VK Computer Games
Game Development Fundamentals

Horst Pichler & Mathias Lux

Universität Klagenfurt

Books

- Free Online Book
  - Killer Games Programming in Java
    - Andrew Davison, O'Reilly Media Inc., 2005
    - http://www.oreilly.com/catalog/killergame
    - http://fivedots.coe.psu.ac.th/~ad/jg
- Books at K-Buch (student discount)
  - - Developing Games in Java
    - Bret Barker, New Riders Publishing, 2004
  - - Microsoft XNA Unleashed
    - Chad Carter, SAMS Publishing, 2008
  - - Fundamentals of Math and Physics for Game Programmers
    - Wendy Stahler, Pearson Education Inc., 2006

Web Tips

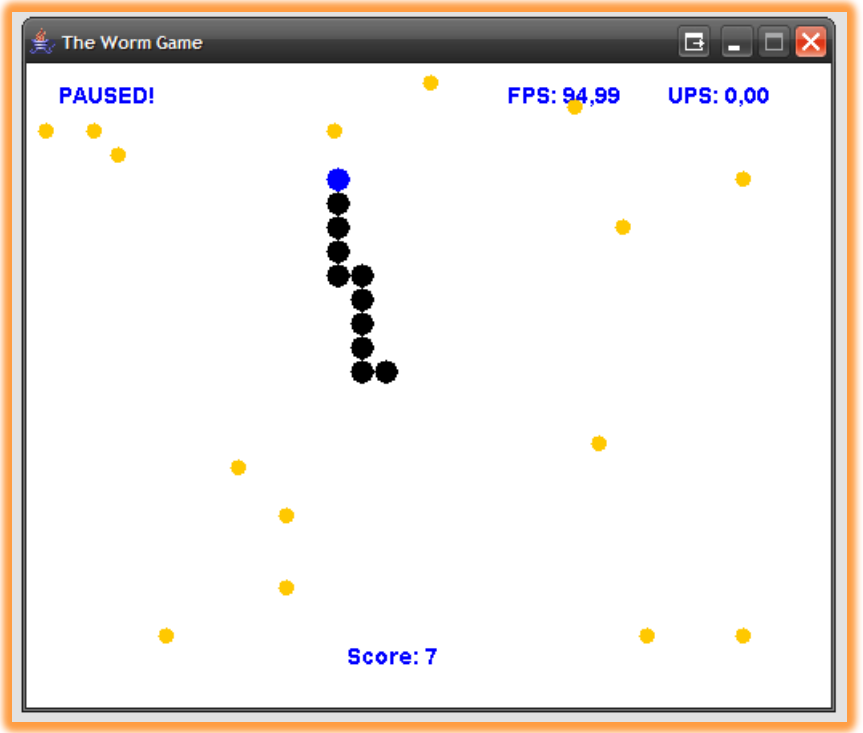
- http://www.gamasutra.com
- http://www.gamedev.net
- http://www.flipcode.com
- http://www.igda.org/Forums

- Several sample programs
  - by Andrew Davison
  - http://fivedots.coe.psu.ac.th/~ad/jg

Today‘s Course Goal

- Simple 1-player game

- Concepts learned
  - Java basics
    - threads
    - 2D graphics
  - game loop & animation
  - game state & objects
  - key-board controls
  - simple collision detection
  - frames per second & timing

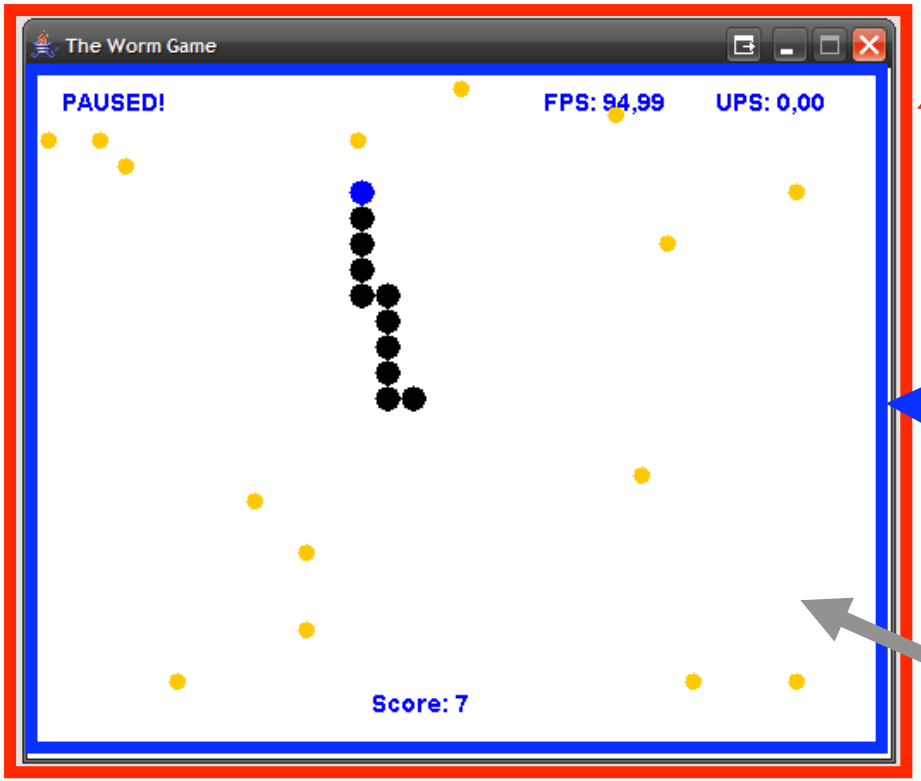


Java JFrame & JPanel

- java.swing.JFrame
  - contains 1..n components (e.g., JPanel)
  - main window – later: full-screen variant

- java.swing.JPanel
  - our „drawing area“
  - paint into it with, e.g., java.awt.Graphics-Object,

Game Components



JFrame
implements WindowListener

contains &
communicates

JPanel (drawing area)
implements Runnable

associated
with

Thread
(processes game logic)

WormChase Class


```
public class WormChase extends JFrame {

    private WormPanel wp;

    public static void main(String args[]) {
        new WormChase();
    }

    public WormChase()  {
        super("The Worm Chase");
        Container c = getContentPane(); // default BorderLayout
        wp = new WormPanel();
        c.add(wp, "Center");            // add panel to frame
        setResizable(false);
        setVisible(true);
    }
}
```

WormPanel Class I


```
public class WormPanel extends JPanel implements Runnable {
    private boolean running = false;
    private static final int PWIDTH = 500; // size of panel
    private static final int PHEIGHT = 400;
    private static final int WORM_DOTSIZE = 20; // size of segment
    private int xCoords=100, yCoords=100; // initial position
    private Thread animator; // the thread that performs the

    // set up message font to display text
    Font font = new Font("SansSerif", Font.BOLD, 24);
    FontMetrics metrics = this.getFontMetrics(font);
    private String message = “Hello Worm!“;

    public WormPanel() {
        setBackground(Color.white);
        setPreferredSize(new Dimension(PWIDTH, PHEIGHT));
        setFocusable(true);
        requestFocus(); // focus on JPanel, so receives events
    }
```

Worm Panel Class II


```
// wait until JPanel added to the JFrame before starting
public void addNotify() {
    super.addNotify(); // creates the peer
    startGame(); // start the thread
}

// initialise and start the game-thread
private void startGame() {
    if (animator == null || !running) {
        animator = new Thread(this);
        animator.start();      // calls run-method of thread
    }
}

// run the thread
run() {
    running = true;
    paintScreen();
}
```

Worm Panel Class III


```
    // paint the game objects – painting order is important!!!
    paintScreen () {
        // get the graphics-handle to paint into panel
        Graphics g = this.getGraphics();

        // clear the background (white) and seth the font.
        g.setColor(Color.white);
        g.fillRect(0, 0, PWIDTH, PHEIGHT);
        g.setColor(Color.blue);
        g.setFont(font);

        // draw message and red worm head at initial position
        g.drawString(message, 20, 25);
        g.setColor(Color.red);
        g.fillOval(xCoords, yCoords, WORM_DOTSIZE, WORM_DOTSIZE);
    }

} // end of class WormPanel
```

Java Graphics


java.awt.Graphics

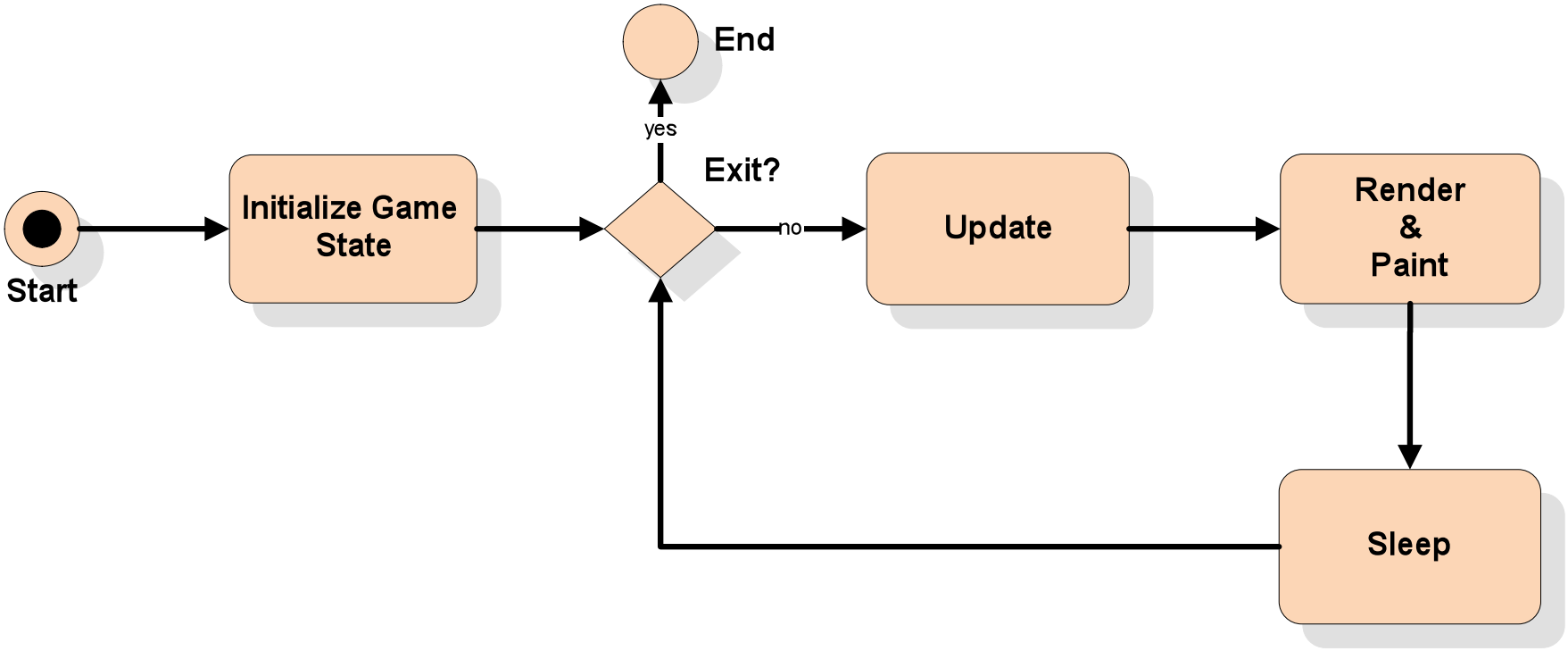
- access to basic drawing capabilities
- draw 2D-objects, like points, lines, rectangles, circles, polygons, etc.
- diverse functions, like fill, rotate, etc.
- draw bitmaps
- draw text with different fonts

➔ we use it to draw into the JPanel

Game | Animation Loop

- Game Loop
  - controls input, processing, and output of the game
  - runs within our thread in JPanel
  - btw: thread is necessary for control over timing!

Game | Animation Loop



Game | Animation Loop

- Game Loop Phases
  - gameUpdate()
    - game logic – updates the game state
    - process inputs, detect collision, calculate movements, ...
  - paintScreen()
    - draw new game state
    - output of status information (score, etc.)
    - fixed objects and moving objects (position updated above)
  - Thread.sleep(n)
    - do nothing for n millis

Game Loop in WormPanel I


```
public class WormPanel extends JPanel implements Runnable {
    ...
    static final int SLEEP_MILLIS = 25;
    int xDirection = 1;
    int yDirection = 0;
    stat
    ...
    public void run() {
        running = true;
        while (running) {
            gameUpdate();
            paintScreen();
            try {
                Thread.sleep(SLEEP_MILLIS);
            } catch (InterruptedException e) { }
        }
        System.exit(0); // so window disappears
    }
    ....
```

Game Loop in WormPanel II


```
    ...

    public void gameUpdate() {
        // move the worm's head
        xCoords = xCoords + xDirection;
        yCoords = yCoords + yDirection;

        // reverse at screen borders
        if (xCoords > PWIDTH || xDirection < 0)
            xDirection = xDirection * -1;
    }

}
```

Control Timing

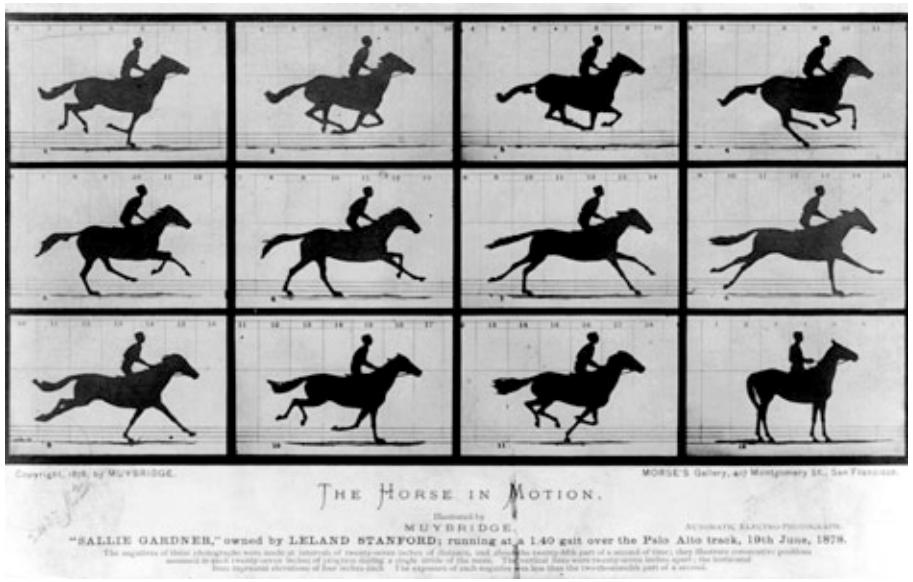
- Why do we need Thread.sleep(n)?
  - a) to gain time for other threads
    - n=0: CPU-utilization 100%
  - b) without sleep-time game would be to fast
    - compare with game-timing on old home/computers
      - single process
      - game was developed for one HW-architecture!
      - fixed sleeping times
        - » e.g., some old PCs had a slow-down button!

  →NOT POSSIBLE ON MODERN MACHINES

  →Solution: control the frame rate!

Frames

- What is a frame?
  - displaying one „picture“
  - displaying frame after frame creates illusion of motion

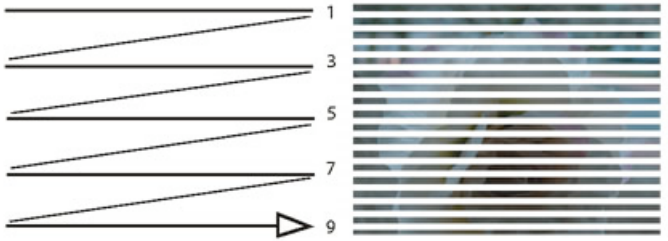


Frame Rate I

- Frames Per Second
  - aka image frequency, picture frequency
  - also measured in Hertz (oscillations per second)
  - a sufficiently high frequency is needed to avoid flickering
  - the human eye processes about 16-18 pictures per second

Frame Rates II

- Frame rates in film and television
  - 60i
    - actually 59,94 interlaced frames per second
    - 29,97 frames per second (FPS)
    - used for NTSC television since 1941
  - 50i = 25 FPS
    - PAL and SECAM television
  - 30p
    - 30 frames progressive - 30 FPS
    - noninterlaced
    - used for widescreen 1954.56



Frame Rates III

- Frame rates in film and television
  - 24p
    - noninterlaced, became de-facto standard in 1920s
  - 25p
    - derived from 50i, used for PAL television
  - 50p and 60p
    - progressive formats, used for high-end HDTV-systems

- Frame rates in games
  - 1 frame = 1 iteration of the game loop

Frame Rate IV

- What is considered a good frame rate for games?

monitors usually display at 60Hz (and above)

- low frame rates cause eye problems
- game‘s frame rate ~ monitor‘s frame rate
- surplus game frames are discarded anyway

→ does a game-frame rate >60 FPS make sense?

basically ~30 frames per second is sufficient

- e.g., Halo 3 runs at 30 FPS
- e.g., Call of Duty 4 runs at 60 FPS

Frame Rate V

- Why aim at higher frame rates than 30 FPS?
- 30 FPS should be constantly possible
- >30 FPS as buffer
  - because in games frame rate may vary heavily
    - depending on what happens in the game
      - » e.g., many objects on the screen
    - depending on other (background) processes
  - FPS drops
    - when increasing resolution
    - when adding details (shadows, distant objects, etc.)

Frame Rate VI

http://www.uni-klu.ac.at

games in early 3D-days

- FPS benchmark on „average“ machines
- high FPS-value was like a „status-symbol“
- still: try to achieve a high frame rate
  - optimise your code!

Measure the Frame Rate


```
int averageFPS = 0;
...
public void run() {
    running = true;
    double averageFPS = 0;
    long lastFPScalc = System.currenTimeMillis();

    while (running) {
        gameUpdate();
        paintScreen();
        Thread.sleep(SLEEP_MILLIS);

        // calculate average FPS every 1 second
        frameCount++;
        double diff = (System.currentTimeMillis() - lastFPScalc) / 1000.0;
        if (diff > 1) {
           averageFPS = frameCount / diff;
           frameCount = 0;
           lastFPScalc = System.currentTimeMillis();
        }
    }
```

Controlling FPS

- n in sleep(n) must be variable, not constant!

- configure average FPS to be reached
  - constant: PERIOD_MILLIS
- measure FPS
  - timeDiff = timeAtLoopEnd – timeAtLoopStart
- calculate variable sleeping time
  - new sleepTime = PERIOD_MILLIS – timeDiff

Controlling FPS


```
...
private static final long PERIOD_MILLIS = 20;  // = 50 FPS
...
public void run() {
    running = true;
    long timeDiff, beforeTime;
    while (running) {
        beforeTime = System.currentTimeMillis();
        gameUpdate();
        paintScreen();
        // calculate time left in loop & sleep time
        timeDiff = System.currentTimeMillis() - beforeTime;
        sleepTime = PERIOD_MILLIS - timeDiff;
        if (sleepTime < 0) // update+render was slow
            sleepTime = 5;    // sleep a bit anyway
        Thread.sleep(sleepTime);
    }
}
```

Timer Resolution I

- Timer Resolution
  - resolution of the timer is important for calculation of FPS (and therefore for controlling game timing)
  - minimum time between two timer calls which yield different values
    - t1 = System.currentTimeMillis();
    - t2 = System.currentTimeMillis();
    - if t1 <> t2: resolution = t2 - t1;

Timer Resolution II

- Granularity depends on Language and OS

- E.g., for Java
  - Windows 98: ~50 ms (20 FPS – not sufficient)
  - Windows 2000, XP: 10 - 15 ms (66 – 100 FPS)
  - OS X, Linux: ~1 ms (~ 1000 FPS)

- Test your timer granularity: SleepAcc & TimerRes

The Java Sleep Problem

- Thread.sleep(n) ... n measured in Millis
  - minimum sleep-time: 1 millisecond

- In reality
  - sleep time is not accurate
    - system chooses when to preempt threads
  - especially on older window systems
    - sleeps ~15 millis even for n < 15

More Accurate Timing

- Java nanoseconds timer (since Java 5)
  - milli 10^{-3}, nano=10^{-9}
  - long System.nanoTime()
- Check if sleep took longer than expected
  - introduce overSleep time
  - use overSleep to reduce next SleepTime

Introducing Oversleep


```
long overSleep = 0;
...
while (running) {
    beforeTime = System.nanoTime();
    gameUpdate();
    paintScreen();
    // calculate time left in loop & sleep time
    timeDiff = System.nanoTime() - beforeTime;
    sleepTime = PERIOD_MILLIS – timeDiff - overSleep;
    if (sleepTime < 0) // update+render was slow
        sleepTime = 5;    // sleep a bit anyway
    beforeSleep = System.nanoTime();
    Thread.sleep(sleepTime);
    overSleep = System.nanoTime() – beforeSleep – sleepTime;
}
```

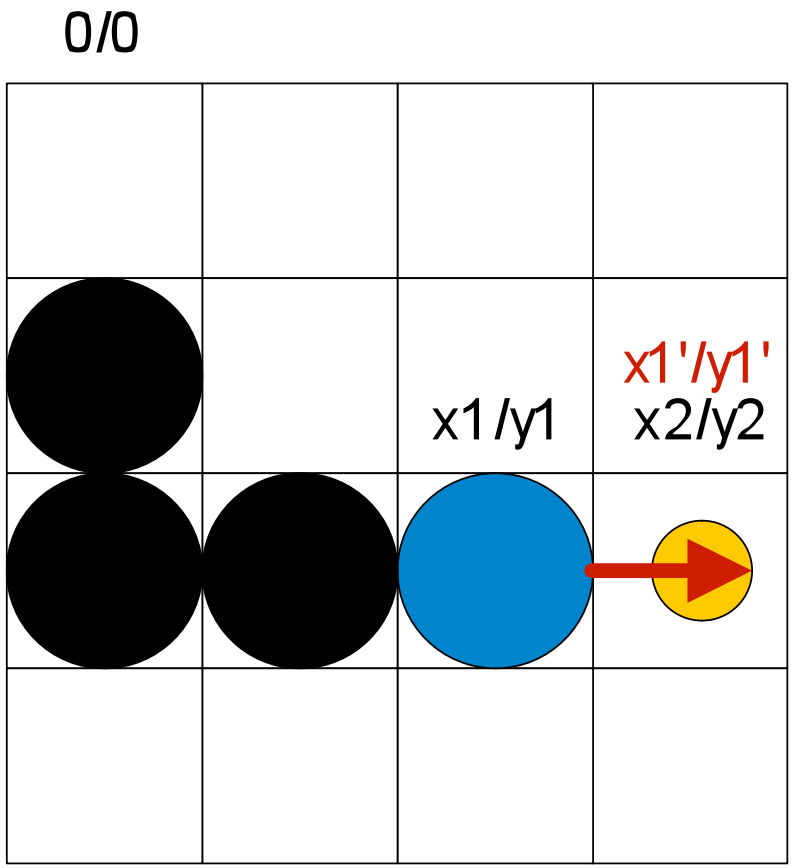
Updates Per Second

- Another game performance value
  - UPS is the number of gameUpdate()-calls per second
- Until now
  - UPS = FPS
  - but: it is not necessary to paint more frames per second than the monitor can display per second
- Save processing-time by skipping displayed frames
  - e.g., gameUpdate(), gameUpdate(), paintScreen()
  - the game „progresses“ faster (2 pixels per loop)
  - more time for complex processing tasks!!!
- Again
  - again: constant UPS-value?

Control UPS

- Goal
  - variable number of updateGame()-calls per iteration
- How?
  - introduce variable excess
    - increase excess when slow: if sleepTime < 0
  - if excess exceeds a critical value
    - call updateGame() in a while-loop
    - reduce excess in each iteration until it is 0 or ...
  - important
    - do not skip to many painted frames → „ruckeln“
    - introduce MAX_FRAME_SKIPS

Game Field – A Matrix



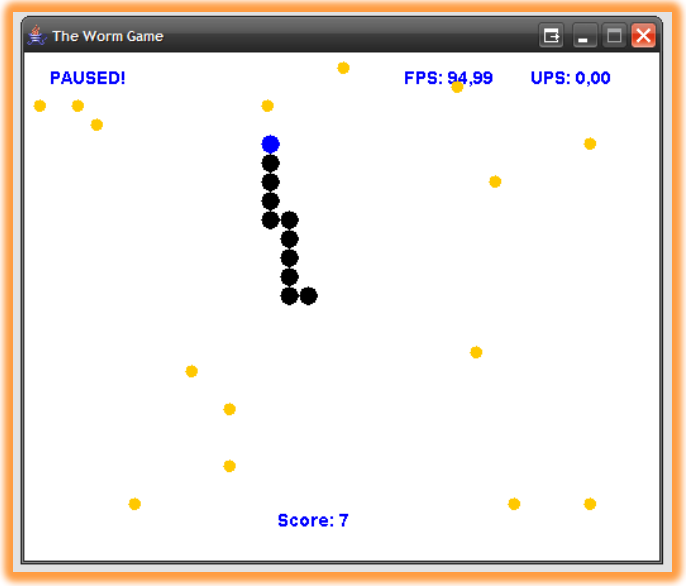
Object moves 1 field per round

Collision:

(x1‘ == x1) and (y1‘ == y2)

Implementing the Game State and Game Objects

- Different Types of Game Objects
  - Worm with head and segments
    - Worm moves in steps
    - Worm moves with const. speed (~ meters/second); FPS-independ.
  - Food
    - fixed position
    - generated randomly
  - Implement Objects as classes
    - each object stores it‘s current state (position, movement-vectors)
    - each object provides it‘s own draw-method



Food Object


```
public class Food {
        private static final int FOODSIZE = 10;
        private int x;
        private int y;

        public Food(int x, int y) {
                this.x = x;
                this.y = y;
        }

        public int getX() { return x; }
        public int getY() { return y; }

        public void draw(Graphics g) {
                g.setColor(Color.orange);
                g.fillOval(x,y,FOODSIZE,FOODSIZE);
        }
}
```

Worm Object I


```
public class Worm {
        // size and number of dots in a worm
        public static final int DOTSIZE = 20;
        public static final int RADIUS = DOTSIZE / 2;
        private static final int MAXDOTS = 40;
        private static final int INITIALDOTS = 5;

        // compass direction/bearing constants
        public static final int NORTH = 0;
        public static final int EAST = 1;
        public static final int SOUTH = 2;
        public static final int WEST = 3;
        private int currentDirection; // current direction

        // movement speed of the worm (1 move/100 millis)
        private static final int PIXEL_MOVE = DOTSIZE;
        private static final int SPEED = 100;
        private long lastMoveTime = 0; // time of last movement
```

Worm Object II


```
// stores dots of the worm (head is last)
private ArrayList<Point> dots;
private int pWidth, pHeight; // panel dimensions

public Worm(int pW, int pH) {
        pWidth = pW;
        pHeight = pH;
        dots = new ArrayList<Point>(MAXDOTS);
        initializeWorm();
}

private void initializeWorm() {
        currentDirection = EAST;
        // generate dots (starting in center)
        int x = pWidth / 2;
        int y = pHeight / 2;
        for (int i=0 ; i < INITIALDOTS-1; i++) {
                dots.add(0, new Point(x,y));
                x = x - DOTSIZE;
        }
}
```

Worm Object III


```
public void move(int newDirection) {

    if ((System.currentTimeMillis() - lastMoveTime) < SPEED)
                  return;

    lastMoveTime = System.currentTimeMillis();

    // skip movements in opposite direction of current
    boolean setDirection = true;
    if (newDirection == WEST && currentDirection == EAST)
          setDirection = false;
    ...
    ...
    if (setDirection) currentDirection = newDirection;

    // remove tail and add head
    dots.remove(0);
    addHead();              // adds head and removes tail
}
```

Worm Object IV


```
// further methods for movement and collission detection
public void move(int newDirection) { ... }
public boolean isCollission(int x, int y) { ... }
public boolean ateMe();

public void draw(Graphics g) {
       if (dots.size() > 0) {
              g.setColor(Color.black);
              int i = 0;
              while (i != (dots.size()-1)) {
                     g.fillOval(dots.get(i).x,
                        dots.get(i).y, DOTSIZE, DOTSIZE);
                     i = (i + 1) % MAXDOTS;
              }
              g.setColor(Color.blue);
              Point head = dots.get(dots.size()-1);
              g.fillOval(head.x, head.y, DOTSIZE, DOTSIZE);
       }
}
```

Game State

- The game state consists of ...
  - one worm-object
  - many food-objects
  - game information (score, lives, FPS, UPS, ...)

Initilization of Game State


```
public class WormPanel extends JPanel implements Runnable {
        ...
        static final int NUM_FOOD = 15;
        Worm fred;
        int fredsDirection = Worm.EAST;  // initial direction
        ArrayList<Food> food = new ArrayList<Food>(NUM_FOOD);
        int score = 0;
        ...
        public WormPanel() {
              setBackground(Color.white);
              ...
              // create food on random positions
              for (int i=0; i < NUM_FOOD; i++) {
                    Food f = createFood();
                    food.add(f);
              }
              foodEaten = 0;
              fred = new Worm(PWIDTH,PHEIGHT); // create the worm
              registerKeyListeners();       // add key-listeners
        }
```

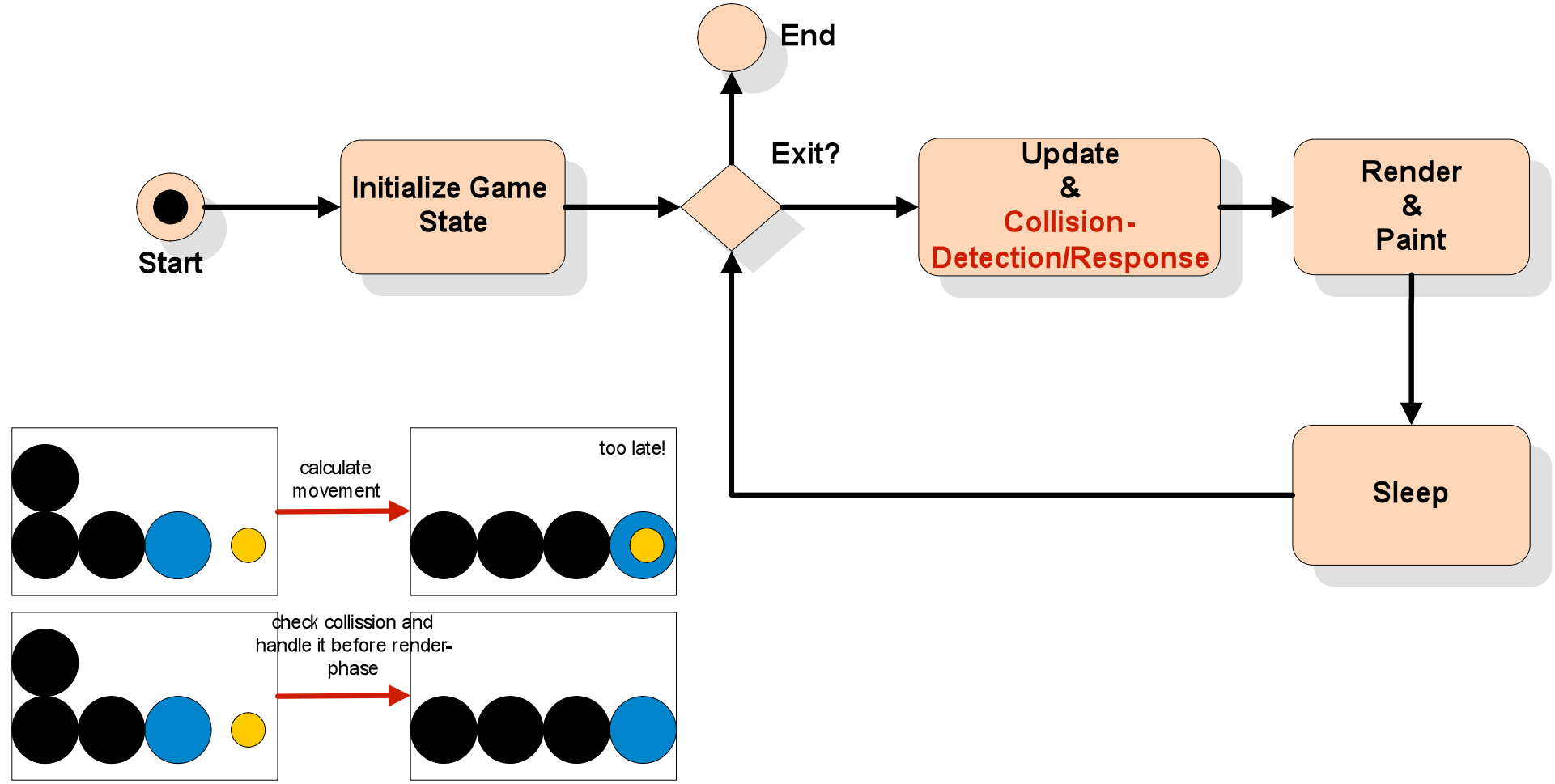
Updating the Game State I

- Updating the state means ...

  - calculate worm movement, corresponding to
    - current position and movement vectors

  - check for collissions
    - worm with food: grow worm, remove food, regrow food
    - worm with self
    - update score, check for game over

Updating the Game State



Updating the Game State


```
public class WormPanel extends JPanel implements Runnable {
        ...
        private void gameUpdate() {
                if (!isPaused && !gameOver) {
                        fred.move(fredsDirection);    // move fred
                        if (fred.ateMe())
                                gameOver = true;

                        // check if found food
                        for (Food fd : food) {
                                if (fred.isCollision(fd.getX(), fd.getY())){
                                        food.remove(fd);
                                        score++;
                                        food.add(createFood());
                                        fred.addHead(); // grow fred
                                }
                        }
                }
        }
```

Painting the Game State

- paint food objects
- paint worm
- paint text elements: score, FPS, etc.

Input Processing

- Keyboard events
  - stop or cancel the game
  - control your player-character

- Goal
  - listen for keyboard input-events
  - move the player character

- Implement according event-handlers

Keyboard Listener I

- Listener must be registered in the Panel
- A keyboard listener handles key events
  - keyPressed
  - keyReleased
  - keyTyped

  →press and release a key triggers 3 events
- On event
  - call corresponding listener method
  - identify key-event and update game state

Keyboard Listener II


```
public class WormPanel extends JPanel implements Runnable {
    ...

    public WormPanel() {
        setBackground(Color.white);
        setPreferredSize(new Dimension(PWIDTH, PHEIGHT));
        setFocusable(true);
        registerKeyListeners();       // add key-listeners
        requestFocus(); // focus on JPanel, so receives events
    }

    ...
```

Keyboard Listener III


```
private void registerKeyListeners() {
    addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
       int keyCode = e.getKeyCode();
       if (keyCode == KeyEvent.VK_ESCAPE) {
            running = false;
       } else if (keyCode == KeyEvent.VK_SPACE) {
            isPaused = !isPaused;
            if (isPaused) message = "PAUSED!";
            else message = "RUNNING AGAIN!";
       } else if (keyCode == KeyEvent.VK_LEFT) {
           fredsDirection = Worm.WEST;
       } else if (keyCode == KeyEvent.VK_RIGHT) {
           fredsDirection = Worm.EAST;
       } else if (keyCode == KeyEvent.VK_DOWN) {
           fredsDirection = Worm.SOUTH;
        } else if (keyCode == KeyEvent.VK_UP) {
           fredsDirection = Worm.NORTH;
        }
}
```

Window Listener I


```
public class WormChase extends JFrame implements WindowListener{

    private WormPanel wp;

    public static void main(String args[]) {
        new WormChase();
    }

    public WormChase()  {
        super("The Worm Chase");
        Container c = getContentPane(); // default BorderLayout
        wp = new WormPanel();
        c.add(wp, "Center");            // add panel to frame
        setResizable(false);
        setVisible(true);
        addWindowListener(this);
    }
```

Window Listener I


```
public class WormChase extends JFrame implements WindowListener{

    private WormPanel wp;

    public static void main(String args[]) {
        new WormChase();
    }

    public WormChase()  {
        super("The Worm Chase");
        Container c = getContentPane(); // default BorderLayout
        wp = new WormPanel();
        c.add(wp, "Center");            // add panel to frame
        setResizable(false);
        setVisible(true);
        addWindowListener(this);
    }
```

Window Listener II


```
public void windowActivated(WindowEvent e) { wp.resumeGame(); }

public void windowDeactivated(WindowEvent e) { wp.pauseGame(); }

public void windowDeiconified(WindowEvent e) {  wp.resumeGame(); }

public void windowIconified(WindowEvent e) {  wp.pauseGame(); }

public void windowClosing(WindowEvent e) { wp.stopGame();  }

public void windowClosed(WindowEvent e) {}

public void windowOpened(WindowEvent e) {}
```