CT255 - NEXT GEN TECHNOLOGIES II

2D GAMES PROGRAMMING IN JAVA

Andrew Hayes

Student ID: 21321503

2BCT

University of Galway

February 2022

Contents

1 Lecture 1

1.1 2D Co-Ordinate System

The **JFrame** class will provide a Window with associated graphics canvas & a pixel-based co-ordinate system with the origin at (0,0) on the top (left). A JFrame is a top-level window with a title & a border. To have access to JFrame and associated methods:
`import java.awt.*;`
`import javax.swing.*;`

The top 50 pixels or so are hidden by the window's title bar (depending on the operating system).

1.2 Basic Graphics in Java

2D graphics can be drawn using the `Graphics` class. This provides methods for drawing "primitives" (lines, circles, boxes) and also images (.jpg, .png, etc.).

The `paint()` method of the `JFrame` class is automatically invoked whenever it needs to be painted (system-invoked). Otherwise, you can force painting to happen via `repaint()` if you need to repaint when the OS doesn't think that it's needed.

```java
    public void paint (Graphics g) {
        // use the "g" object to draw graphics
    }
```

1.3 Drawing Text using Methods of the Graphics Class

```java
    public void paint (Graphics g) {
        Font f = new Font("Times", Font.PLAIN, 24);
        g.setFont(f);
        Color c = Color.BLACK;
        g.setColor(c);
        g.drawString("Pacman!", 20, 60);
    }
```

.
This should be added as a method of the application class.

2 Lecture 2

2.1 Animation with Threads

Animation is the changing of grahics over time, e.g., moving a spaceship across the screen, changing its position by one pixel every 0.02 seconds.

One of the best ways to do periodic execution of code is to use **threads**. Threads allow multiple taks to run independently/concurrently within a program. Essentially, this means that we spawn a separate execution "branch" that operates independently of our program's main flow of control. For example, the new thread could repeatedly sleep for 20ms, then carry out an animation, and call `this.repaint()` on the application.

2.1.1 Implementing Threads in Java

Your application class should implement the `Runnable` interface, i.e.:

```java
public class MyApplication extends JFrame implements Runnable {
}
```

Your application now **must** provide an implementation for the `run()` method, which is executed when a thread is started, serving as its "main" function, i.e.:

```java
public void run(){
}
```

To create & start a new thread running from your application class:

```java
Thread t = new Thread(this);
t.start();
```

The typical actions of an animation thread are as follows:

1. Sleep for (say) 20ms using `Thread.sleep(20);`. Note that you will be **required** to handle `InterruptedException`.
2. Carry out movement of game objects.
3. Call `this.repaint();` which (indirectly) invokes our `paint(Graphics g)` method.
4. Go back to Step 1.

2.1.2 Threads Test

```java
import java.awt.*;
import javax.swing.*;

public class ThreadsTest implements Runnable {
    public ThreadsTest() {
        Thread t = new Thread(this);
        t.start();
    }

    public void run() {
        System.out.println("Thread started");

        for (int i = 0; i < 15; i++) {
            System.out.println("Loop " + i + "Start");
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            System.out.println("Loop " + i + " end");
        }
        System.out.println("Thread ended");
    }

    public static void main(String[] args) {
        ThreadsTest tt = new ThreadsTest();
    }
}
```

2.2 Game Object Classes

Games typically have **game object classes** (spaceships, alines, cars, bullets, etc.). Numerous instances of each may exist at runtime. This class encapsulates the data (position, colour, etc.) and code (move, draw, die, etc.) associated with the game object.

Typically, we store these instances in a data structure such as an array, so that during our animation & painting phases we can iterated through them all and invoke the `animate()` & `paint()` methods on each instance.

3 Lecture 03

3.1 Handling Keyboard Input

In GUI-based languages such as Java (with AWT), the mouse & keyboard are handled as "**events**". These events may happen at any time. Events are queued as they happen and are dealt with at the next free, idle time. AWT handles events coming from the OS by dispatching them to *listeners* registered to those events.

To handle keyboard input:

1. Make a class that implements `KeyListener`. Make sure that you have an instance of this class.
2. Add this instance as a key listener attached to the `JFrame` that receives the messages from the OS. The simplest way is to make your JFrame-derived class handle the events it receives itself.

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MyApplication extends JFrame implements KeyListener {
    public MyApplication() {
        // send keyboard events arriving into this JFrame to its own event
    handlers.
        addKeyListener(this);
    }

    // 3 keyboard event handler functions
    public void keyPressed(KeyEvent e) {
    }
    public void keyReleased(Event e) {
    }
    public void keyTyped(KeyEvent e) {
    }
}
```

Notes:

- The `KeyEvent` parameter `e` provides the "virtual keycode" of the key that has triggered the event, and constants are defined to match these values, e.g., `KeyEvent.VK_Q` or `KeyEvent.VK_ENTER`.
- To get the keycode, use `e.getKeyCode()`.
- For our game applications, our application class will implement both `KeyListener` & `Runnable`.
- Note the extra import: `import java.awt.event.*;`.

3.2 Loading & Displaying Raster Images

The constructor of the `ImageIcon` class (defined in `javax.swing` loads an image from disk (`.jpg`. `.gif`, or `.png`) and returns it as a new instance of the `ImageIcon` class.

The `getImage()` method of this `ImageIcon` object gives you a useable `Image` class object, which can be displayed in your `paint()` method by the `Graphics` class.

```java
import java.awt.*;
import javax.swing.*;

public class DisplayRasterImage extends JFrame {
    // member data
    private static String workingDirectory;
    private Image alienImage;

    // constructor
    public DisplayRasterImage() {
        // set ip JFrame
        setBounds(100, 100, 300, 300);
        setVisible(true);

        // load image from disk. Make sure you have the path right!
        ImageIcon icon = new ImageIcon(workingDirectory + "/alien_ship_1.png");
        alienImage = icon.getImage();

        repaint();
    }

    // application's paint method (may first happen *before* image is finished
    loading, hence repaint() above
    public void paint(Graphics g) {
        // draw a black rectangle on the whole canvas
        g.setColor(Color.BLACK);
        g.fillRect(0,0,300,300);

        // display the image (final argument is an "ImageObserver" object)
        g.drawImage(alienImage, 150, 150, null);
    }

    // application entry point
    public static void main(String[] args) {
        workingDirectory = System.getProperty("user.dir");
        System.out.println("workingDirectory = " + workingDirectory);
        DisplayRasterImage d = new DisplayRasterImage();

    }
}
```

4 Lecture 04 - Screen Flicker

Screen flicker is caused by software redrawing a screen out of sync with the screen being refreshed by the graphics hardware, resulting in a half-drawn image occasionally being displayed.

The solution to screen flicker is to use **double buffering**. Double buffering involves first rendering all graphics to an offscreen *memory buffer*. Then, when finished drawing a frame of animation, flip the offscreen buffer onscreen furing the "vertical sync" period.

`java.awt` provides a `BufferStrategy` class that applies the best approach based off the capabilities of the computer on which the software is running.

4.1 Implementing Double Buffering

1. `import java.awt.image.*;`
2. Add a new member variable to the Application class: `private BufferStrategy strategy;`
3. In the Application class's constructor method:

   ```java
   createBufferStrategy(2);
   strategy = getBufferStrategy();
   ```

   Note that this code should be executed **after** the JFrame has been displayed, i.e. after `setBounds()` & `setVisible()`.
4. At the start of the `paint(Graphics g)` method, include `g = strategy.getDrawGraphics();` to redirect our drawing calls to the offscreen buffer.
5. At the end of the `paint(Graphics g)` method, include `strategy.show();` to indicate that we want to flip the buffers.

5 Lecture 05 - Finishing Space Invaders

5.1 Animated 2D Sprites

To animate a 2D sprite, we can simply load two or more images and alternate or cycle between them. For our game, switching image once per second (i.e., every 50th redraw is about right).

E.g., use this in a modified `Sprite2D` class.

```java
public void paint(Graphics g) {
    framesDrawn++;
    if (framesDrawn % 100 < 50) {
        g.drawImage(myImage, (int) x, (int) y, null);
    }
    else {
        g.drawImage(myImage2, (int) x, (int) y, null);
    }
}
```

5.2 Collision Detection

To detect collisions, we cna simply check for overlapping rectangles.

```java
if ( ((x1<x2 && x1+w1>x2) ||(x2<x1 && x2+w2>x1) ) && ( (y1<y2 && y1+h1>y2 ) || (
    y2<y1 && y2 + h2>y1) ) )
```

5.3 Game States

Games normally have at least two high-level "states", i.e., is the game in progress or are we currently displaying a menu before the game starts (or after it finishes)? To do this, we can simply add a boolean member to the application class: `isGameInProgress`. Depending on the value of this, we can handle various things differently:

- Keypresses.
- The `paint()` method.
- The thread's game loop.

6 Lecture 06 - Conway's Game of Life

6.1 Mouse Events

Mouse events notify when the user uses the mouse (or similar input device) to interact with a component. Mouse events occur when the pointer enters or exits a component's onscreen area and when the user presses or releases one of the mouse buttons.

Additional events such as mouse movement & the mouse wheel can be handled by implementing the `MouseMotionListener` and `MouseWheelListener` interfaces.

1. Have your class implement `MouseListener`.
2. Add `addMouseListener(this);` to the clas constructor.
3. Implement the methods below:

```java
// mouse events which must be implemented for MouseListener
public void  mousePressed(MouseEvent e) { }
public void mouseReleased(MouseEvent e) { }
public void  mouseEntered(MouseEvent e) { }
public void   mouseExited(MouseEvent e) { }
public void  mouseClicked(MouseEvent e) { }
```

6.2 Conway's Game of Life: Rules

`private boolean gameState[][][] new boolean[40][40][2];`

1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
2. Any live cell with two or three live neighbours lives on to the next generation.
3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

Each generation (iteration) of the game is created by applying the above rules simultaneously to every cell in its preceding generation: births & deaths occur simultaneously. To implement this properly, we will need to have two separate game states in memory:

- One is the "front buffer" that we are currently displaying, and which we are checking the above rules on.
- The other is the "back buffer" that we're applying the results of the rules to.
- The "back buffer" will be switched to the "front" after applying the rules to every cell.

We will check the eight neighbours of each cell as follows:

```java
for (int x=0;x<40;x++) {
    for (int y=0;y<40;y++) {
        // count the live neighbours of cell [x][y][0]
        for (int xx=-1;xx<=1;xx++) {
            for (int yy=-1;yy<=1;yy++) {
                if (xx!=0 || yy!=0) {
                    // check cell [x+xx][y+yy][0]
                    // but.. what if x+xx==-1, etc. ?
                }
            }
        }
    }
}
```

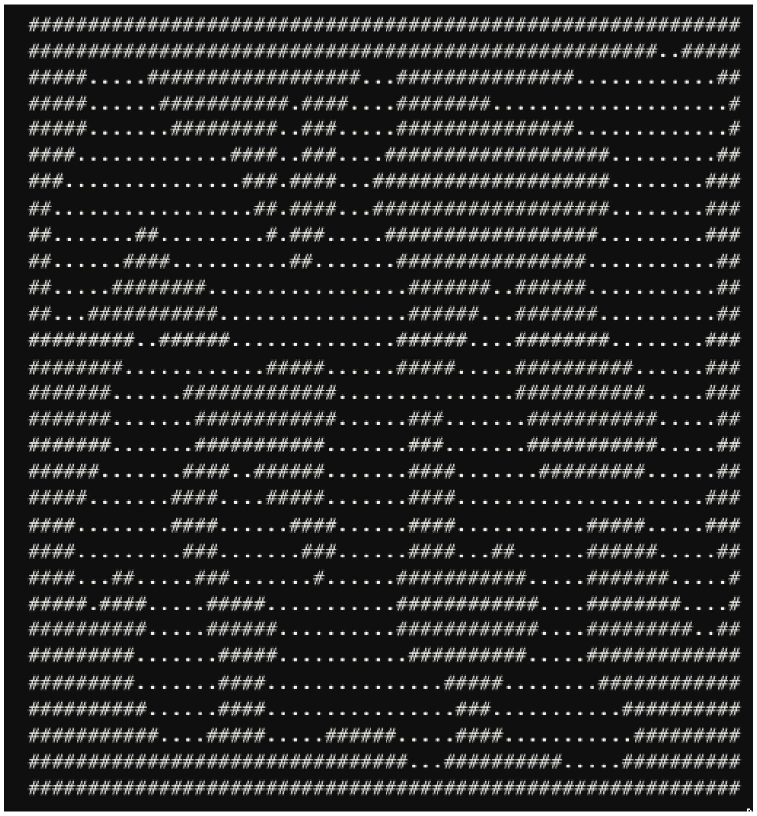
Note that we need to define the neighbours for cells at the edges of the map. The usual procedure is to "wrap around" to the opposite side.

6.3 Another Example of a Cellular Automata Algorithm in Use

The image below is of an algorithmically-generated cave-like structure, for use in a 2D computer game. Each of the cells, laid out in a 60x30 grid, either has a wall (denoted by #) or a floor (denoted by .).

The cellular automata algorithm which generated this output uses the following steps:

- For each cell, randomly define it as: wall (60% chance) or floor (40% chance).
- Perform the following procedure four times: Calculate the number of wall neighbours for each cell, and define each cell which has at least 5 neighbouring wall cells, as a wall cell itself. Otherwise (i.e., if it has less than 5 wall neighbours), define it as a floor cell.



```
############################################################
#####################################################..#####
#####.....##################...###############............##
#####......###########.####....########....................#
#####.......#########..###.....###############.............#
####.............####..###....###################.........##
###...............###.####...####################........###
##.................##.####...####################........###
##.......##.........#.###.....##################.........###
##......####..........##.......#################..........##
##.....########.................#######..######...........##
##...###########................######...#######..........##
#########..######..............######....########........###
########............#####......#####.....##########......###
#######......#############...............###########.....###
#######.......############......###.......###########.....##
#######.......###########.......###.......###########.....##
######.......####..######.......####.......#########......##
#####.......####....#####.......####.....................###
####........####......####......####...........#####.....###
####.........###.......###......####...##......######.....##
####...##.....###.......#......###########.....#######.....#
#####.####.....#####...........############....########....#
##########.....######..........############....#########..##
#########.......#####...........##########.....#############
#########.......####...............#####........############
##########......####................###...........##########
###########....#####.....######.....####...........#########
################################...##########.....##########
############################################################
```

7 Week 8

7.1 Reading from Text Files

The `java.io` provides file handling classes:

- `FileReader` to read from a text file.
- `BufferedFileReader` to read from a text file more efficiently (reads larger blocks & buffers/caches them).
  - *Exception handling is required.*
  - Uses the `FileReader` class constructor to open a file.
  - Uses the `readLine()` method to read a line of text (which returns a `String`).
  - Uses the `close()` method to close the file.

Sample code that reads just one line from the file (stopping at the end of the file or when a carriage return is encountered):

```java
String filename = "C:\\Users\\Sam\\Desktop\\lifegame.txt";
String textinput = null;
try {
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    textinput = reader.readLine();
    reader.close();
}
catch (IOException e) { }
```

Sample code that reads all the carriage return-separated lines from a file:

```java
String line=null;
String filename = "C:\\Users\\Sam\\Desktop\\lifegame.txt";
try {
    BufferedReader reader = new BufferedReader(new FileReader(filename));
    do {
        try {
            line = reader.readLine();
            // do something with String here!
        } catch (IOException e) { }
    }
    while (line != null);

    reader.close();

} catch (IOException e) { }
```

7.2 Writing to Text Files

Use the `FileWriter` & `BufferedWriter` classes:

`BufferedWriter`:

1. Use the `FileWriter` class constructor to open a file.
2. Use the `write(String s)` method to write a line to the file (with a CR appended automatically).
3. Use the `close()` method to close the file.

E.g., to write a single String to a file:

```java
String filename = "C:\\Users\\Sam\\Desktop\\lifegame.txt";
try {
    BufferedWriter writer = new BufferedWriter(new FileWriter(filename));
    writer.write(outputtext);
    writer.close();
}
catch (IOException e) { }
```

7.3 Handling Mouse Motion Events

In addition to mouse button events, we can also receive mouse *movement* events. Have the class implement the `MouseMotionListener` interface as well as `MouseListener`. In the application class constructor have: `addMouseMotionListener(this);`. Add these methods (receives the same data as the mouse events we have already seeen):

```java
public void mouseMoved(MouseEvent e)
public void mouseDragged(MouseEvent e)
```

This is useful for making it less tedious to create a new initial game set-up.

8 A* Pathfinding

The fundamental operations of the **A*** algorithm is to traverse a map by exploring promising positions (nodes) beginning at a starting location, witht the goal of finding the best route to a target location.

Each node has four attributes, other than its position on the map:

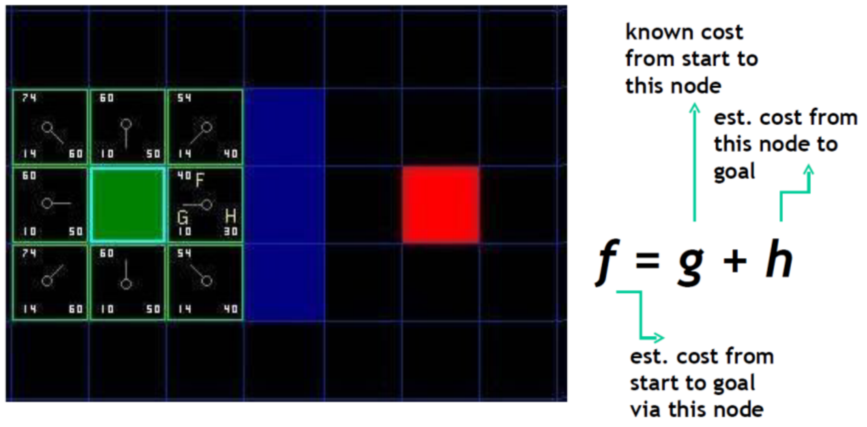
- g is the cost of getting from the starting node to this node.
- h is the heuristic (estimated) cost of getting from this node to the target node. h is a best guess, since the algorithm does not yet know the actual cost.

- f is the sum of g & h, and is the algorithm;s best current estimate as to the total cost of travelling from the starting location to the target location via this node.
- *parent* is the identity of the node which connected to this node along a potential solution path.

The algorithm maintains two lists of nodes: the **open** list & the **closed** list.

- The open list consists of nodes to which the algorithm has already found a route, i.e. one of its connected neighbours has been evaluated (*expanded*) but which have not yet themselves been expanded.
- The closed list consists of nodes that have been expanded and which therefore should not be re-visited.

Progress is made by identifying the most promising node in the open list, i.e., the one with the lowest f value, and expanging it by adding each of its connected neighbours to the open list, unless they are already closed. As nodes are expanded, they are moved to the closed list. As nodes are added to the open list, their f, g, h, & *parent* values are recorded. The g value of a node is, of course, equal to the g value of its parent plus the cost of moving from the parent to the node itself.


known cost from start to this node
est. cost from this node to goal
f = g + h
est. cost from start to goal via this node

8.1 Implementing A* Pathfinding