MORE ARRAYS: THE GAME OF LIFE

CITS1001
Scope of this lecture

- The Game of Life
- Implementation
- Performance Issues

References:
- http://www.bitstorm.org/gameoflife/
- (and hundreds of other websites)
Game of Life

• Invented by John Conway over 30 years ago
• An example of a “cellular automaton”
• The game is played on a rectangular grid of “cells”
A model world

• This grid is a “model world”, where each cell is either 
  *occupied*, or *vacant*

• The initial configuration can be specified by the user, or chosen 
  randomly

We colour the 
occupied cells, and 
leave the others 
blank; the colour 
scheme is irrelevant 
to the game, but 
makes the 
application look 
more attractive
Births and Deaths

• In this model world, time passes in discrete steps known as *generations*
• The beginning of time is *generation zero*
• At each time step, the occupants of the model world live or die according to the following rules:
  • any individual with zero or one neighbours dies of *loneliness*
  • any individual with four or more neighbours dies of *overcrowding*
  • a new individual is *born* in any unoccupied cell with exactly three neighbours
• *(The neighbours of a cell are the occupants of the 8 cells surrounding that particular cell)*
Example

Initial configuration

Number of neighbours of each cell

Next generation
Interpretation

The occupants of these cells died of loneliness.

These two are new-born “babies”.

Table:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
At the edges

- The edges of the world are assumed to “wrap around” so that the map is a flat map of a torus shaped world
Program Design

• The program needs classes to perform the following operations
  • Maintaining and updating the map of the world
  • Displaying and updating screen view of the world

• We will design two classes for these two aspects, and use a SimpleCanvas to provide the display

• Therefore we will create two classes:
  • Life
  • LifeViewer
Separate the responsibilities

- The **Life** class should
  - Maintain the current map of the world
  - Update to “next generation” when requested
  - Provide access to the map for client use

- The **LifeViewer** class should
  - Create a **SimpleCanvas** to display the map
  - Display a single generation
  - Animate a client-specified number of generations
The **Life** instance variables

```java
public class Life {
    private boolean[][] map;
    private int width;
    private int height;

    // constructors & methods
}
```

We use a 2d array of type `boolean` for the map of the world; the value of the \([i][j]\) entry in the array `map` indicates whether the \((i, j)\) cell is populated or not.
The first constructor

public Life(boolean[][] initial) {
    map = initial;
    width = map.length;
    height = map[0].length;
}

This constructor allows the client to create a Life object with any initial pattern of occupied and unoccupied cells that they want.
The second constructor

```java
public Life(int width, int height, double probability) {
    map = new boolean[width][height];
    this.width = width;
    this.height = height;
    initializeMap(probability);
}
```

This allows the client to specify just the width and height and have a *random* starting configuration created
private void initializeMap(double prob) {
    for (int i=0; i<width; i++) {
        for (int j=0; j<height; j++) {
            if (Math.random() < prob) {
                map[i][j] = true;
            } else {
                map[i][j] = false;
            }
        }
    }
}

If the user specifies 0.2 as the probability, then approximately 20% of the cells will be occupied.

Math.random() is a quick way to get random doubles between 0 and 1.
The default constructor

• It is always polite to give a “default” constructor for users who just wish to get going quickly, and are willing to accept programmer chosen values

```java
public Life() {
    this(128, 128, 0.1);
}
```

Calls the 3-arg constructor with 128, 128 and 0.1 as the values
A new generation

```java
public void nextGeneration() {
    boolean[][] nextMap = new boolean[width][height];
    for (int i=0;i<width;i++) {
        for (int j=0;j<height;j++) {
            int n = numNeighbours(i,j);
            if (n <= 1 || n >= 4) {nextMap[i][j] = false;}
            if (n == 2) {nextMap[i][j] = map[i][j];}
            if (n == 3) {nextMap[i][j] = true;}
        }
    }
    map = nextMap;
}
```
This creates the *space* for the “next generation” to be stored in; we use the instance variables `width` and `height` which are always maintained at the correct values.

We cannot do the updating “in place” because the rules specify that the updating from generation to generation occurs over the entire grid *simultaneously*; we cannot change some values and then use the new values in the calculation for other cells.
Code dissection

The main loop processes each cell \((i,j)\) in turn. The number of neighbours is calculated and the \((i,j)\) cell of \texttt{nextMap} is set according to the rules of the Game of Life.

```java
int n = numNeighbours(i, j);
if (n <= 1 || n >= 4) {nextMap[i][j] = false;}
if (n == 2) {nextMap[i][j] = map[i][j];}
if (n == 3) {nextMap[i][j] = true;}
```
The final statement now makes the instance variable `map` refer to the newly completed “next generation”. Now everything is correctly updated.
Counting the neighbours

private int numNeighbours(int i, int j) {
    int n = 0;
    int ip = (i+1) % width;
    int im = (width + i - 1) % width;
    int jp = (j+1) % height;
    int jm = (height + j - 1) % height;

    if (map[im][jm]) n++;
    if (map[im][j]) n++;
    if (map[im][jp]) n++;
    if (map[i][jm]) n++;
    if (map[i][jp]) n++;
    if (map[ip][jm]) n++;
    if (map[ip][j]) n++;
    if (map[ip][jp]) n++;
    return n;
}
Code Dissection

Here, $ip$, $im$, $jp$ and $jm$ refer to $i+1$, $i-1$, $j+1$ and $j-1$ respectively.
Dealing with wrap-around

When \( i \) is equal to \( width-1 \), then \( i+1 \) is equal to \( width \), which is off the right-hand side of the picture.

Calculating \( (i+1) \% width \) has the effect of “wrapping around” back to 0 which is what is needed.

Similarly, when \( i=0 \), then \( (width+i-1) \% width \) “wraps around” back to \( width-1 \).
More dissection

```java
if (map[im][jm]) n++;  
if (map[im][j]) n++;  
if (map[im][jp]) n++;  
if (map[i][jm]) n++;  
if (map[i][jp]) n++;  
if (map[ip][jm]) n++;  
if (map[ip][j]) n++;  
if (map[ip][jp]) n++;  
return n;
```

This is simply a sequence of 8 `if` statements that just add 1 to the variable `n` if the corresponding cell is occupied.

Finally the value `n` is returned.

Notice that the method is `private`, and only to be used by the `Life` object itself.
One performance issue

- As written, this code consumes a large amount of memory
What happens during the update?

- After the statement `map = nextMap`
What happens during the update?

- When the method finishes.. This area of memory is now “garbage”
Garbage Collection

- Java will automatically “collect” the garbage for you and recycle it, but there is a performance penalty associated with this.
- If the world is large and you are simulating many generations then the memory will very rapidly fill up and the garbage collector will keep interrupting the smooth “animation”.
- To fix this, it is better to keep two “maps of the world” as instance variables and then just swap them over at each generation!

```java
private boolean[][] map;
private boolean[][] nextMap;
```
In the `nextGeneration()` method

- Delete the code
  ```java
  boolean[][] nextMap = new boolean[width][height];
  ```
  because we do not want to create a new array every generation, but instead re-use the instance variable `nextMap`

- Replace
  ```java
  map = nextMap;
  ```
  with
  ```java
  boolean[][] swap = map;
  map = nextMap;
  nextMap = swap;
  ```
How does this work?
Program Design

• Recall that our program design called for three classes
  • Life
  • LifeViewer
  • SimpleCanvas

• The internal structure of the Life class has been written

• Now we must decide
  • How the LifeViewer interacts with the Life object
  • The internal structure of the LifeViewer object
Interaction

- The **LifeViewer** needs the **Life** object to do the following:
  - Send the data for the current generation to be displayed
  - Update to the next generation when required
- This means that the **Life** class must have the following methods

```java
public boolean[][] getMap()
```
- When called, this will return the array representing the current map

```java
public void nextGeneration()
```
- When called, this will update the **Life** object to the next generation
The extra method for Life

```java
public boolean[][] getMap() {
    return map;
}
```

Very simple method; simply returns a reference to the array representing the current generation.
Variables defined in `LifeViewer`

```java
public class LifeViewer {
    private Life life;
    private int width;
    private int height;
    private SimpleCanvas c;

    private final static int CELL_SIZE = 4;

    private final static java.awt.Color BACK_COLOUR = java.awt.Color.white;
    private final static java.awt.Color CELL_COLOUR = java.awt.Color.red;
    private final static java.awt.Color GRID_COLOUR = java.awt.Color.black;
```
public class LifeViewer {
    private Life life;
    private int width;
    private int height;
    private SimpleCanvas c;

    Each LifeViewer is responsible for displaying one Life object.
    The width and height are stored as instance variables for convenience.
    Each LifeViewer has one SimpleCanvas on which to draw.
Here we define useful constants for the size of the cells and the colours to be used for the drawing. They are declared **final** because they will not change during the runtime of the program.
The constructor for LifeViewer

```java
public LifeViewer(Life life) {

    this.life = life;
    width = life.getMap().length;
    height = life.getMap()[0].length;

    c = new SimpleCanvas("Life", width*CELL_SIZE+1,
                                        height*CELL_SIZE+1, false);

    display();
}
```
What the constructor does

```java
public LifeViewer(Life life) {
    this.life = life;
    width = life.getMap().length;
    height = life.getMap()[0].length;
}
```

The constructor’s job is to initialize all the instance variables; the argument is an instance of the class `Life`, and so this must be saved in the corresponding instance variable. The `width` and `height` variables are then determined by asking the `life` object for a reference to its 2-d `boolean` array, and finding its dimensions.
The constructor for `LifeViewer`

c = new SimpleCanvas("Life", width*CELL_SIZE+1,
                          height*CELL_SIZE+1, false);

We need the "+1" to give the right hand edge and the bottom edge a nice border
Auto-repaint

- We are using the 4-arg constructor for SimpleCanvas where the fourth argument is a boolean indicating whether the user wants to use automatic repainting or manual repainting.
- Behind the scenes, every drawing command has two effects:
  - The new pixel values are stored by the SimpleCanvas.
  - The new pixel values are displayed on the computer screen.
- The first operation is very fast, because it just involves the computer’s memory, while the second is much slower.
- When automatic repainting is turned off, drawing commands just affect the memory and the screen is only changed when the client explicitly requests a repaint.
  - Can do lots of drawing commands, followed by one single repaint.
Displaying the **Life** object

- Now we need a method to display the **Life** object
- This is basically three steps
  - getting the current map from the **Life** object
  - drawing the appropriate picture on the drawing area, which will involve painting every cell of the grid the right colour
  - calling `repaint()` to have the drawing rendered to the screen
A method to display the Life object

```java
private void display() {
    erase();
    drawCells();
    drawGrid();
    c.repaint();
}
```

Notice how the displaying functions – erasing the old picture, drawing the cells and drawing the grid are all migrated to separate methods; this makes for code that is very easy to read and to update.
Erasing the old picture

```java
private void erase() {
    c.setForegroundColour(BACK_COLOUR);
    for (int i=0; i<width; i++) {
        c.drawLine(i,0,i,height);
    }
}
```

Familiar code to simply overwrite the entire screen with a collection of vertical lines in the background colour
Drawing the cells

```java
private void drawCells() {
    boolean[][] map = life.getMap();
    c.setForegroundColour(CELL_COLOUR);
    for (int i=0;i<width;i++) {
        for (int j=0;j<height;j++) {
            if (map[i][j]) {
                for (int k=0;k<CELL_SIZE;k++){
                    c.drawLine(i*CELL_SIZE+k,j*CELL_SIZE,
                                i*CELL_SIZE+k,(j+1)*CELL_SIZE);
                }
            }
        }
    }
}
```

First we ask the Life object to tell us what the current world map is
Drawing the cells

```java
for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
        if (map[i][j]) {
            for (int k=0; k<CELL_SIZE; k++)
                c.drawLine(i*CELL_SIZE+k, j*CELL_SIZE,
                            i*CELL_SIZE+k, (j+1)*CELL_SIZE);
        }
    }
}
```

Go through the 2-d array; if the value is `true` then draw a filled-in square to represent a populated cell.
private void drawGrid() {
    c.setForegroundColour(GRID_COLOUR);

    for (int i=0; i<=width; i++) {
        c.drawLine(i*CELL_SIZE,0,i*CELL_SIZE,height*CELL_SIZE);
    }

    for (int j=0; j<=height; j++) {
        c.drawLine(0,j*CELL_SIZE,width*CELL_SIZE,j*CELL_SIZE);
    }
}
A method to display the Life object

private void display() {
    erase();
    drawCells();
    drawGrid();
    c.repaint();
}

Finally, we repaint the SimpleCanvas with the new generation drawn on it in order to make all the changes visible on the screen.
public void animate(int n) {
    for (int i=0; i<n; i++) {
        life.nextGeneration();
        display();
        c.wait(250);
    }
}

The only public method is the one that allows the user to specify that they want to see n generations of “The Game of Life” one after the other; it is a simple loop that asks the Life object to calculate the next generation, then displays it, and then waits briefly to give an “animated” effect.
Create the Life object
It appears on the workbench
Now create a LifeViewer to view it
The initial randomly chosen configuration is displayed on a SimpleCanvas.
After 100 generations, the map looks a bit more “organized”
A new colour scheme can be obtained just by changing the three constants.
Extensions

• There are hundreds of research papers, websites and recreational articles about Conway’s Game of Life
• Life is just one example of an entire class of systems known as cellular automata, many of which have been heavily studied
• There are many variants, but we will just consider what happens if we change the rules determining birth and death
• Conway’s rules are sometimes described as the String B3/S23
  • a new individual is born in a cell with 3 neighbours
  • an individual survives in a cell with 2 or 3 neighbours
Extending the code

• We could simply alter the code to accommodate any *particular* set of rules, but a more object-oriented approach would be to devise a class so that *any* such rule set could be “plugged in” by the client.

• This maximizes the re-use of the existing code, and minimizes the temptation to “cut-n-paste” every time the user wants to experiment with a new rule set.

• However, this does require re-engineering the *Life* class.
The CARule interface

The CARule class has the following interface - that is, the following list of methods

The implementation of the methods isn’t yet specified

```java
public class CARule {
    public CARule(String b, String s);
    public boolean birth(int value);
    public boolean survival(int value);
}
```

The role of a CARule object is simply to respond as to whether a particular value is a “birth” value or not, or a “survival” value or not
The new **Life** class

- The **Life** class will now maintain a **CARule** object as an instance variable `car`, and refer to it whenever it wants to decide whether a particular value should cause an individual to be born or survive.

```java
for (int i=0; i<width; i++) {
    for (int j=0; j<height; j++) {
        int n = numNeighbours(i, j);
        nextMap[i][j] = false;
        if (map[i][j] && car.survival(n)
            || !map[i][j] && car.birth(n))
            nextMap[i][j] = true;
    }
}
```