CT255 NGT2
Digital Media

[2D Games in Java]

Dr Sam Redfern
sam.redfern@nuigalway.ie
@psychicsoftware

2D Games Programming in Java

- As a direct support for CT2106 and CT2109
- Problem-Based approach
- 'Just in Time' focus
- Emphasis on researching features of Java yourselves (under my direction)
- Various useful topics related to input/output, graphical display, realtime update, etc.
- Even some Artificial Intelligence! (A.I.)

Labs & Lectures etc.

- Lecture/Discussion:
  - Mondays 12-1pm, AC213
- Workshop/Practical Work:
  - Mondays 2-4pm, Lab IT101 (CS Building)
  - Some weeks we'll do less than an hour in the lecture/discussion, plus 2 hrs in the lab
- Course notes:
  - Blackboard

Discord

- I have created a Discord server for this module
- Please use this to make comments, ask questions, share resources etc.
- Using this approach, I can generally answer questions fairly quickly anytime during the semester, not just during class/lab times

- Use this Discord link to join the server:
  - https://discord.gg/uWem2rQg7a

Grading

- Weekly assignments will account for 25% of my part of the CT255 course (M. Schukat did the other half of CT255 in sem. 1)
- Assignments are always due at the start of the subsequent lecture (where you will be given a sample solution and we'll spend some time discussing it)
- There will also be questions in the Summer exam paper, accounting for 75% of my part of the CT255 course

Developing Games in Java

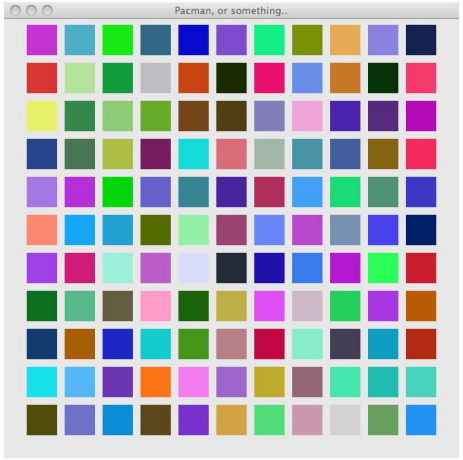
David Brackeen

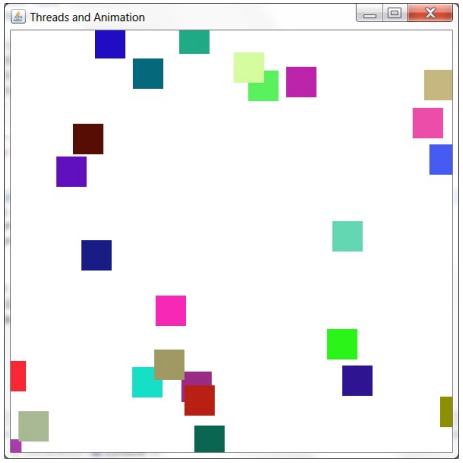
New Riders

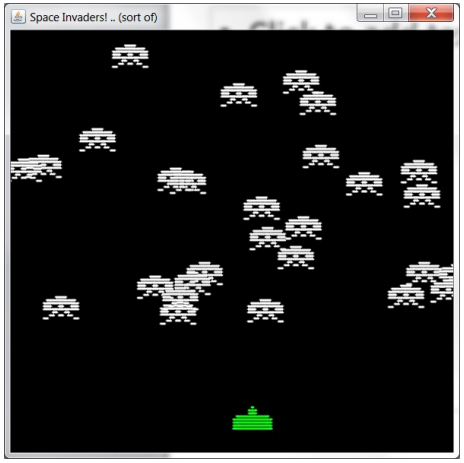
NRG

Topics will include

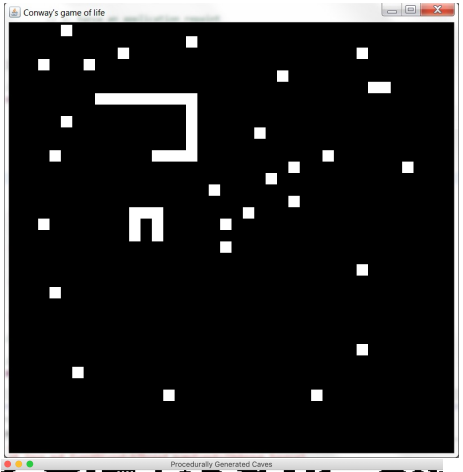
- Graphics with the JFrame class
- Raster & vector graphics methods of the Graphics class
- Graphics contexts, double buffering
- 2D ‘sprite’ animation
- Constructing game object classes
- Collision detection
- Multi-threading
- Keyboard and mouse input
- Game states using 2D arrays, hash tables and other collection classes
- Mazes and A.I. pathfinding using the A* algorithm


Pacman, or something..

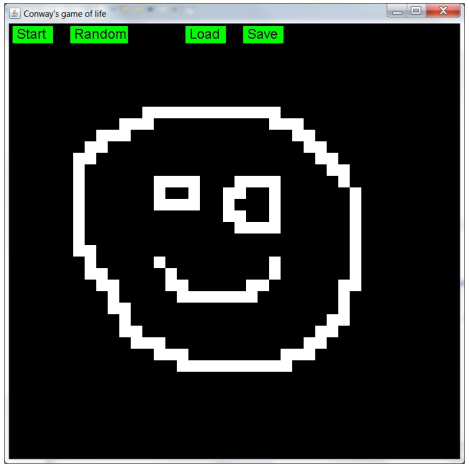

Threads and Animation

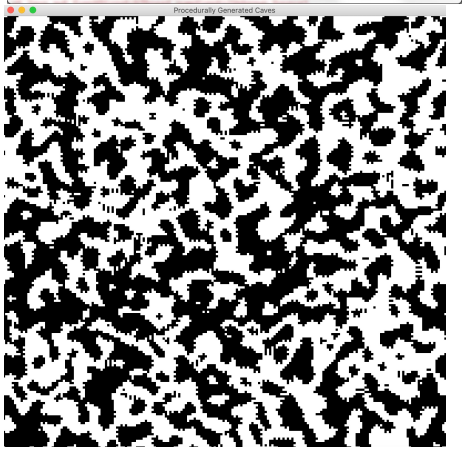

Space Invaders! .. (sort of)

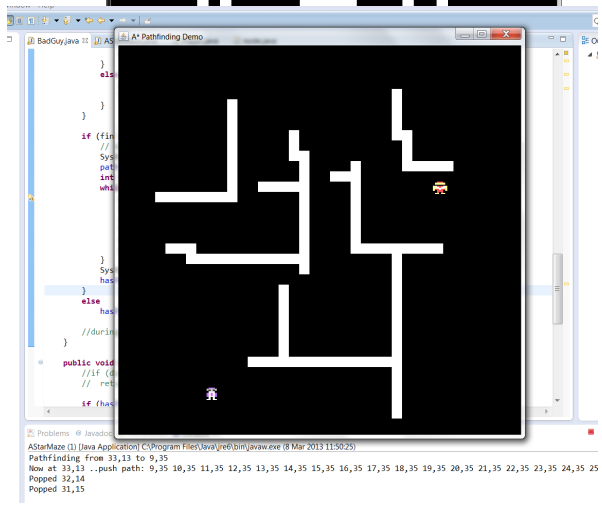

Space Invaders!
Score: 120 Best: 1040


Conway's game of life


Conway's game of life
Start
Random


Conway's game of life
Start
Random
Load
Save


Procedurally Generated Caves

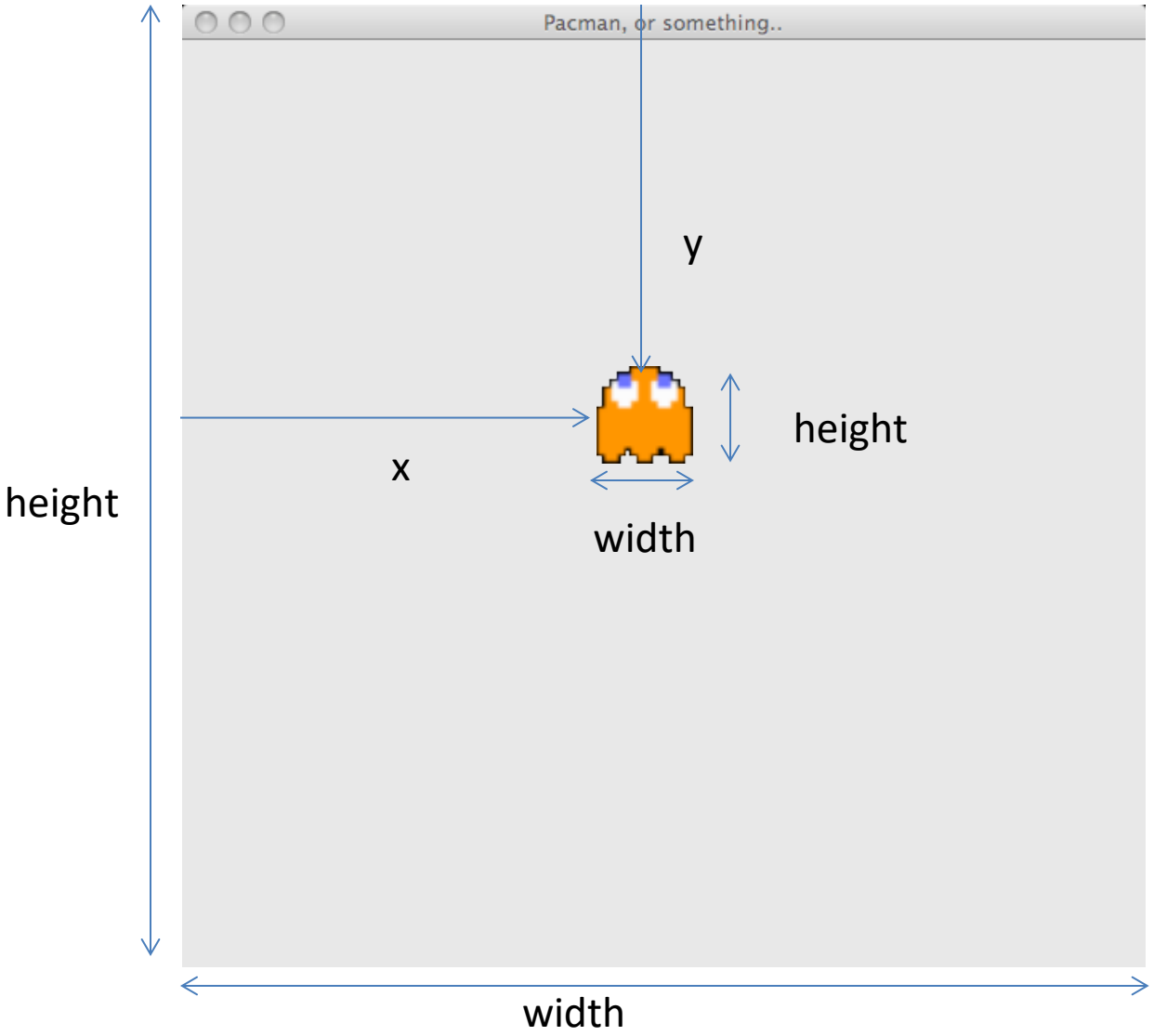

A* Pathfinding Demo
AStarMaze (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (8 Mar 2013 11:50:25)
Pathfinding from 33,13 to 9,35
Now at 33,13 ..push path: 9,35 10,35 11,35 12,35 13,35 14,35 15,35 16,35 17,35 18,35 19,35 20,35 21,35 22,35 23,35 24,35 25
Popped 32,14
Popped 31,15

2D Co-ordinate System

The JFrame class will provide a Window with associated graphics canvas, and a pixel-based coordinate system

Origin (0,0) at top-left

Top 50 pixels or so are hidden by the window's title bar (depends on Operating System)



Creating a Window-based Application

- Create a new Java project in Eclipse (or other IDE)
- Right-click the project
- New > class
- Name your class, e.g. ‘MyApplication’

- In Java, you need to have a method named **main** in at least one class.

- A JFrame is a top-level window with a title and a border. To have access to JFrame and associated methods:

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

A Minimal JFrame-based app.

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

public class MyApplication extends JFrame {

    private static final Dimension WindowSize = new Dimension(600,600);

    public MyApplication() {
        //Create and set up the window.
        this.setTitle("Pacman, or something..");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        //Display the window, centred on the screen
        Dimension screensize = java.awt.Toolkit.getDefaultToolkit().getScreenSize();
        int x = screensize.width/2 - WindowSize.width/2;
        int y = screensize.height/2 - WindowSize.height/2;
        setBounds(x, y, WindowSize.width, WindowSize.height);
        setVisible(true);
    }


    public static void main(String [ ] args) {
      MyApplication w = new MyApplication();
    }

}
```

A note about Eclipse:

- It will attempt to suggest code fixes such as creating new methods for you
- Be careful! A method call from your base class that can't be compiled often means you have mistyped or forgotten an import, or forgotten to extend from a base class
- So don't let it trick you into creating an empty method in its place!

Basic graphics in Java

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

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

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

Drawing text using methods of the Graphics class

setColor(Color c)

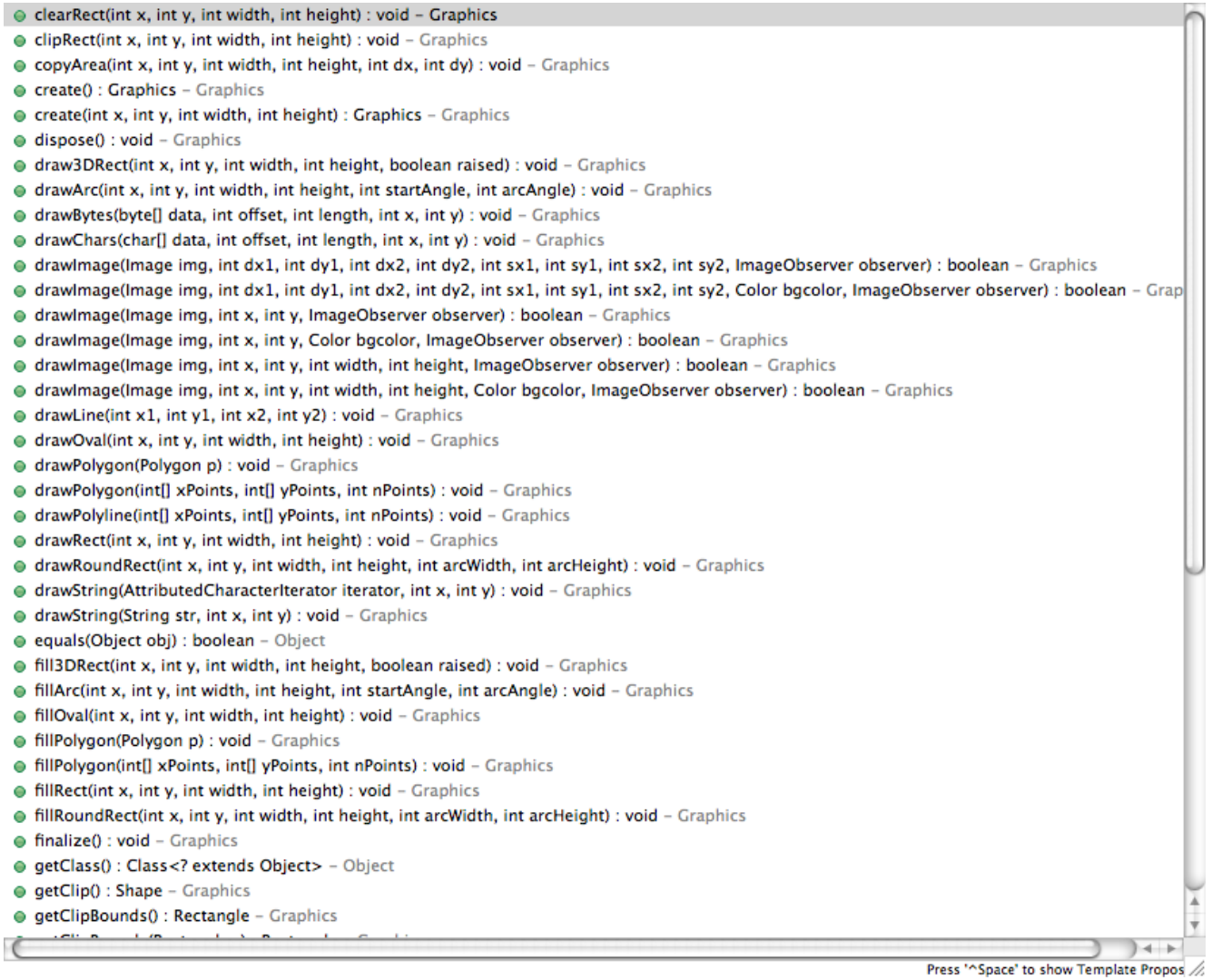
setFont(Font font)

drawString(String str, int x, int y)

```
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 MyApplication class
- Note the usefulness of the context-help in Eclipse – tells you method names, their parameters and a description of them

The graphics class has lots of useful methods!

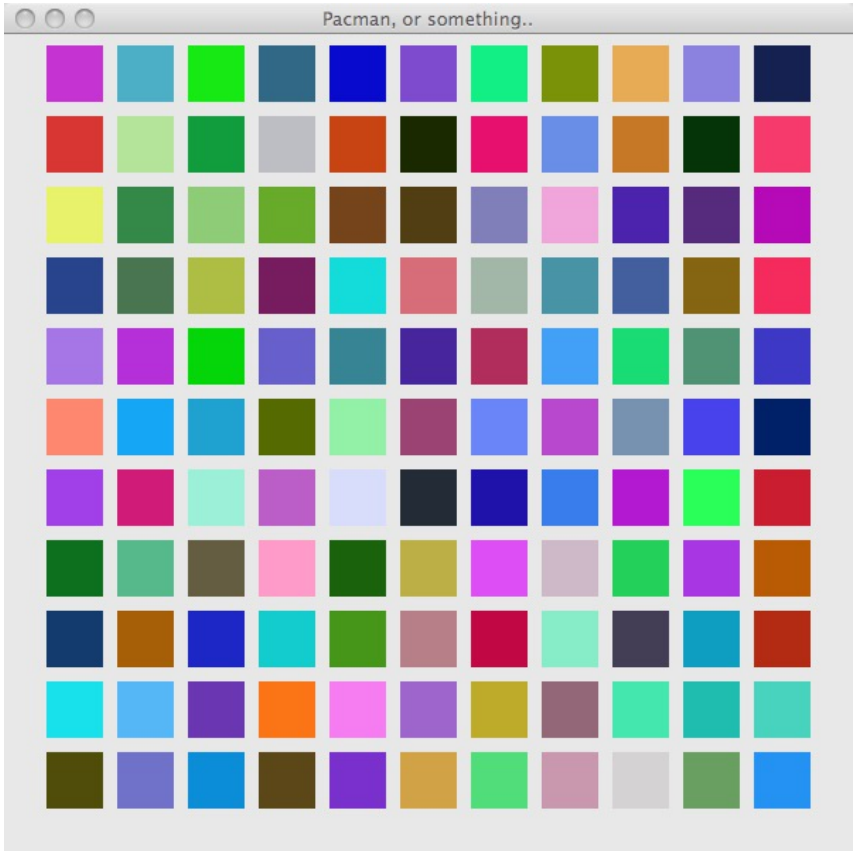


Week 1 Assignment

- Write a JFrame-based program that fills its window with randomly coloured squares

Hints:

- Use nested loops to produce the squares
- Think about your co-ordinate system (x/y).. what are the min/max values of these you want to stay between?
- Investigate the `fillRect()` method of the Graphics class
- Investigate how to specify an arbitrary Color rather than using a stock Color



To get a random integer between 0 and 255: **int red = (int)(Math.*random()*256);**

Your assignment code should be submitted via Blackboard.

CT255 NGT2
Digital Media/ 2D Games

Week 2

[2D Games in Java]

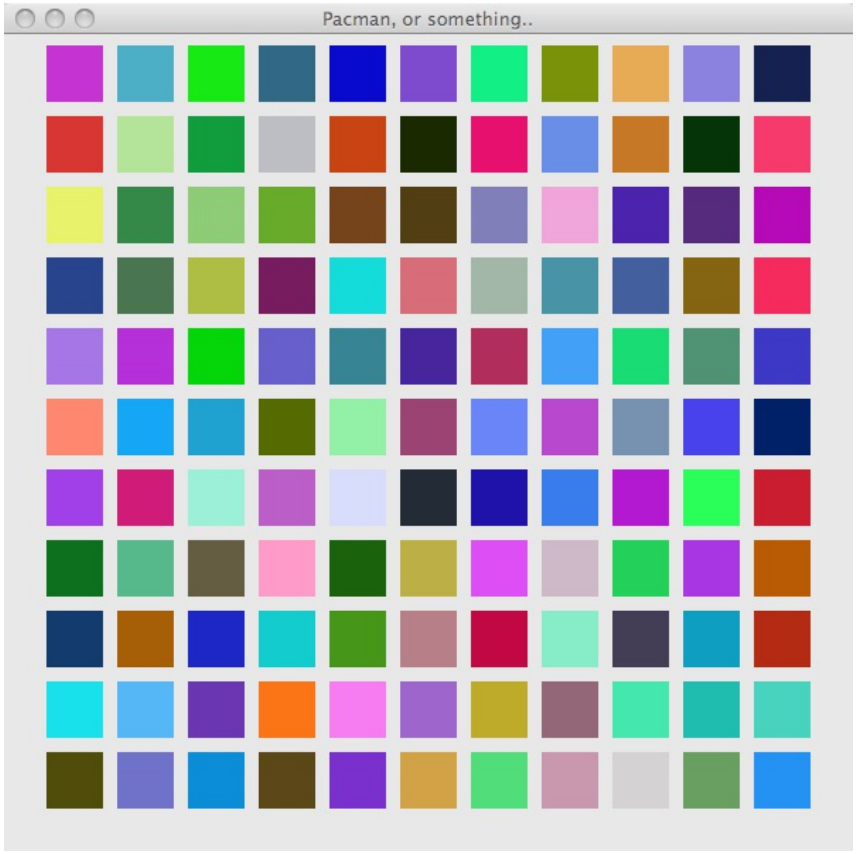
sam.redfern@universityofgalway.ie
@psychicsoftware

https://discord.gg/uWem2rQg7a



Week 1 Assignment

- Write a JFrame-based program that fills its window with randomly coloured squares



Hints:

- Use nested loops to produce the squares
- Think about your co-ordinate system (x/y)
- Investigate the `fillRect()` method of the Graphics class
- Investigate how to specify an arbitrary Color rather than using a stock Color

To get a random integer between 0 and 255: **int red = (int)(Math.*random()*255);**

Topics this week

- Animation with threads
- Creating a simple game object class
- Making a game which animates an array of game object instances

Animation with Threads

- Animation is the changing of graphics over time
- E.g. moving a spaceship across the screen, changing its position by 1 pixel every 0.02 seconds
- One of the best ways to do periodic execution of code is to use **threads**
- Threads: allow multiple tasks to run independently/concurrently within a program
- Essentially, this means we spawn a separate execution 'branch' that operates independently of our program's main flow of control
- The new Thread repeatedly sleeps for (say) 20ms, then carries out animation, and calls this.repaint() on the application

Implementing Threads in Java

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

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

- Your class **must** now provide an implementation for the run() method, which is executed when a thread is started, and serves as its “main” function i.e.:

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

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

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

Typical Actions of an Animation Thread

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

Threads Test

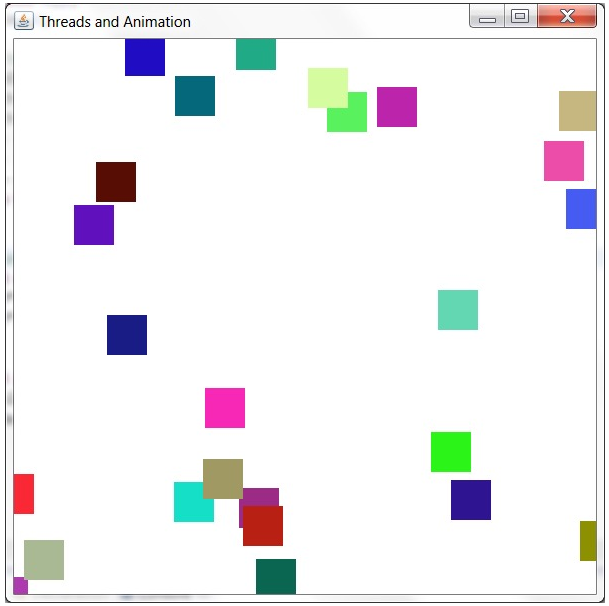
```
2  import java.awt.*;
3  import javax.swing.*;
4
5  public class ThreadsTest implements Runnable {
6
7      public ThreadsTest() {
8          Thread t = new Thread(this);
9          t.start();
10     }
11
12     public void run() {
13         System.out.println("Thread started");
14
15         for (int i=0; i<15; i++) {
16             System.out.println("Loop "+i+" start");
17             try {
18                 Thread.sleep(500);
19             } catch (InterruptedException e) {
20                 // TODO Auto-generated catch block
21                 e.printStackTrace();
22             }
23             System.out.println("Loop "+i+" end");
24         }
25
26         System.out.println("Thread ended");
27     }
28
29     public static void main(String[] args) {
30         ThreadsTest tt = new ThreadsTest();
31
32     }
33
34 }
35
```

Game object classes

- Games typically have game object classes (spaceships, aliens, 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 and painting phases, we can iterate through them all and invoke the animate() and paint() methods on each instance

Week #2 Assignment

- Create a program which performs simple random animation of coloured squares
- Use two classes:
  1. MovingSquaresApplication
     - extends JFrame
     - Implements Runnable
     - has main() method
     - Member data includes an array of GameObject instances
     - Constructor method does similar setup as last week's code, and in addition instantiates the GameObjects in the array, and creates+starts a Thread
     - Uses a Thread to perform animation of the GameObjects by calling their move() methods
     - Paint() method draws the GameObjects by calling their paint(Graphics g) methods
  2. GameObject
     - Member data includes x,y,color
     - Constructor method randomises the object's position and color
     - Public move() method is used to randomly alter x,y members
     - Public paint(Graphics g) method draws the object as a square using g.fillRect()



Code should be uploaded on Blackboard (deadline: see Blackboard)

Assignment #2
Suggested Class Interfaces

*MovingSquaresApplication.java

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

public class MovingSquaresApplication extends JFrame implements Runnable {

    // member data
    private static final Dimension WindowSize = new Dimension(600,600);
    private static final int NUMGAMEOBJECTS = 30;
    private GameObject[] GameObjectsArray = new GameObject[NUMGAMEOBJECTS];

    // constructor
    public MovingSquaresApplication() {

    // thread's entry point
    public void run() {

    // application's paint method
    public void paint(Graphics g) {

    // application's entry point
    public static void main(String[] args) {

}
```

GameObject.java

```
import java.awt.*;

public class GameObject {

    // member data
    private double x,y;
    private Color c;

    // constructor
    public GameObject() {

    // public interface
    public void move() {

    public void paint(Graphics g) {

}
```

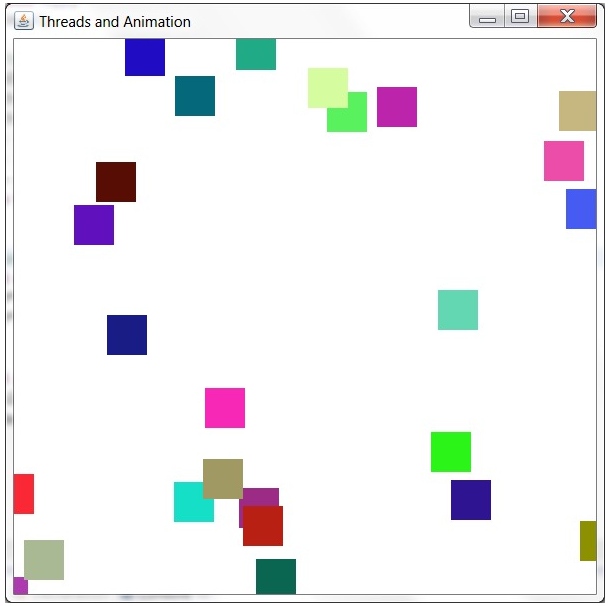
CT255 / NGT II
Digital Media / 2D Games Dev.

Week 3

sam.redfern@universityofgalway.ie
@psychicsoftware

Last Week's Assignment

- Create a program which performs simple random animation of coloured squares
- Use two classes:
  1. MovingSquaresApplication
     - extends JFrame
     - Implements Runnable
     - has main() method
     - Member data includes an array of GameObject instances
     - Constructor method does similar setup as last week's code, plus instantiates the GameObjects in the array, and creates+starts a Thread
     - Uses a Thread to perform animation of the GameObjects by calling their move() methods
     - Paint() method draws the GameObjects by calling their paint(Graphics g) methods
  2. GameObject
     - Member data includes x,y,color
     - Constructor method randomises the object's position and color
     - Public move() method is used to randomly alter x,y members
     - Public paint(Graphics g) method draws the object as a square using g.fillRect()



Topics this week

- Handling the keyboard in Java
- Loading and displaying raster images (.jpg, .png etc.)
- Moving a player's game object under control of the keyboard

Handling Keyboard Input

- In GUI-based languages such as Java (with AWT) the mouse and keyboard are handled as ‘Events’
- They may happen at any time
- They are queued as they happen and are dealt with at the next free idle time
- AWT handles events coming in from the operating system by dispatching them to any *listeners* registered to those events

Handling Keyboard Input

- Make a class that implements KeyListener
- Make sure you have an instance of this class
- Add this instance as a key listener attached to the JFrame that receives the messages from the Operating System
- The simplest way is to make your JFrame-derived class itself handle the events it receives.. (see next slide)

Handling Keyboard Input

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

public class MyApplication extends JFrame implements KeyListener  {

   public MyApplication() {  // constructor
    // send keyboard events arriving into this JFrame to its own event handlers
       addKeyListener(this);
   }

   // Three Keyboard Event-Handler functions
    public void keyPressed(KeyEvent e) {
    }

    public void keyReleased(KeyEvent e) {
    }

    public void keyTyped(KeyEvent e) {
    }
    //
}
```

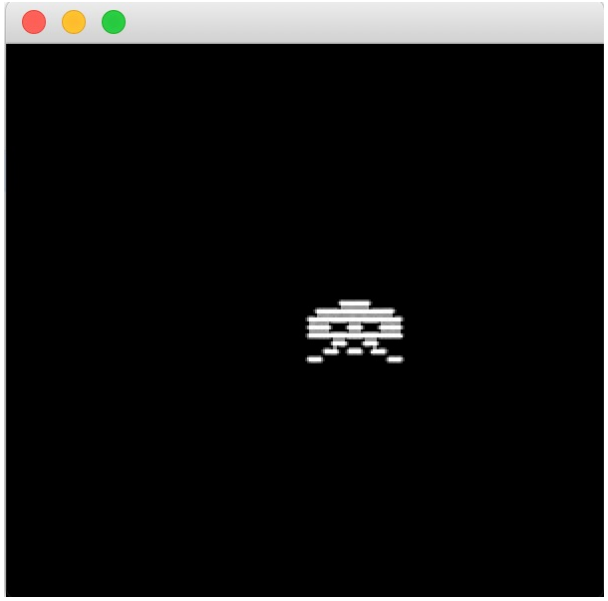
Notes:

- 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 and Runnable
- Note the extra import!! - java.awt.event.*

Loading and 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

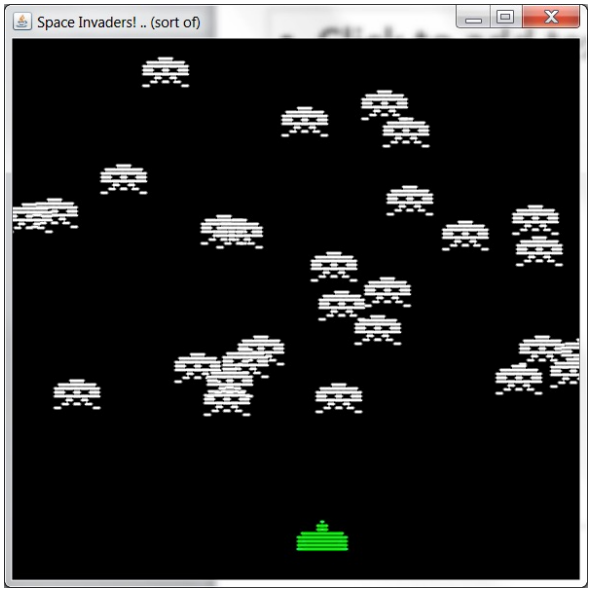
Example



```
import java.awt.*;
import javax.swing.*;
public class DisplayRasterImage extends JFrame {

// member data
private static String workingDirectory;
private Image alienImage;

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

        // load image from disk. Make sure you have the path right!
        // NB Windows uses \\ in paths whereas MacOS uses / in paths
        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("Working Directory = " + workingDirectory);
       DisplayRasterImage d = new DisplayRasterImage();
}

}
```

Week 3 exercise

- Create a JFrame-based, Runnable KeyListener application class and a separate class for handling game objects
- Use these names for your classes:
  - InvadersApplication
  - Sprite2D
- The InvadersApplication class should have, as its member data, an array of Sprite2D objects for aliens, and another single Sprite2D object for the player ship
- The InvadersApplication class should use Thread-based animation to move the aliens randomly (similar to last week)
- The Sprite2D objects display a raster image that you have loaded from disk (instead of a coloured square)
  - See **ct255-images.zip** for png files to use
- Use the left and right arrow keys to move the player spaceship, rather than moving it randomly like the aliens
  - Do NOT move the spaceship directly in the keyboard event handlers, since that will mean it will move in steps based on your keyboard repeat rate
  - The correct way to do it is to have the keyboard events notify the spaceship when movement should start and stop; the actual movement should be done every frame (i.e. 50 times per second) by the movePlayer() method suggested on the next slide



- ***Code should be submitted on Blackboard.***
- ***Deadline: before next lecture.***

Assignment #3
Suggested Class Interfaces

InvadersApplication.java

```
import java.awt.*;

public class InvadersApplication extends JFrame implements Runnable, KeyListener {

    // member data
    private static final Dimension WindowSize = new Dimension(600,600);
    private static final int NUMALIENS = 30;
    private Sprite2D[] AliensArray = new Sprite2D[NUMALIENS];
    private Sprite2D PlayerShip;

    // constructor
    public InvadersApplication() {

    // thread's entry point
    public void run() {

    // Three Keyboard Event-Handler functions
    public void keyPressed(KeyEvent e) {

    public void keyReleased(KeyEvent e) {

    public void keyTyped(KeyEvent e) {
    //

    // application's paint method
    public void paint(Graphics g) {

    // application entry point
    public static void main(String[] args) {

}
```

Sprite2D.java

```
import java.awt.*;

public class Sprite2D {

    // member data
    private double x,y;
    private double xSpeed=0;
    private Image myImage;

    // constructor
    public Sprite2D(Image i) {

    // public interface
    public void moveEnemy() {

    public void setPosition(double xx, double yy) {

    public void movePlayer() {

    public void setXSpeed(double dx) {

    public void paint(Graphics g) {

}
```

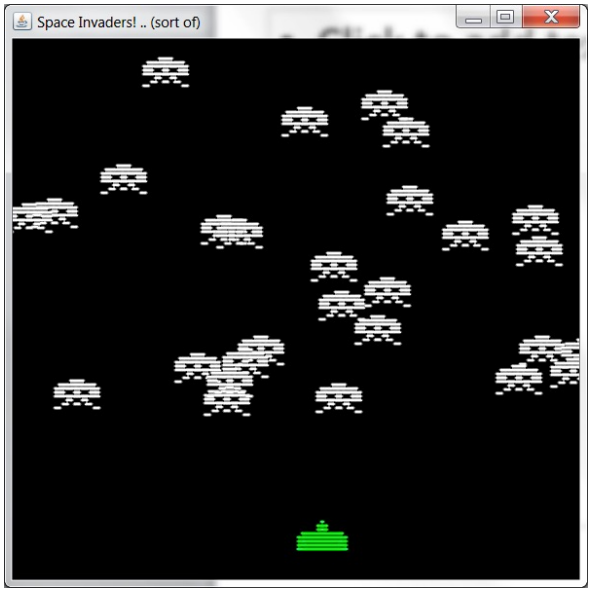
CT255 / NGT2
2D games using Java

Week 4

Sam Redfern
sam.redfern@universityofgalway.ie

Last week's exercise

- Create a JFrame-based, Runnable KeyListener application class and a separate class for handling game objects
- Use these names for your classes:
  - InvadersApplication
  - Sprite2D
- The InvadersApplication class should have an array of Sprite2D objects for aliens, another single Sprite2D object for the player ship, and should use Thread-based animation to move them all (similar to last week)
- The Sprite2D objects display a raster image that you have loaded from disk (instead of a coloured square)
- Use the left and right arrow keys to move the player spaceship, rather than moving it randomly like the aliens



Screen Flicker

- Caused by software redrawing a screen out-of-sync with the screen being refreshed by the graphics hardware (so occasionally a half-drawn image is displayed)
- Solution: use ‘double-buffering’:
  - Render all graphics to an offscreen memory buffer
  - When finished drawing a frame of animation, flip the offscreen buffer onscreen during the ‘vertical sync’ period
- Java awt provides a BufferStrategy class which applies the best approach based on your computer’s capabilities

Implementing Double Buffering

- **In the imports section at the top of the program:**

```
import java.awt.image.*;
```

- **Add a new member variable to the Application class:**

```
private BufferStrategy strategy;
```

- **In the Application class' constructor function:**

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

- **NB this code should be executed *after* the JFrame has been displayed, i.e. after setBounds() and setVisible().. why might that be?**

- **At the start of the paint(Graphics g) method (redirect our drawing calls to the offscreen buffer):**

```
g = strategy.getDrawGraphics();
```

- **At the end of the paint(Graphics g) method (indicate that we want to flip the buffers):**

```
strategy.show();
```

Let's consider some refactoring..

Sprite2D

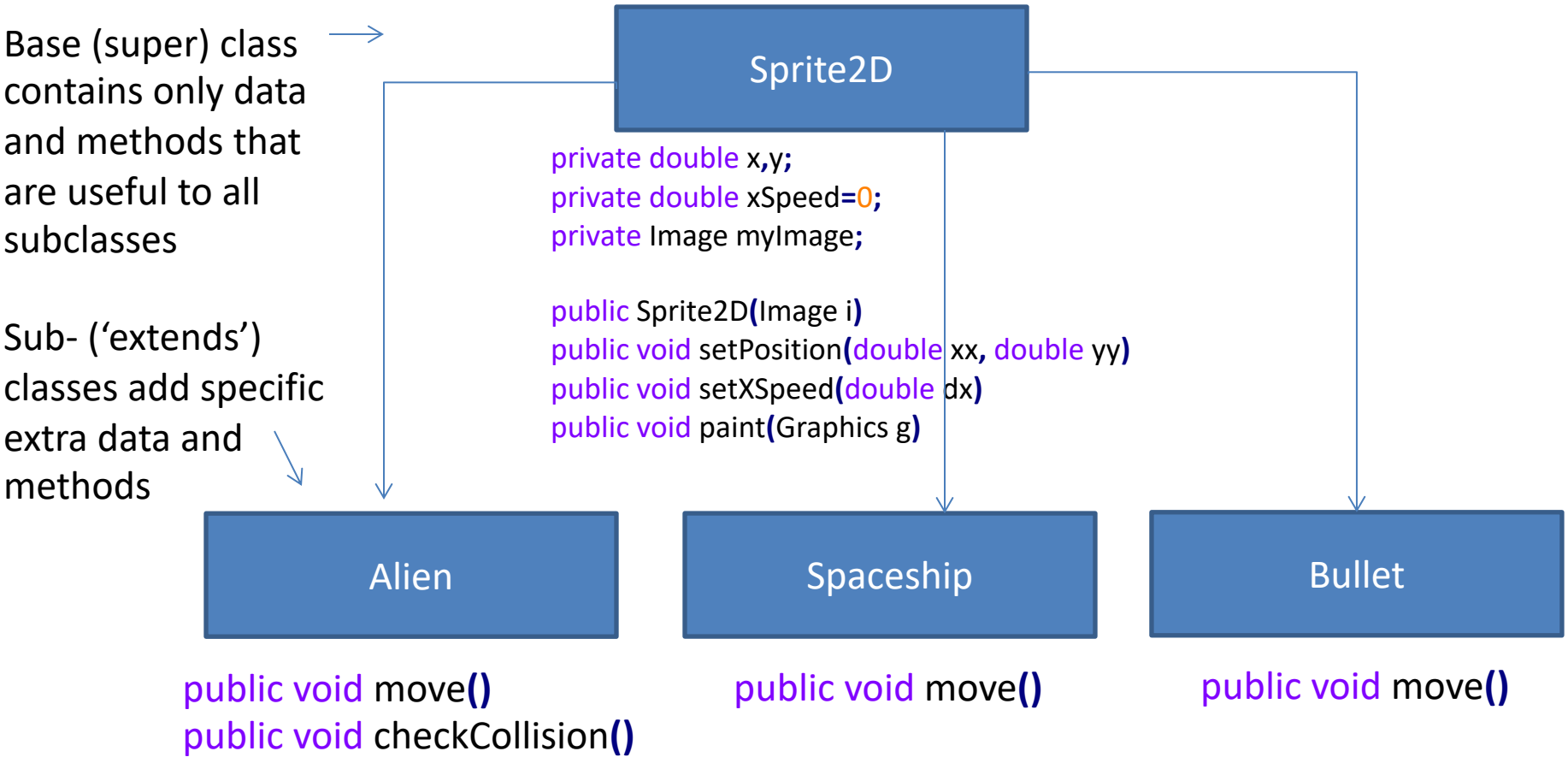
```
private double x,y;
private double xSpeed=0;
private Image myImage;

public Sprite2D(Image i)
public void moveEnemy()
public void setPosition(double xx, double yy)
public void movePlayer()
public void setXSpeed(double dx)
public void paint(Graphics g)
```

We're currently using one class to handle both Aliens and the PlayerShip objects

Some member variables and methods are used by both types of objects, while others are specific to one or the other

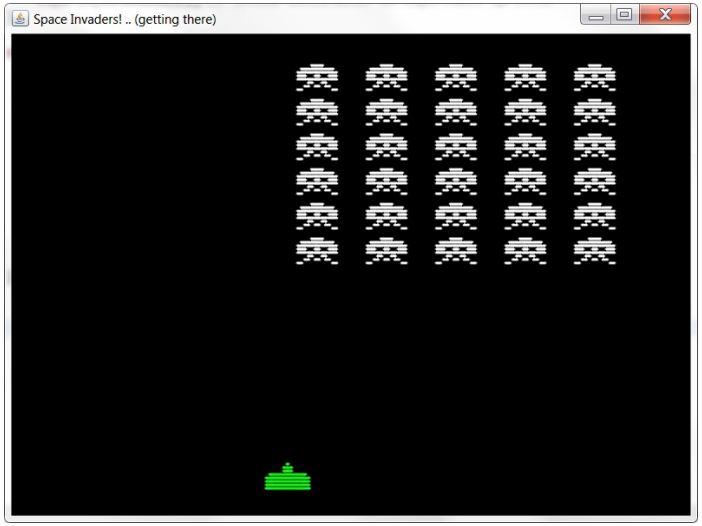
What about when we add Bullets as a third type of object? -> the Sprite2D class gets bloated, confusing and inefficient

Let's consider some refactoring..



This week's assignment

- We're moving closer to a finished game!
- Refactor the game:
  - Make the application window larger (800x600)?
  - Create an **Alien** class and a **Spaceship** class, both subclasses of **Sprite2D**. Move functionality from Sprite2D to the new classes as appropriate.
  - Modify the member variables of the **InvadersApplication** class so that it stores an array of **Alien** objects and a single **Spaceship** object (previously all were Sprite2D objects)
  - Make sure all of the above is working before moving on!



- Implement double buffering to get rid of flickering
- Initialise the aliens in a grid formation rather than randomly positioned
- Modify alien movement so that they all move left or right together (i.e. aliens should use the xSpeed variable similar to how the spaceship does)
- Make all the aliens reverse their movement direction and move down a bit when *any* of them hits the edge of the screen.. ***but how?***

Some useful points regarding class inheritance

1. Rather than using ‘private’ members in the superclass, declare them as ‘protected’ in order for the subclass to be able to access them
2. To call the constructor from a base class, use super();

- E.g:

```
public class Spaceship extends Sprite2D {
        public Spaceship(Image i) {
                super(i); // invoke constructor on superclass (Sprite2D)

                // and do any Spaceship-specific initialisation here..

        }
}
```

Suggested class interfaces

InvadersApplication.java

```
import java.awt.*;

public class InvadersApplication extends JFrame implements Runnable, KeyListener {

    // member data
    private static final Dimension WindowSize = new Dimension(800,600);
    private BufferStrategy strategy;
    private static final int NUMALIENS = 30;
    private Alien[] AliensArray = new Alien[NUMALIENS];
    private Spaceship PlayerShip;

    // constructor
    public InvadersApplication() {

    // thread's entry point
    public void run() {

    // Three Keyboard Event-Handler functions
    public void keyPressed(KeyEvent e) {

    public void keyReleased(KeyEvent e) {

    public void keyTyped(KeyEvent e) {
    }
    //

    // application's paint method
    public void paint(Graphics g) {

    // application entry point
    public static void main(String[] args) {

}
```

Sprite2D.java

```
import java.awt.*;

public class Sprite2D {

    // member data
    protected double x,y;
    protected double xSpeed=0;
    protected Image myImage;
    int winWidth;

    // constructor
    public Sprite2D(Image i, int windowWidth) {

    public void setPosition(double xx, double yy) {

    public void setXSpeed(double dx) {

    public void paint(Graphics g) {

}
```

Alien.java

```
import java.awt.Image;

public class Alien extends Sprite2D {

    public Alien(Image i, int windowWidth) {

    // public interface
    public boolean move() {

    public void reverseDirection() {

}
```

Spaceship.java

```
import java.awt.Image;

public class Spaceship extends Sprite2D {

    public Spaceship(Image i, int windowWidth) {

    public void move() {

}
```

CT255 / NGT2 / Digital Media

Week 5

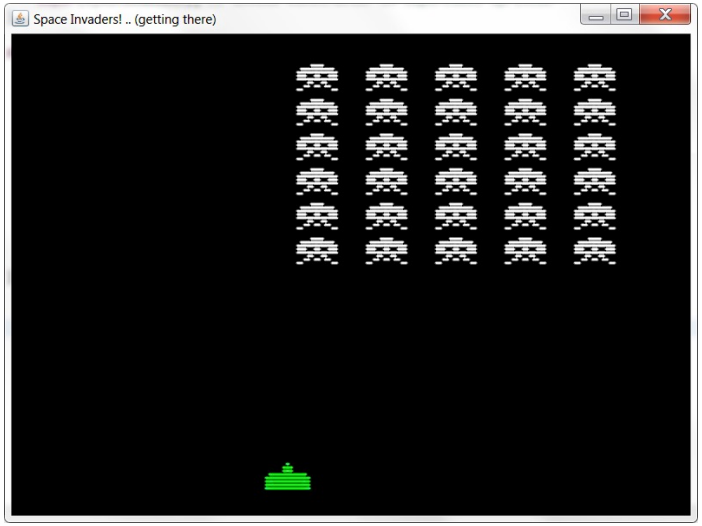
[2D Games in Java]

Dr. Sam Redfern

sam.redfern@universityofgalway.ie

Last week's assignment

- We're moving closer to a finished game!
- Refactor the game:
  - Make the application window larger (800x600)?
  - Create an **Alien** class and a **Spaceship** class, both subclasses of **Sprite2D**. Move functionality from Sprite2D to the new classes as appropriate.
  - Modify the member variables of the **InvadersApplication** class so that it stores an array of **Alien** objects and a single **Spaceship** object (previously all were Sprite2D objects)
  - Make sure all of the above is working before moving on!

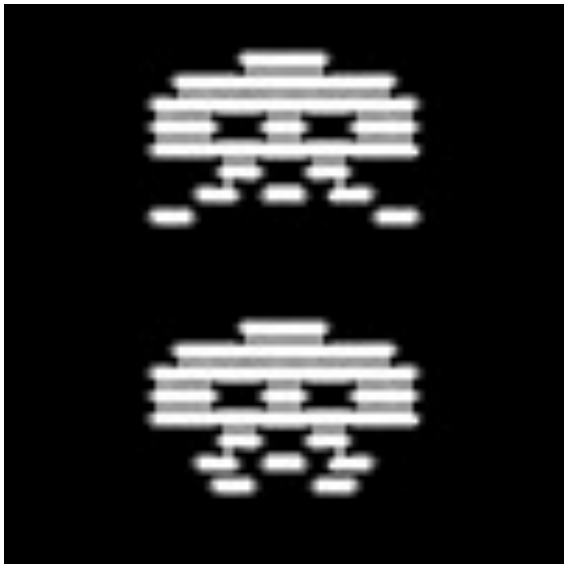


- Implement double buffering to get rid of flickering
- Initialise the aliens in a grid formation rather than randomly positioned
- Modify alien movement so that they all move left or right together (i.e. aliens should use the xSpeed variable similar to how the spaceship does)
- Make all the aliens reverse their movement direction and move down a bit when *any* of them hits the edge of the screen.. somehow

Topics this week

- Animated 2D sprites
- Collision detection in 2D raster games
- ArrayLists
- Game States

Animated 2D sprites

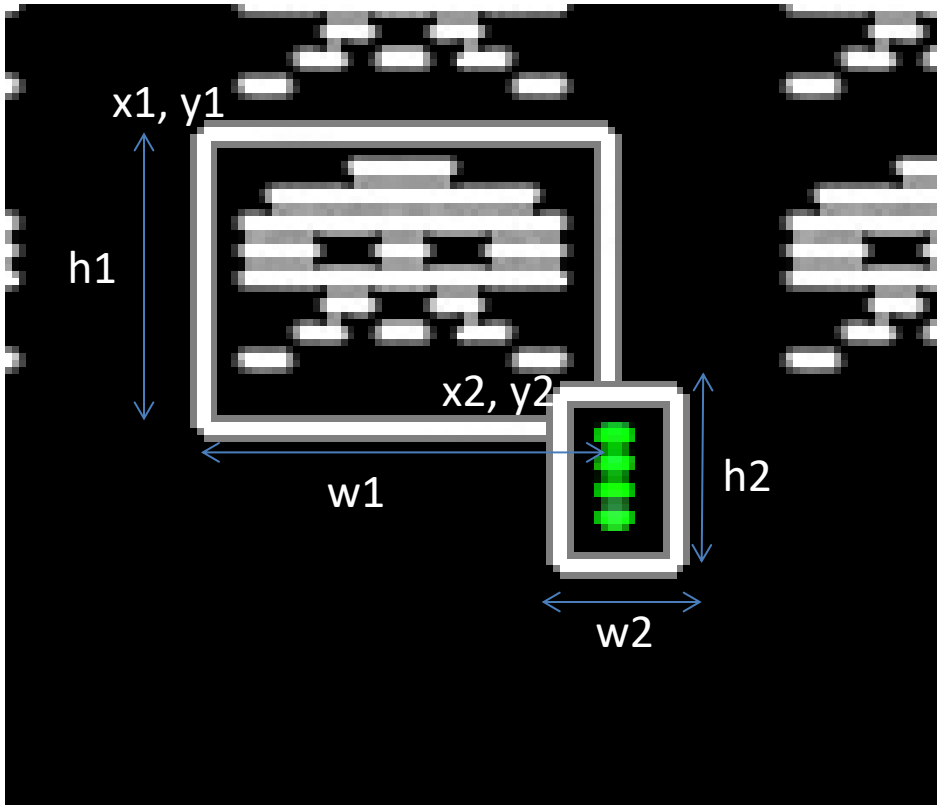


- Simply load two or more images, rather than just one
- Alternate between, or cycle through, the images
- For our game, switching image once per second (i.e. every 50th re-draw) is about right

e.g. use this in a modified Sprite2D class:

```
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);
}
```

Collision Detection



- Check for overlapping rectangles..

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

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)?
- 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

This week's assignment – Finishing off the Invaders game!

- Modify the Sprite2D class so that it has two separate member Images, rather than one. When painting, it should alternate between these images every 50 frames
- For the Alien class constructor, receive two animation frames. For the Spaceship class, one frame will do.
- Create a new class, **public class PlayerBullet extends Sprite2D**, and program it to fire when the spacebar is pressed. It should be initialised to the player ship's x/y position, and move upwards (negative y direction) every frame
- While calling the move method on the bullet, check for collision with each Alien object.
- You'll need some way of knowing whether the bullet and each Alien are alive, in order to decide whether to paint them
- If possible, enable several bullets at a time rather than just one
- Add game states (in progress or in menus): you will need to modify various methods of the application class based on the state it's in
- Add scoring
- Add collision detection between aliens and the player spaceship (game over when this happens => switch back to menus state)
- When all aliens are killed, re-create a new, faster-moving wave of them
- I would suggest you carefully assess where certain code is running (e.g. setting the initial positions of aliens and player ship) – the constructor is no longer the best place for this – you will need to create new methods such as startNewWave() and startNewGame()


Space Invaders!
GAME OVER
Press any key to play
[Arrow keys to move, space to fire]


Space Invaders!
Score: 120 Best: 1040

Suggested class interfaces

InvadersApplication.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.awt.image.*;
import java.util.ArrayList;
import java.util.Iterator;

public class InvadersApplication extends JFrame implements Runnable, KeyListener {

    // member data
    private static final Dimension WindowSize = new Dimension(800,600);
    private BufferStrategy strategy;
    private static final int NUMALIENS = 30;
    private Alien[] AliensArray = new Alien[NUMALIENS];
    private Spaceship PlayerShip;
    private Image bulletImage;
    private ArrayList bulletsList = new ArrayList();

    // constructor
    public InvadersApplication() {

    // thread's entry point
    public void run() {

    // Three Keyboard Event-Handler functions
    public void keyPressed(KeyEvent e) {

    public void keyReleased(KeyEvent e) {

    public void keyTyped(KeyEvent e) {
    //

    // method to handle shooting
    public void shootBullet() {

    // application's paint method
    public void paint(Graphics g) {

    // application entry point
    public static void main(String[] args) {

}
```

Sprite2D.java

```
import java.awt.*;

public class Sprite2D {

    // member data
    protected double x,y;
    protected double xSpeed=0;
    protected Image myImage, myImage2;
    int framesDrawn=0;
    int winWidth;

    // constructor
    public Sprite2D(Image i, Image i2, int windowWidth) {

    public void setPosition(double xx, double yy) {

    public void setXSpeed(double dx) {

    public void paint(Graphics g) {

}
```

Spaceship.java

```
import java.awt.Image;

public class Spaceship extends Sprite2D {

    public Spaceship(Image i, int windowWidth) {

    public void move() {

}
```

Alien.java

```
import java.awt.Graphics;

public class Alien extends Sprite2D {

    public boolean isAlive = true;

    public Alien(Image i, Image i2, int windowWidth) {

    public void paint(Graphics g) {

    // public interface
    public boolean move() {

    public void reverseDirection() {

}
```

PlayerBullet.java

```
import java.awt.Image;

public class PlayerBullet extends Sprite2D {

    public PlayerBullet(Image i, int windowWidth) {

    public boolean move() {

}
```

A note on Java Collection Classes

- Java provides a number of useful classes for dealing with collections of objects
- More advanced/flexible than arrays
- To allow a number of bullets rather than just one at a time, consider using an ArrayList
- See: http://tutorials.jenkov.com/java-collections/list.html [discuss in class]
- E.g. code to add a PlayerBullet object, and to draw all PlayerBullet objects:

```
public void shootBullet() {
   // add a new bullet to our list
   PlayerBullet b = new PlayerBullet(bulletImage,WindowSize.width);
   b.setPosition(PlayerShip.x+54/2, PlayerShip.y);
   bulletsList.add(b);
}
```

To remove an element *while iterating the list*:
`iterator.remove();`
(a 'for' loop is not safe for this)

```
Iterator iterator = bulletsList.iterator();
while(iterator.hasNext()){
   PlayerBullet b = (PlayerBullet) iterator.next();
   b.paint(g);
}
```

CT255 / NGT2

Week 6
[2D Games in Java]

Dr Sam Redfern
sam.redfern@universityofgalway.ie

Last week's assignment: Finishing off the Invaders game

- Add game states (in progress or in menus): you will need to modify various methods of the application class based on the state it's in
- Add scoring
- Add collision detection between aliens and the player spaceship (game over when this happens => switch back to menus state)
- When all aliens are killed, re-create a new, faster-moving wave of them
- I would suggest you carefully assess where certain code is running (e.g. setting the initial positions of aliens and player ship) – the constructor is no longer the best place for this – you will need to create new methods such as startNewWave() and startNewGame()

Conway's Game of Life..

- http://en.wikipedia.org/wiki/Conway's_Game_of_Life
- A famous 'cellular automata' 0-player game from 1970
- Each cell follows a simple set of rules as the game progresses, and a (relatively) complex system behaviour emerges
- Gives us the opportunity to study:
  - Handling the mouse in Java
  - 2D arrays
  - Reading/writing files
- And will also form the basis of two further projects:
  - Making procedural 'cave-like' maps with cellular automata
  - Solving mazes /pathfinding using the A* algorithm

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, and the mouse wheel, can be handled by implementing the MouseMotionListener and MouseWheelListener interfaces
- Step 1: have your class implement MouseListener
- Step2: In the class constructor: addMouseListener(**this);**
- Step 3: implement the methods below

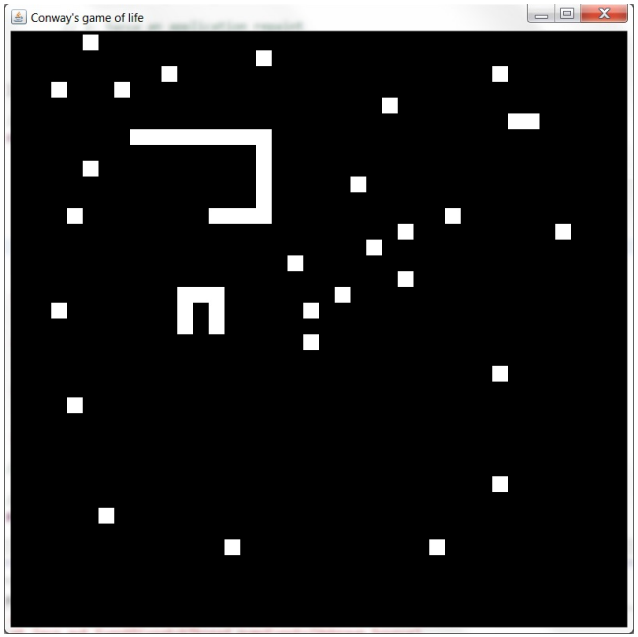
```
// mouse events which must be implemented for MouseListener
    public void mousePressed(MouseEvent e) { }

     public void mouseReleased(MouseEvent e) { }

     public void mouseEntered(MouseEvent e) { }

     public void mouseExited(MouseEvent e) { }

     public void mouseClicked(MouseEvent e) { }
```

Methods of the MouseEvent class

- int getClickCount()
  - Returns the number of quick, consecutive clicks the user has made (including this event). For example, returns 2 for a double click.
- int getX()
- int getY()
- Point getPoint()
  - Returns the (x,y) position at which the event occurred, relative to the component that fired the event.
- int getButton()
  - Returns which mouse button, if any, has a changed state. One of the following constants is returned: NOBUTTON, BUTTON1, BUTTON2, or BUTTON3.

This week's assignment
Starting the Game of Life

- Create a new Java project, with a main application class that extends JFrame and implements Runnable and MouseListener
- The window should be 800x800 pixels in size
- Use double buffering to avoid flicker when we animate [next week]
- Implement periodic repainting (and later, animation) via a Thread
- Create a 2 dimensional array to store the game state: e.g. a 40x40 array of Booleans, assuming that we want each cell to be 20x20 pixels
- When the mouse clicks on the window, toggle the state of the game state cell at that position (i.e. true becomes false, and false becomes true)
- The paint method should paint, as a rectangle, each game state cell that is currently 'true'



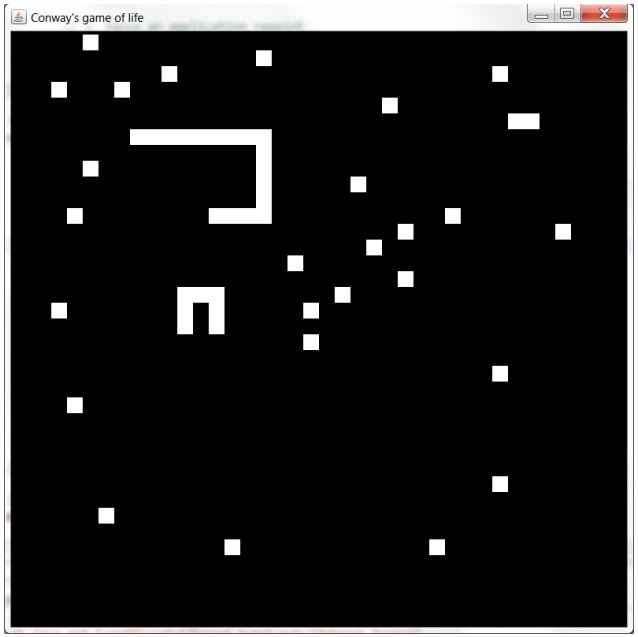
CT255 / NGT2

Week 7
[2D Games in Java]

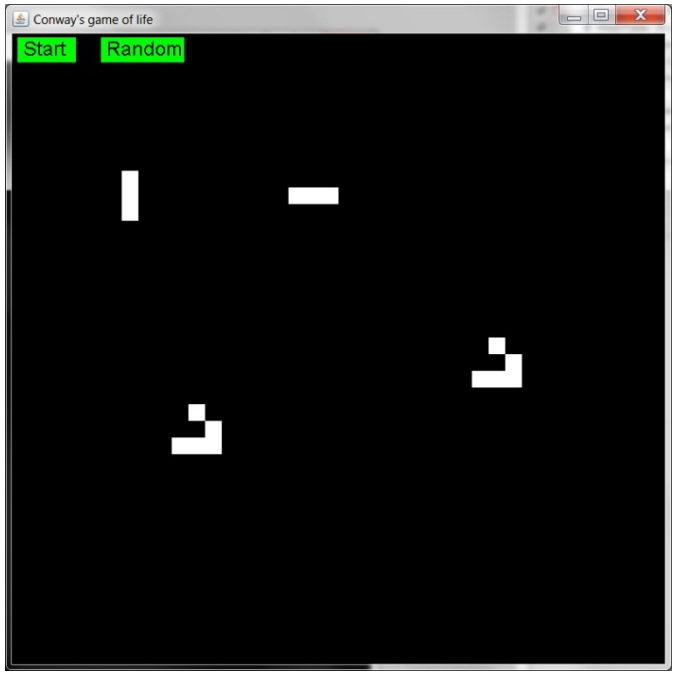
Dr. Sam Redfern
sam.redfern@universityofgalway.ie

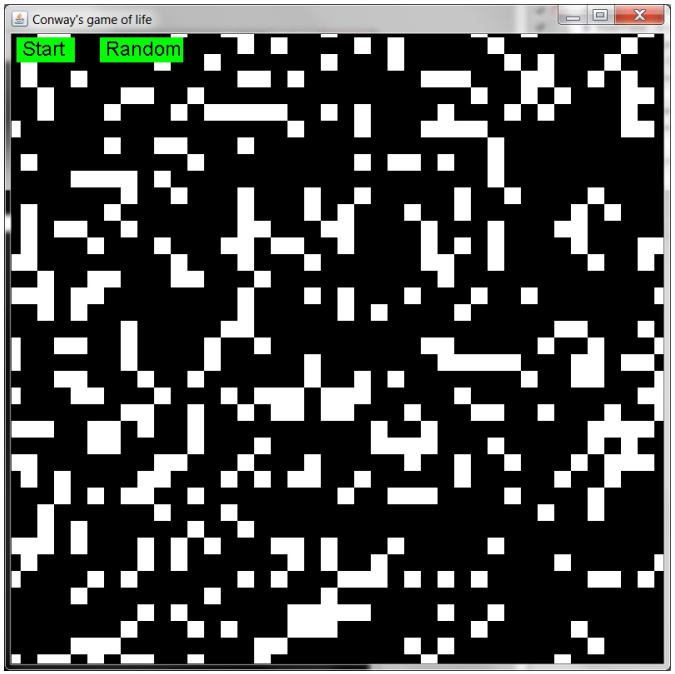
Last week's assignment
Starting the Game of Life

- Create a new Java project, with a main application class that extends JFrame and implements Runnable and MouseListener
- The window should be 800x800 pixels in size
- Use double buffering to avoid flicker when we animate [next week]
- Implement periodic repainting (and later, animation) via a Thread
- Create a 2 dimensional array to store the game state: e.g. a 40x40 array of boolean
- When the mouse clicks on the window, toggle the state of the game state cell at that position (i.e. true becomes false, and false becomes true)
- The paint method should paint, as a rectangle, each game state cell that is currently 'true'



This week's assignment





- Add game states (playing and not playing)
- When not playing, render two rectangles as 'buttons'
- Modify the mousePressed method so that it checks for clicks on the button's regions
  - Start – switches the game state to 'playing'
  - Random – randomises the game state
- When in playing state, apply the rules of Conway's Game of Life at each repaint (see next slide)

Conway's Life: Rules

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.

- We will need to iterate through all game cells, counting the amount of live neighbours that each has, before applying the above rules

Conway's Life: Rules

- Each generation (iteration) is created by applying the above rules simultaneously to every cell in its preceding generation: births and 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're 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' is switched to 'front' after applying the rules to every cell

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

Checking the 8 neighbours of each cell

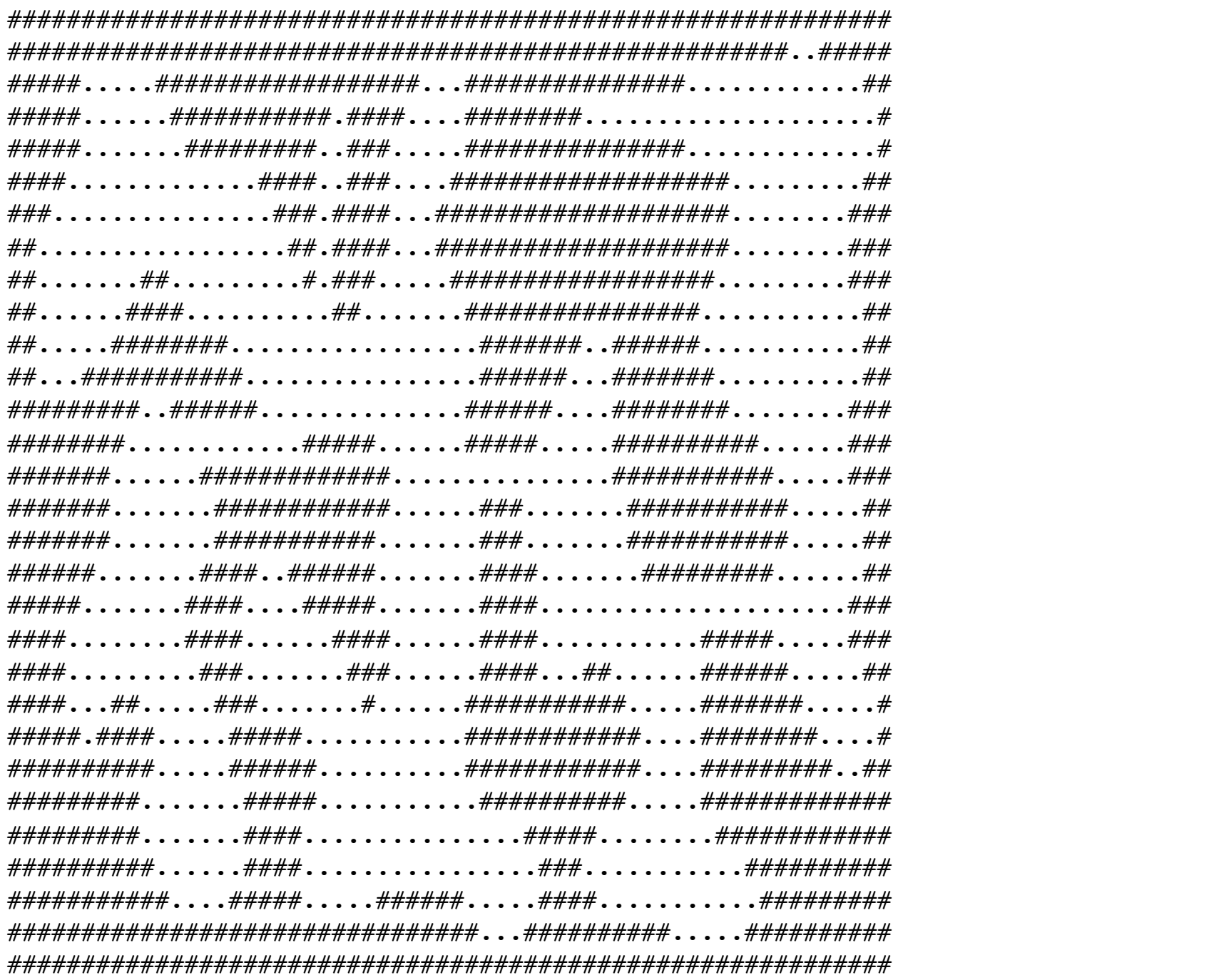
```
for (int x=0;x<40;x++) {
    for (int y=0;y<40;y++) {
            // count the live neighbours of cell [x][y][0]
            for (int xx=-1;xx<=1;xx++) {
                for (int yy=-1;yy<=1;yy++) {
                    if (xx!=0 || yy!=0) {
                        // check cell [x+xx][y+yy][0]
 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

                    }
                }
            }
     }
}
```

NB 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.

Another example of a Cellular Automata algorithm in use

- The image on the next slide 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 4 times:
    - Calculate the number of wall neighbours of 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.



Exam Question (2017)

- Your task is to write a Java class to implement this cellular automata algorithm:

- The class should store the cave-like structure in suitable member data
- The data should be randomly initialized according to the 1st step of the algorithm indicated above.
  - Hint: use Math.random() to generate a random float between 0 and 1
- The 2nd step of the algorithm (which repeats 4 times) should be implemented. You should pay particular attention to array bounds when examining a cell's neighbours.
- The resulting data should be printed to the console, (using System.out.println) as the '#' and '.' symbols, as shown below.

“Genetix”

An artificial life program I wrote a while ago (1998) – in order to learn Java!

This Artifical Life program simulates the evolution of a population of abstract creatures (*'agents'*). The 'genetic make-up' of each agent is defined by its speed and vision abilities, which determine how fast it can move and how far it can see. The colour of an agent reflects its genetics- the greener the agent, the better its vision is; the redder an agent is, the faster it can move. When the program starts, all agents have speed and vision scores of 1, and appear as khaki-green blobs.

In order to survive, agents must eat food, and on each move (*'epoch'*) an agent will move towards the greatest source of food that is within its vision range. Food is depicted by grey blobs: light grey indiciates a strong food source, while dark grey indicates a weak food source.

Healthy agents may reproduce (asexually). In most cases, an agent's offspring will be identical to it; occasionally, a newly born agent may have either its speed or its vision abilities increased or decreased (*'mutated’*).

Before running the program, you can decide the number of food deposits, the size of each deposit, the speed at which food replenishes after being eaten, and the number of agents.

http://www2.it.nuigalway.ie/~sredfern/genetix.html

Some examples of “Genetix” running

- “Conquest”
  - in this example, the effect of population pool size is evident as several separate populations develop on the 'islands' of food, and the agents from the larger islands eventually discover and conquer the less advanced agents from the other islands.
  - https://www.youtube.com/watch?v=30ztf6bMZSY

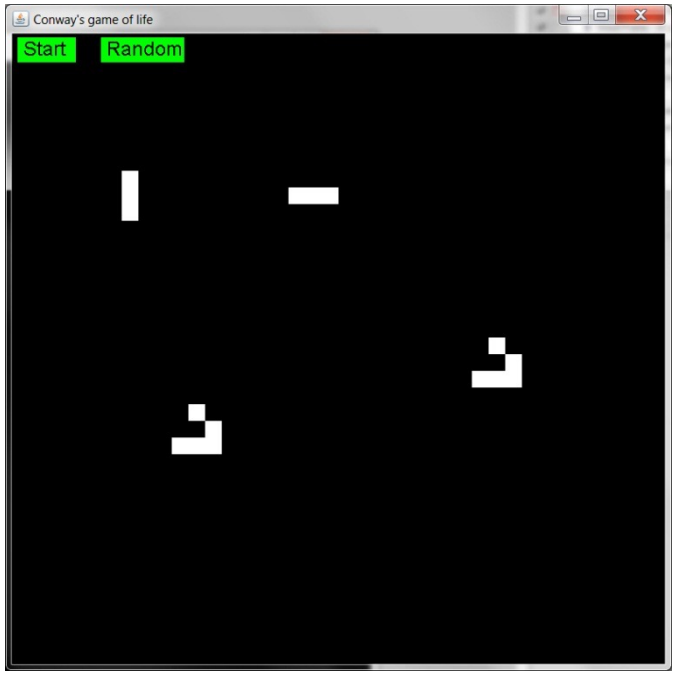
- “Extinction”
  - in this example, slow-growing food leads to cycles of population explosion, famine, and mass migration. Eventually, the instability of this model becomes evident as total extinction occurs.
  - https://www.youtube.com/watch?v=RJv0Z-sO17o

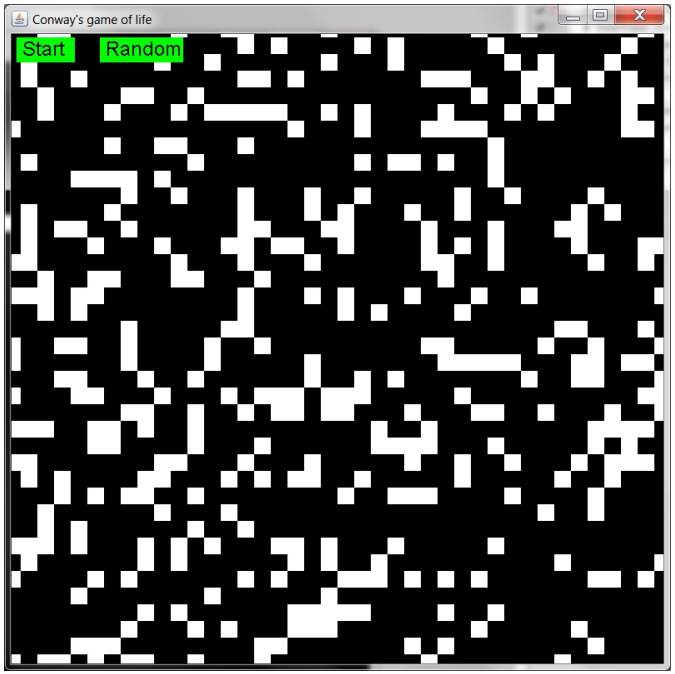
CT255
NGT2 – 2D Games in Java

Week 8
[2D Games in Java]

Dr. Sam Redfern
sam.redfern@universityofgalway.ie

Last week's assignment
Conway's Game of Life





- Add game states (playing and not playing)
- When not playing, render two rectangles as 'buttons'
- Modify the mousePressed method so that it checks for clicks on the button's regions
  - Start – switches the game state to 'playing'
  - Random – randomises the game state
- When in playing state, apply the rules of Conway's Game of Life at each repaint (see next slide)

Topics this week

- Loading and saving using text files
- Mouse move events
- Introducing A* pathfinding

Reading from text files

- The **java.io** package provides file handling classes
- **FileReader** to read from a text file
- **BufferedFileReader** to do so more efficiently (reads larger blocks and buffers/caches them)
- *Exception handling is required..*
- **BufferedFileReader:**
  - Use **FileReader** class constructor to open a file
  - Use **readLine()** method to read a line of text (returns a String)
  - Use **close()** method to close file

Sample code

```
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) { }
```

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

Sample Code

```
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) { }
```

This reads all (CR-separated) lines from the file

Writing to text files

- Use the **FileWriter** and **BufferedWriter** classes
- **BufferedFileWriter:**
  - Use **FileWriter** class constructor to open a file
  - Use **write(String s)** method to write a line to the file (CR appended automatically)
  - Use **close()** method to close file
- E.g., to write a single string to a file:

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

Handling mouse motion events

- As well as mouse button events, we can also receive mouse movement events
- .. This is useful for making it less tedious to manually create a new initial game set-up
- Have the class implement the MouseMotionListener interface as well as MouseListener
- In the application class constructor:

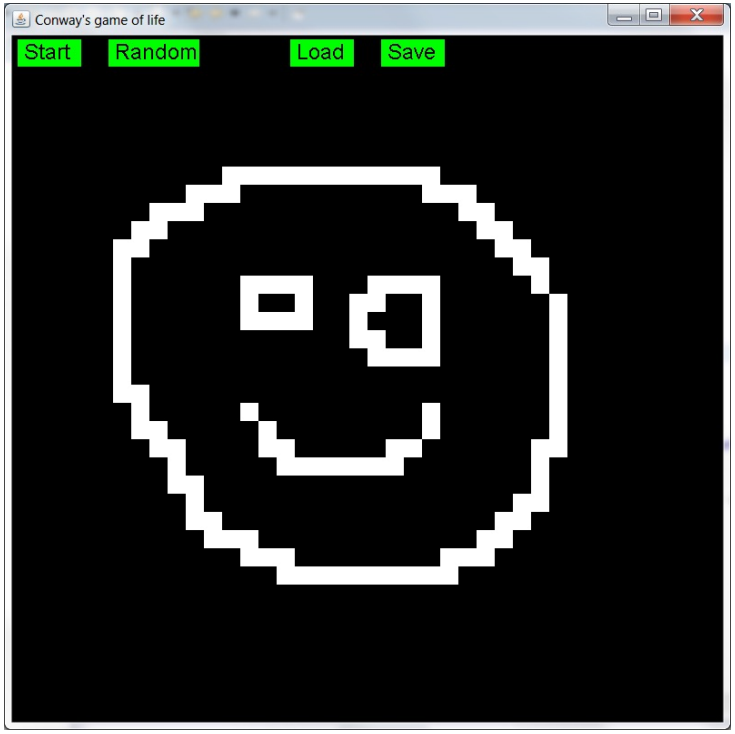
```
addMouseMotionListener(this);
```

- Add these methods (receives same data as the mouse events we have already seen):

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

This week's assignment

- Implement mouse dragging for game state setup
- Implement game state loading and saving (via 'buttons' as before)
  - How to encode the game state as string(s) ?
- Read the following A* webpage for next week!



http://www.psychicsoftware.com/AStarForBeginners.html

A* Pathfinding

- An important AI algorithm used in games and elsewhere
- Next week we will discuss implementing this in Java, and will use the Game of Life project as a basis for making a maze-solving AI game
- Read this A* webpage for next week!

  http://www.psychicsoftware.com/AStarForBeginners.html
- These are good introductions too:

  https://www.redblobgames.com/pathfinding/a-star/introduction.html
  https://www.raywenderlich.com/3016-introduction-to-a-pathfinding

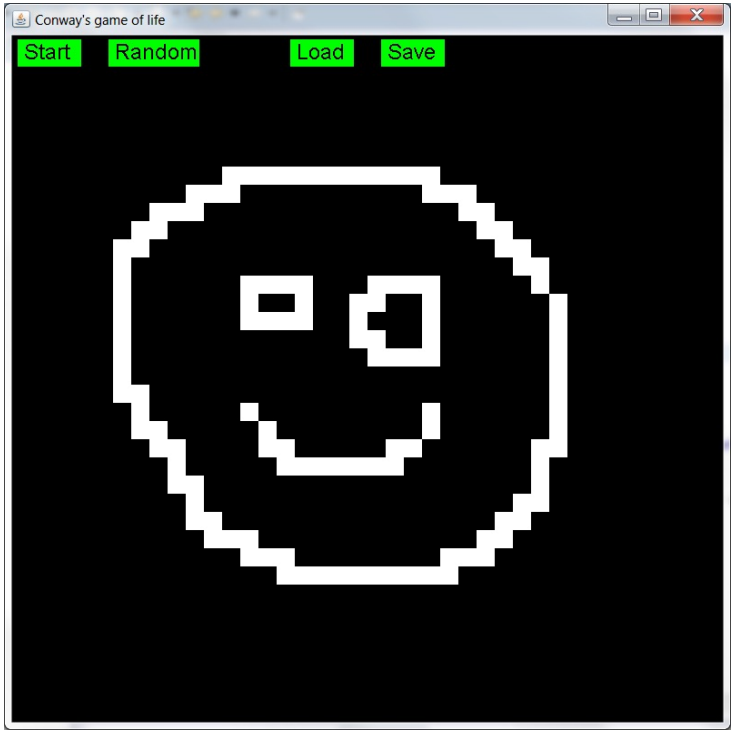
CT255
NGT2

Week 9
[2D Games in Java]

Dr. Sam Redfern
sam.redfern@universityofgalway.ie

Last week's assignment [Conway's Game of Life]

- Implement mouse dragging for game state setup
- Implement game state loading and saving (via 'buttons' as before)
- Read the following A* webpage for next week!



http://www.psychicsoftware.com/AStarForBeginners.html

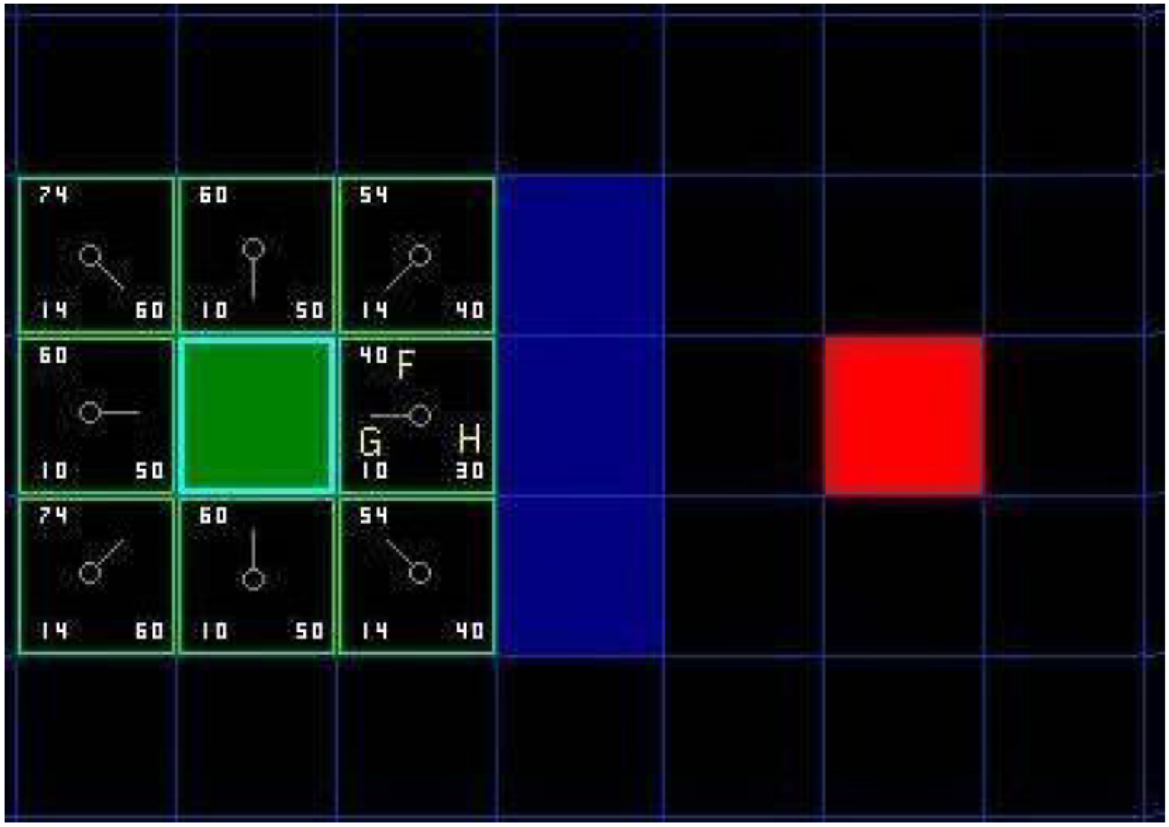
A* Pathfinding

- The fundamental operation of the A* algorithm is to traverse a map by exploring promising positions (nodes) beginning at a starting location, with 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 estimated (heuristic) cost of getting from this node to the target node. It is a best guess, since the algorithm doesn't (yet) know the actual cost
  - *f* is the sum of *g* and *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

A* Pathfinding

- The algorithm maintains two lists of nodes, the *open* list and 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 or *expanded*) but which have not themselves, yet, been expanded.
- The CLOSED LIST consists of nodes that have been expanded and which therefore should not be revisited.
- Progress is made by identifying the most promising node in the open list (i.e., the one with the lowest *f* value) and expanding 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* and *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.



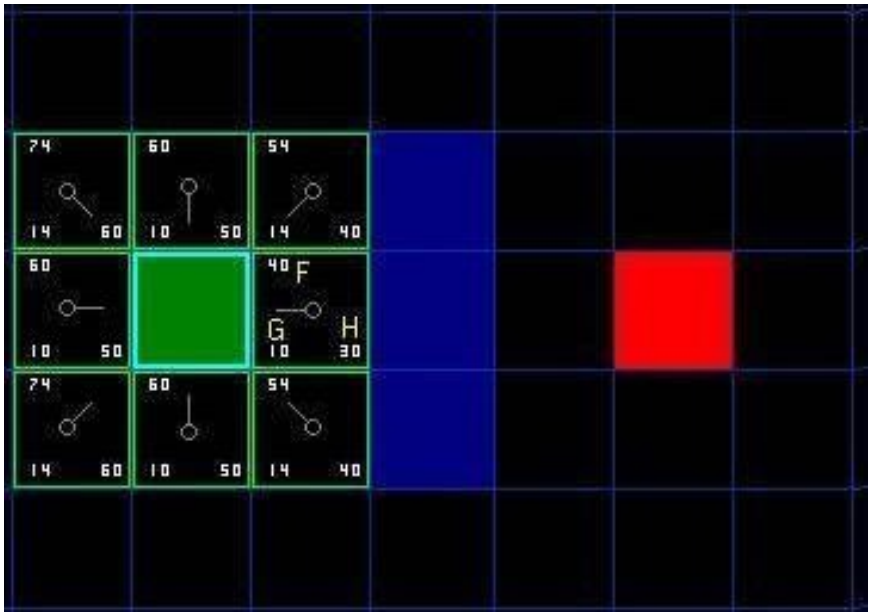
known cost from start to this node

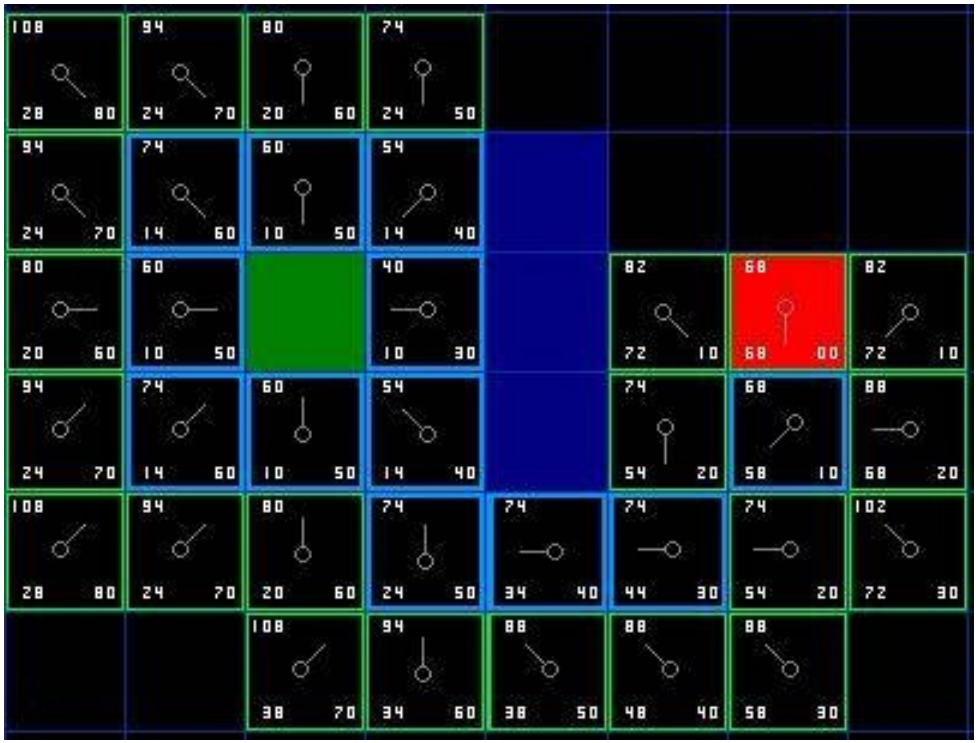
est. cost from this node to goal

$$f = g + h$$

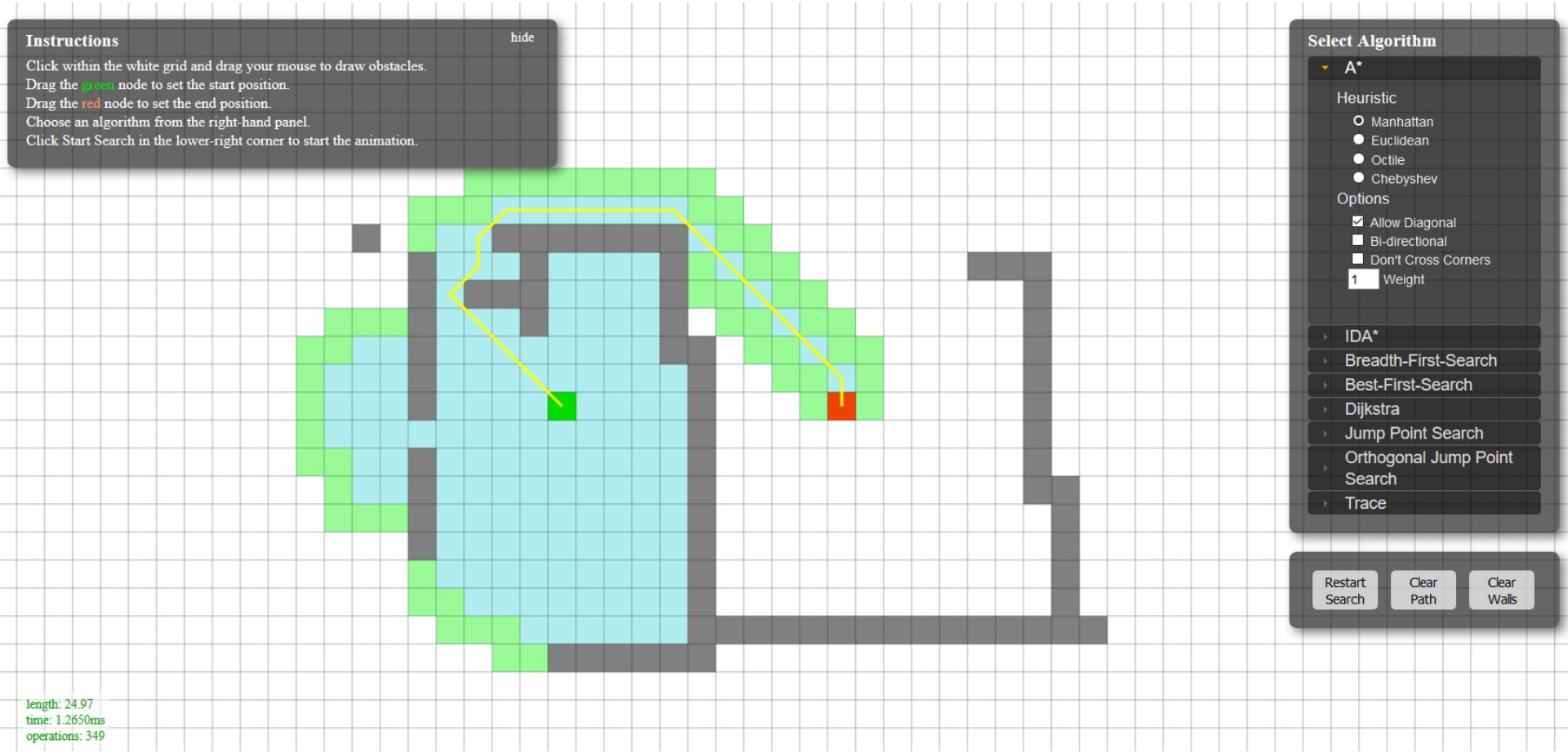
est. cost from start to goal via this node

images from: http://www.policyalmanac.org/games/aStarTutorial.htm


74
14 60
60
10 50
54
14 40
60
10 50
40 F
G H
10 30
74
14 60
60
10 50
54
14 40


108
28 80
94
24 70
80
20 60
74
24 50
94
24 70
74
14 60
60
10 50
54
14 40
80
20 60
60
10 50
40
10 30
82
72 10
68
68 00
82
72 10
94
24 70
74
14 60
60
10 50
54
14 40
74
54 20
68
58 10
88
68 20
108
28 80
94
24 70
80
20 60
74
24 50
74
34 40
74
44 30
74
54 20
102
72 30
108
38 70
94
34 60
88
38 50
88
48 40
88
58 30

https://qiao.github.io/PathFinding.js/visual/



(PathFinding.js.html)

Implementing A* Pathfinding..

- What data do we need? How might we structure the data?
  - Start loc, target loc
  - Nodes to map the game board (2D array of nodes)
  - Walkable/unwalkable map (i.e. our original 2Darray of booleans)
  - Open list (as linked list of nodes?)
  - Storage of final path (as a stack of nodes?)

- What are the initial conditions for this data?
  - Each wall node is unwalkable -> ‘closed’
  - All the rest are not open and not closed
  - Calculate f,g,h for the **starting node** and set to ‘open’

Implementing A* Pathfinding..

- What is the initial algorithmic step?
- What is the general algorithmic step?
  - Find open node with lowest f (call it X)
  - EXPAND: Look at its neighbours: any not closed and not open should become opened: calculate f,g,h and record parent position (i.e. position of X)
  - Close node X

- How will we know when we're finished?
  - If a neighbour is the target, we're done searching
  - If there are no open nodes, the maze is unsolvable

- How will we use what we found in order to have an AI-controlled '*badguy*' chase after a '*player*'?
  - Push target onto stack,
  - Push its parent onto stack
  - Push its parent onto stack
  - Etc.. Until we have pushed start node

Data for A*

- It makes sense to define a ‘node’ class, and to store nodes in specific kinds of data structures. I suggest:
  - a 2D array covering the whole game area (quick to find based on x,y)
  - a linked-list for the Open List (quick to add/remove members)
  - a stack storing the calculated path to follow (good for reversing order via LIFO)
- Of course, each node instance can happily exist in multiple data structures, since they're actually only storing pointers to it
- The nature of the A* algorithm means that we obtain our calculated path in the reverse order to how we need it
  - use a **2D array** to store all possible node cells during calculations, then when the target is found:
  - use a **stack** to store the path that a ‘badguy’ will follow, as this is a handy way to reverse the order of data
- The **linked list** is not strictly required, but since only a subset of all nodes will be Open at any given time, it's more efficient to store these in a separate data structure rather than have to search all nodes to find the best Open node to expand next

Stack

'Last in First Out' (LIFO)

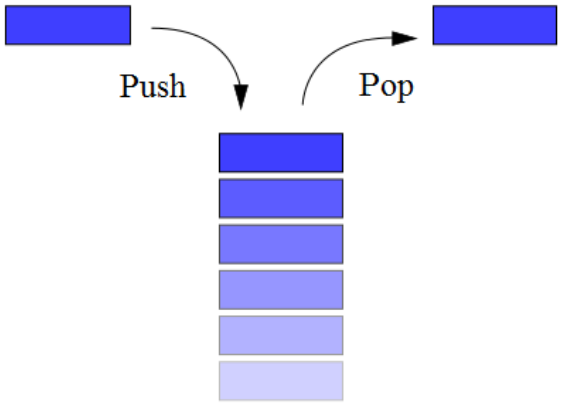
- In Java, use the Stack class:

```
import java.util.Stack;
```

Use the `push` and `pop` methods of this class

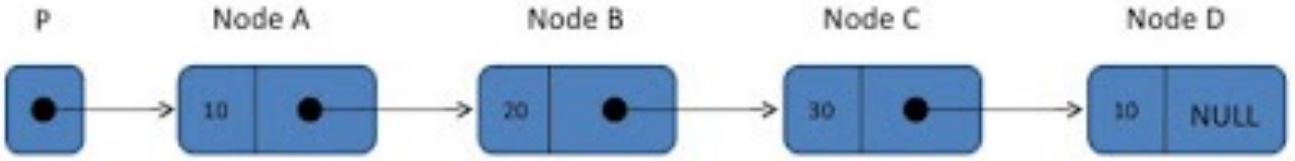
```
myStack.push(myObject);
myObject = (myClass)myStack.pop();
```

You can push any object you like onto a stack (e.g. our node object) and when popping it you must cast it to its correct data type



Linked List

- A list implemented by each item having a link to the next item.
- Head points to the first node.
- Last node points to NULL.



- Very efficient for insertion and deletion
- Can only be iterated sequentially (i.e. not random access)

```java
import java.util.*;
public class LinkedListDemo {

   public static void main(String args[]) {
      // create a linked list
      LinkedList ll = new LinkedList();

      // add elements to the linked list
      ll.add("F");
      ll.add("B");
      ll.add("D");
      ll.add("E");
      ll.add("C");
      ll.addLast("Z");
      ll.addFirst("A");
      ll.add(1, "A2");
      System.out.println("Original contents of ll: " + ll);

      // remove elements from the linked list
      ll.remove("F");
      ll.remove(2);
      System.out.println("Contents of ll after deletion: " + ll);

      // remove first and last elements
      ll.removeFirst();
      ll.removeLast();
      System.out.println("ll after deleting first and last: " + ll);

      // get and set a value
      Object val = ll.get(2);
      ll.set(2, (String) val + " Changed");
      System.out.println("ll after change: " + ll);
   }
}
```

This will produce the following result –

Output

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

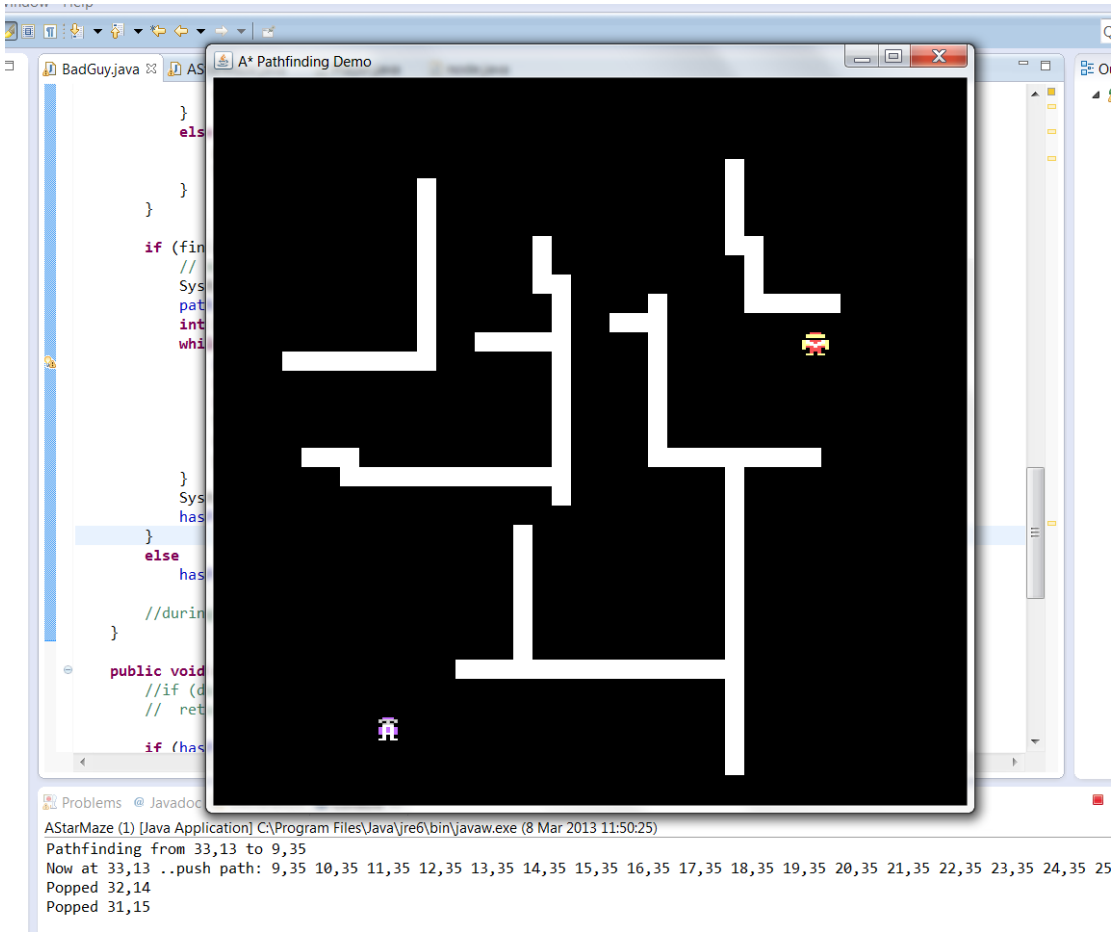
Iteration:

```
Iterator i = ll.iterator();
while (i.hasNext()) {
  string s = (String)i.next();
}
```

From: https://www.tutorialspoint.com/java/java_linkedlist_class.htm

Assignment

- Download base code for ‘badguy chases the player’ game
- This provides maze drawing, loading, saving
- It also moves the player with arrow keys, and badguy moves according to a dumb ‘straight line’ chase path – stops at walls
- Your goal is to implement A* pathfinding to make the badguy chase more effectively
- The A* path should be recalculated whenever the player moves or the maze is modified



Base code:

AStarDemoBaseCode.zip

(posted on Blackboard)

Debugging

- I'd recommend using **System.*out.print*** to debug the A* calculations and path following code

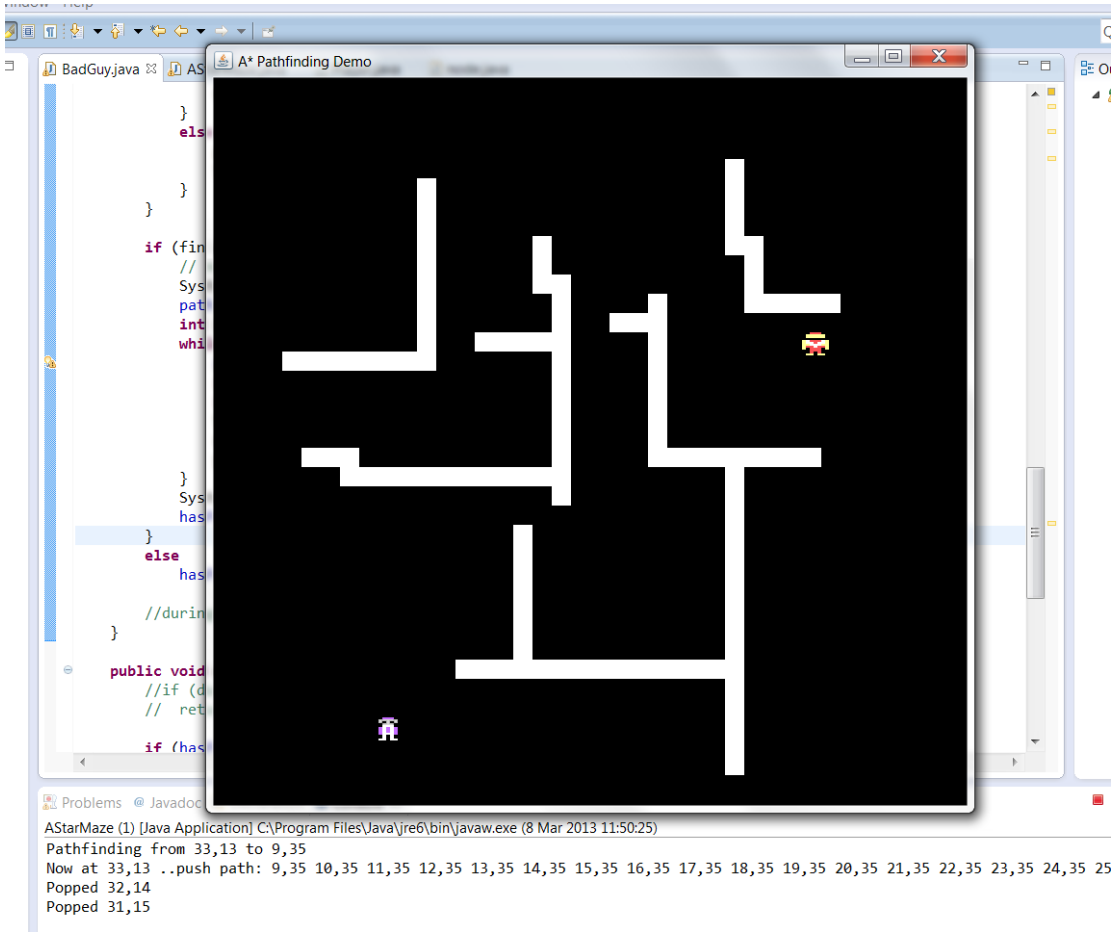
CT255
NGT2

Week 10
[2D Games in Java]

Dr. Sam Redfern
sam.redfern@universityofgalway.ie

Last Week's Assignment (A*)

- Download base code for 'badguy chases the player' game
- This provides maze drawing, loading, saving
- It also moves the player with arrow keys, and badguy moves according to a dumb 'straight line' chase path – stops at walls
- Your goal is to implement A* pathfinding to make the badguy chase more effectively
- The A* path should be recalculated whenever the player moves or the maze is modified



Base code:

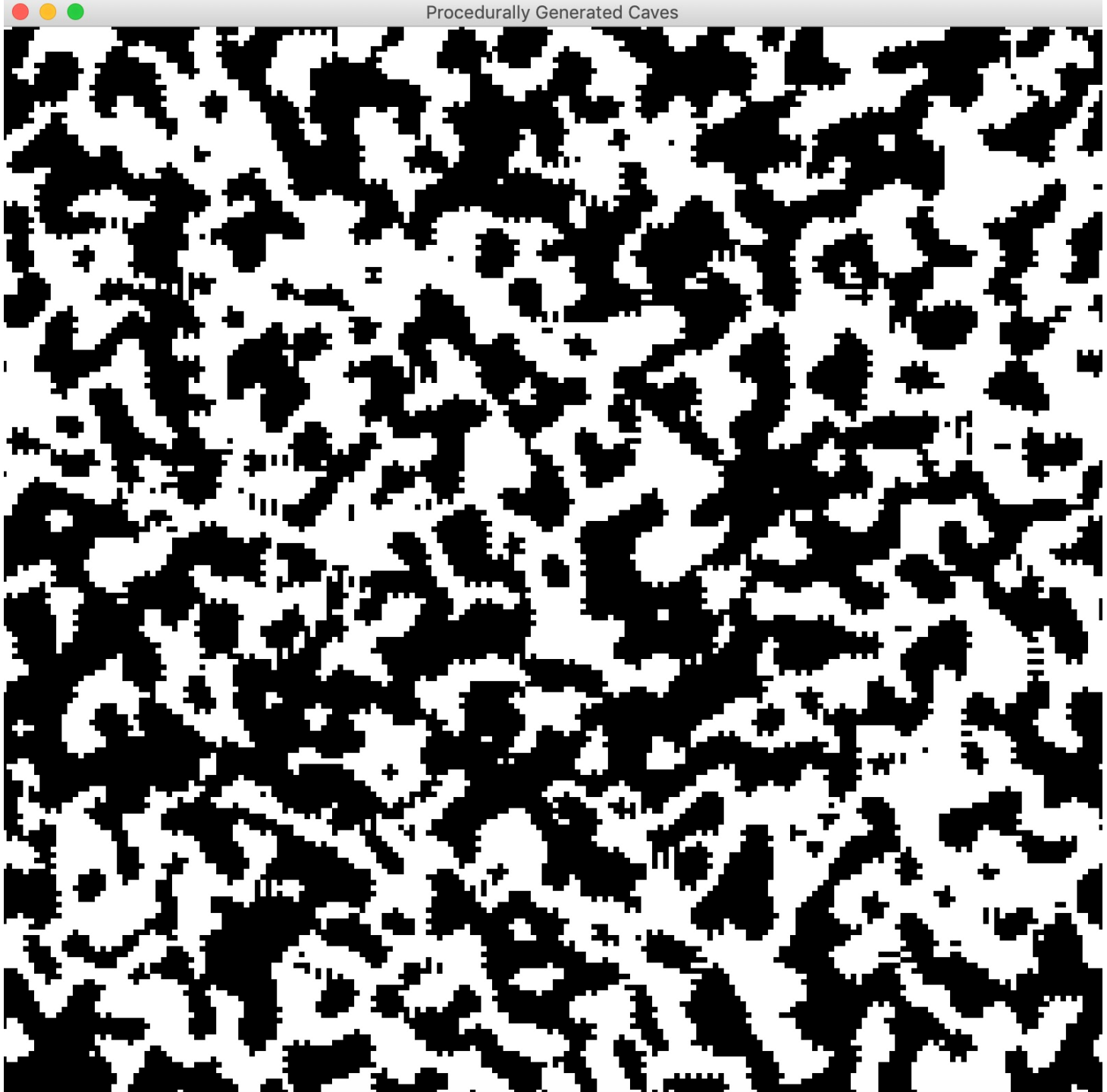
AStarDemoBaseCode.zip

(posted on Blackboard)

Another example of a Cellular Automata algorithm in use

- The image on the next slide is of an algorithmically-generated cave-like structure, for use in a 2D computer game. Each of the cells, laid out in a 200x200 grid, either has a wall (white) or a floor (black).

- 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 4 times:
    - Calculate the number of wall neighbours of 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.


Procedurally Generated Caves

Assignment

- Write a Java program which implements the above algorithm
  - You'll need to deal with the 'edge wrapping' issue
  - Put a sizeable delay between each iteration of the algorithm, so that the user can see progress rather than just the end result
  - There's no need to use double buffering