Cells in states[].

For example, states[13] holds the state of the current cell, while states[0] is it's left-hand neighbor in the bottom y-axis row at the back.

Only numberLiving needs to be explicitly copied over to the script, since it's a simple int type. The values in the states[] array will be visible without another call to ScriptEngine.put().

beBorn and toDie are copied from the completed script, and then applied to the current state to decide its next value. If something goes wrong with the copying then both variables have default values of false.

6.4. The Rules Script

rules.js can utilize the states[] array and numberLiving integer in any way it chooses. However, the script should assign boolean values to beBorn and toDie.

A typical set of script rules:

```
var beBorn = false;
var toDie = false;
```

```
if (states[4])
  beBorn = true;

if (states[5] && states[6] && states[17] && states[14])
  toDie = true;

if (numberLiving > 5)
  toDie = true;
```

This illustrates that scripts offer the user a great deal of flexibility for changing the behavior of the application, without the user needing to know the intricacies of the implementation, or even how to program in Java.