Chapter 11: Sprites - Introduction

- This section deals with sprites
- They are discussed in terms of a game called *BugRunner*
  
  - The game combines aspects of *Pong* and *Space Invaders*
  
  - The basics of game play are
    
    - The player controls an ant (referred to as a *bat*, based on its use)
    - The ant stays at the bottom of the screen and can be moved left and right, using the arrow keys or mouse
      - The ant wraps around when it exits a side
    - A ball is launched into the playing area from the top of the screen with a random downward trajectory
    - The goal of the player is to intercept the ball, causing it to rebound upwards and out of the playing area
    - Scoring is in terms of *rebounds*, which has an initial value of eight
      - *Rebounds* is incremented for every ball successfully intercepted
      - *Rebounds* is decremented for every ball that reaches the ground
      - Game play is over when *rebounds* reaches zero (lose) or 16 (win)
  
  - Balls are represented by sprites, as is the ant

- The game is based on code already discussed:
  
  1. Animation loop with timing control from C2
  2. *Worm* game construction from C3, 4
  3. Image loading and image playing code from C6

- The game incorporates sound as discussed in C7 through 10
  
  - These aspects are ignored in this discussion
Chapter 11: Sprites - Introduction (2)

- Class diagram:
Chapter 11: Sprites - Defined

- KGB defines a sprite as a "moving, graphical image"
  - This definition is too simplistic
    * Any animated graphic satisfies this definition
  - A sprite is generally considered to be a data structure
    * This structure contains data like
      · A predefined image
      · Current coordinates
      · Velocity
    * Additional information may be included

- KGB states that a general-purpose sprite is hard to design, as characteristics depend on game context
  - Does the sprite move past the background?
    * Sprite needs to move horizontally and vertically
  - Does the background move past the sprite?
    * Sprite only moves vertically

- Other things that may need to be considered:
  - Need to monitor for collisions
  - May include a z value
    * In a 2D game, a z component is not required
      · Programmer must insure that sprites are drawn from farthest to closest to the eye so that they obscure each other in the proper manner when one is in front of another
    * However, explicitly storing a z value facilitates
      1. Collision detection when multiple sprites exist on the same z plane
        · In fact, it would be required in this case
      2. Drawing sprites in the proper order
        · This is especially true when sprites can be added and deleted from play
  - Also want to be able to change a sprite’s image over time
Chapter 11: Sprites - The *Sprite* Class

- This class, while not abstract, is intended to provide a generic representation
  - Things like collision detection, user interaction, sound effects are too specific to be handled in this class
  - Specific sprites should extend this class to provide the additional functionality

- Data members:

```java
public class Sprite {
    private static final int XSTEP = 5;
    private static final int YSTEP = 5;
    private static final int SIZE = 12;
    private ImagesLoader imsLoader;
    private ImagesPlayer player;
    private String imageName;
    private BufferedImage image;
    private int width, height;
    private boolean isLooping;
    private int pWidth, pHeight;
    private boolean isActive = true;
    protected int locx, locy;
    protected int dx, dy;

    - XSTEPX, XSTEPY represent default distances to move along the X and Y axes per update
    - SIZE is a default size
    - ImagesLoader is used to load the associated images (C6)
    - ImagesPlayer is used for animated images (C6)
    - (width, height) are dimensions of the sprite
    - isLooping is set true for sprites that utilize an animation loop
    - (pWidth, pHeight) are dimensions of the panel
    - isActive indicates whether a sprite is involved in game play
    - (locx, locy) represents the current position of the sprite
    - (dx, dy) represents the sprite’s speed along the x and y axes
```
Chapter 11: Sprites - The *Sprite* Class (2)

- The constructor:

  ```java
  public Sprite(int x, int y, int w, int h, ImagesLoader imsLd, String name)
  {
      locx = x; locy = y;
      pWidth = w; pHeight = h;
      dx = XSTEP; dy = YSTEP;
      imsLoader = imsLd;
      setImage(name); // the sprite’s default image
  }
  ```

- Method *setImage()*

  - Used to assign an image to a sprite
  - Utilizes an *ImagesLoader* object
  - Assumes no animation and sets *isLooping* to false
  - Code:

    ```java
    public void setImage(String name)
    {
        imageName = name;
        image = imsLoader.getImage(imageName);
        if (image == null) { // no image of that name was found
            System.out.println("No sprite image for " + imageName);
            width = SIZE;
            height = SIZE;
        } else {
            width = image.getWidth();
            height = image.getHeight();
        }
        player = null;
        isLooping = false;
    } // end of setImage()
    ```
Chapter 11: Sprites - The Sprite Class (3)

• Method `loopImage()`

  – Used to activate animation looping
  – Creates an `ImagesPlayer` object
  – Sets `isLooping` to true
  – Code:

    ```java
    public void loopImage(int animPeriod, double seqDuration)
    {
        if (imsLoader.numImages(imageName) > 1) {
            player = null;  // to encourage garbage collection of previous player
            player = new ImagesPlayer(imageName, animPeriod, seqDuration, true, imsLoader);
            isLooping = true;
        } else
            System.out.println(imageName + " is not a sequence of images");
    }
    ```

• Method `stopLooping()`

  – Undoes `loopImage()`
  – Code:

    ```java
    public void stopLooping()
    {
        if (isLooping) {
            player.stop();
            isLooping = false;
        }
    }
    ```

• Method `updateSprite()`

  – Does exactly what you’d think it does
  – Code:

    ```java
    public void updateSprite()
    {
        if (isActive()) {
            locx += dx;
            locy += dy;
            if (isLooping)
                player.updateTick();
        }
    }
    ```

  – Called by `run()` (via `gameUpdate()`) in `BugPanel`, which implements the animation framework
  – Called for every sprite on every update
Chapter 11: Sprites - Drawing Sprites

• Sprites are drawn once per animation period by *drawSprite()*
  
  – Code:

```java
public void drawSprite(Graphics g) {
  if (isActive()) {
    if (image == null) { // the sprite has no image
      g.setColor(Color.yellow); // draw a yellow circle instead
      g.fillOval(locx, locy, SIZE, SIZE);
      g.setColor(Color.black);
    } else {
      if (isLooping)
        image = player.getCurrentImage();
      g.drawImage(image, locx, locy, null);
    }
  }
}
```

– *drawSprite()* is called for each sprite by *gameRender()* , which is called immediately after *gameUpdate()* in *run()*

• If the animation loop maintains a steady rate, a sprite’s speed (in terms of pixels per second) is determined by the animation period and \((dx, dy)\)

• If the frame rate is not consistent, a steady speed can be achieved by
  
  1. Calculating the time since the last update
  2. Calculating \((dx, dy)\) in terms of this time and the desired speed
Chapter 11: Sprites - Representing Behavior - State Transition Diagrams

• In game play, sprites (characters/objects/etc.) exhibit behavior in response to events that occur in the game

• This can be thought of as moving from one state to another

• To make coding easier, KGB recommends using a graphical format as an aid to programming

• He utilizes UML state charts, but his first example is closer to a traditional state transition diagram

• A state transition diagram (used to represent finite state machines, state machines, finite automata, augmented state machines, etc.) can be used to represent the behavior of a sprite

  – A finite state machine is an abstraction that represents a system’s behavior or a process
    * It consists of
      1. *States*, that represent a generalized step/condition/situation
      2. *Transitions*, that represent events (usually associated with triggering events) that cause a change from one state to another

  – A state transition diagram is a directed graph used to represent a machine
    * Nodes represent the states
    * Directed edges represent transitions

  – Example:

  ![State Transition Diagram](image)

  – Implementation
    1. Machine class approach
      (a) Create a class for the machine
      (b) Create an instance variable for each state
      (c) Create a method for each event
Chapter 11: Sprites - Representing Behavior - State Transition Diagrams (2)

* Example:

```java
class FA {
    final static enum STATES, STATEA, STATEB, STATEC;
    int currentState = STATES;

    public FA () {
        // any initialization needed
    }

    public void eventA () {
        if (currentState == STATES) {
            currentState = STATEB;
            // process as needed
        } else if (currentState == STATEA) {
            currentState = STATES;
            // process as needed
        } else if (currentState == STATEB) {
            currentState = STATEB;
            // process as needed
        }
        ...
    }

    public void eventB () {
        if (currentState == STATES) {
            currentState = STATEB;
            // process as needed
        } else if (currentState == STATEA) {
            currentState = STATEA;
            // process as needed
        } else if (currentState == STATEB) {
            currentState = STATEA;
            // process as needed
        }
        ...
    }

    public static void main (String[] args) {
        machine = new FA();
        machine.eventA();
        machine.eventB();
        machine.eventA();
        ...
    }
```
2. State class approach (*State Pattern* approach)

(a) Create a class for the machine
(b) Create a class for each state

* Example:

```java
public interface State {
    public void eventA();
    public void eventB();
    public void eventC();
}

public StateS implements State {
    private FA machine;
    public StateS (FA machine) {
        this.machine = machine;
    }
    public void eventA () {
        machine.setState(machine.getState(StateA);
        //process as needed
    }
    public void eventB () {
        machine.setState(machine.getSTATE(StateA);
        //process as needed
    }
    ...
}

public StateA implements State {
    private FA machine;
    public StateA (FA machine) {
        this.machine = machine;
    }
    public void eventA () {
        machine.setState(machine.getState(StateS);
        //process as needed
    }
    public void eventB () {
        machine.setState(machine.getState(StateA);
        //process as needed
    }
    ...
}
```
public class FA {

    State stateS;
    State stateA;
    State stateB;
    State state#;

    State currentState = stateS; // start state

    public class FA () {
        state0 = new State0(this);
        state1 = new State1(this);
        ...
    }

    public void eventA () {
        currentState.eventA();
    }

    public void eventB () {
        currentState.eventB();
    }

    ...
}

public class Driver {

    static public void main (String[] args) {
        machine = new FA();

        machine.eventA();
        machine.eventB();
        machine.eventA();
        ...
    }

    ...
}

Chapter 11: Sprites - Representing Behavior - State Transition Diagrams (5)

3. State transition table approach
   (a) Create a table that lists states, events, and the associated transition states
   (b) Create a driver that - given a current state and event - does a table lookup to find the next state

* Example table:

<table>
<thead>
<tr>
<th>State</th>
<th>event A</th>
<th>event B</th>
<th>event C</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>S</td>
<td>A</td>
<td>#</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>A</td>
<td>#</td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>#</td>
<td>#</td>
</tr>
</tbody>
</table>

* Example:

```java
public class FA {
    final static enum STATES, STATEA, STATEB, STATE#;
    final static enum EVENTA, EVENTB, EVENT#;
    int currentState = STATES;
    private int transitionTable = {{STATEA, STATEB, STATEA},
                                {STATES, STATEA, STATE#},
                                {STATEB, STATEA, STATE#},
                                {STATE#, STATE#, STATE#}};

    public FA () {
        //any initialization needed
    }

    public void transition (int event) {
        currentState = transitionTable[currentState][event];
        //process as needed
    }

    ...
}

public class Driver {
    static public void main (String[] args) {
        machine = new FA();
        machine.transition(A);
        machine.transition(B);
        machine.transition(A);
        ...
    }
    ...
}
```
Chapter 11: Sprites - Representing Behavior - UML State Charts

• UML state charts are a variant of state transition diagrams used to show the behavior of objects in OOP

• Components (see references on Course Resources page):
  – Initial state
    * State of an object when created
    * Represented by a circle
  – States represented by rounded rectangles
    * They are structured similar to class diagrams
    * State name at the top
    * Actions listed underneath
      · Action syntax: \texttt{do / <action>}
      · These are actions that are performed while in this state
  – Transitions represented by directed edges
    * These may be labeled by \texttt{conditions}
      · Syntax: \texttt{[condition]}
      · Indicates the condition that must be true in order for the object to make the transition
      · They may also include actions to be performed while making the transition, indicated by \texttt{[<condition>]/<action-list> or just /<action-list>}
        · Conditionals sometimes represented using standard flow chart diamonds
  – Super states
    * Often, multiple states make the same transition on the same condition
    * A super state encapsulates this
    * It is represented by a round rectangle that encompasses several states
    * Common transitions are drawn once from the super state, and not from each individual component state
  – Concurrency
Chapter 11: Sprites - Representing Behavior - UML State Charts (2)

- Representation:
• Note that these are a combination of traditional state transition diagrams and flow charts, since
  1. States can represent processing
  2. Transitions can be triggered by conditions
  3. Note that the states frequently resemble object methods, as they can have processing associated with them
• Translating a UML state chart to code is not as straightforward as for traditional transition state diagrams
Chapter 11: Sprites - The Ball Sprite

- The state chart for the ball sprite:
• Data members

```java
public class BallSprite extends Sprite {
    private static final int STEP = 8;
    private static final int STEP_OFFSET = 2;
    private static final String[] ballNames = {"rock1", "orangeRock", "computer", "ball"};
    private static final int MAX_BALLS_RETURNED = 16;
    private int nameIndex;
    private ClipsLoader clipsLoader;
    private BugPanel bp;
    private BatSprite bat;
    private int numRebounds;

    - (STEP, STEP_OFFSET) are (dx, dy)
    - ballNames[]: set of images used to provide variety of images
    - nameIndex: used to access ballNames[]
    - clipsLoader: sound
    - numRebounds: counter to keep track of score

• The constructor

```java
public BallSprite(int w, int h, ImagesLoader imsLd, ClipsLoader cl, BugPanel bp, BatSprite b) {
    super( w/2, 0, w, h, imsLd, ballNames[0]);
    clipsLoader = cl;
    this.bp = bp;
    bat = b;
    nameIndex = 0;
    numRebounds = MAX_BALLS_RETURNED/2;
    initPosition();
}

- Implements the initialize state
Chapter 11: Sprites - The Ball Sprite (3)

• `initPosition()` method

```java
private void initPosition()
{
    setImage(ballNames[nameIndex]);
    nameIndex = (nameIndex+1)%ballNames.length;
    setPosition((int)(getPWidth() * Math.random()), 0);
    int step = STEP + getRandRange(STEP_OFFSET);
    int xStep = ((Math.random() < 0.5) ? -step : step); // move left or right
    setStep(xStep, STEP + getRandRange(STEP_OFFSET)); // move down
} // end of initPosition()
```

```java
private int getRandRange(int x)
// random number generator between -x and x
{
    return ((int)(2 * x * Math.random())) - x;
}
```

- Set initial position along top of panel
- Set speed (`step`)
- Set trajectory (`xStep`, call to `setStep()`)

• `updateSprite()` method

```java
public void updateSprite()
{
    hasHitBat();
    goneOffScreen();
    hasHitWall();
    super.updateSprite();
}
```

- Implements the `update` super state
- The three primary method calls roughly correspond to the states examining environment, finishing, and re-initializing (or so states KGP)
- The call to `super.updateSprite` corresponds to state `move`
- The looping has been linearized since these events can only occur 1X per update and are independent
  * Such simplifications should be taken with care
• **hasHitBat()** method

```java
private void hasHitBat() {
    Rectangle rect = getMyRectangle();
    if (rect.intersects(bat.getMyRectangle())) {
        clipsLoader.play("hitBat", false);
        Rectangle interRect = rect.intersection(bat.getMyRectangle());
        dy = -dy;
        locy -= interRect.height;
    }
}
```

– Implements the [hit bat] conditional
– Collision detection based on bounding rectangle
  * On a collision, an extra shift upwards is made to preclude another positive hit on the next update
  * This could happen if the bounding rectangle has not cleared the bat by the next update **

• **hasHitWall()** method

```java
private void hasHitWall() {
    if ((locx <= 0) && (dx < 0)) {
        clipsLoader.play("hitLeft", false);
        dx = -dx;
    }
    else if ((locx+getWidth() >= getPWidth()) && (dx > 0)) {
        clipsLoader.play("hitRight", false);
        dx = -dx;
    }
}
```

– Implements the [hit right wall] and [hit left wall] conditionals
– Note that there is not an adjustment to location here to preclude a double hit detection
  * Rather, direction of movement is considered when detecting these collisions
Chapter 11: Sprites - The Ball Sprite (5)

- `goneOffScreen()` method

```java
private void goneOffScreen()
{
    if (((locy+getHeight()) <= 0) && (dy < 0)) {
        numRebounds++;
        if (numRebounds == MAX_BALLS_RETURNED)
            bp.gameOver();
        else
            initPosition();
    }
    else if ((locy >= getPHeight()) && (dy > 0)) {
        numRebounds--;
        if (numRebounds == 0)
            bp.gameOver();
        else
            initPosition();
    }
}
```

- Implements the [off top] and [off bottom] conditionals
Chapter 11: Sprites - The Bat Sprite

- The state chart for the bat sprite:

  - Concurrency used when have user interaction
    - Will have a chart representing user controlled aspects
    - One chart representing clock controlled aspects
Chapter 11: Sprites - The Bat Sprite (2)

• Data members

```java
public class BatSprite extends Sprite {
    private static double DURATION = 0.5;
    private static final int FLOOR_DIST = 41;
    private static final int XSTEP = 10;
    private int period;

    – FLOOR_DIST: Distance of ant’s top from the floor
    – DURATION: Animation period in seconds

• The constructor

```java
public BatSprite(int w, int h, ImagesLoader imsLd, int p) {
    super( w/2, h-FLOOR_DIST, w, h, imsLd, "leftBugs2");
    period = p;
    setStep(0,0);
}
```

– Implements the initialize state
– Starts near the center of the panel
– Initially not moving
• Movement methods

```java
public void moveLeft()
{
    setStep(-XSTEP, 0);
    setImage("leftBugs2");
    loopImage(period, DURATION);
}

public void moveRight()
{
    setStep(XSTEP, 0);
    setImage("rightBugs2");
    loopImage(period, DURATION);
}

public void stayStill()
{
    setStep(0, 0);
    stopLooping();
}

public void mouseMove(int xCoord)
{
    if (xCoord < locx)
        moveLeft();
    else if (xCoord > (locx + getWidth()))
        moveRight();
    else
        stayStill();
}
```

– They implement the corresponding transitions in the state chart
– The first three called by `processKey()` (in BugPanel) when key pressed
– `mouseMove()` called by `testPress()` when a mouse button is pressed

• `updateSprite()` method

```java
public void updateSprite()
{
    if ((locx+getWidth() <= 0) && (dx < 0))
        locx = getPWidth()-1;
    else if ((locx >= getPWidth()-1) && (dx > 0))
        locx = 1 - getWidth();
    super.updateSprite();
}
```

– Implements the `[off left]` and `[off right]` conditionals
Chapter 11: Sprites - Collision Detection

- Simplest approach is to use bounding rectangle (box) of object

- Should be as close-fitting as possible
- If boxes do not intersect, objects do not intersect

- If boxes intersect, objects may or may not intersect

* In this case, will need to perform analytic calculations of intersections
* This can be expensive
Chapter 11: Sprites - Collision Detection (2)

- To reduce the amount of analytic calculations
  
  1. Use a shape that more closely conforms to the object

  - Intersection computations with non-rectangular shapes are more expensive

  2. Use multiple rectangles

  - Requires more computations - one per box
Chapter 11: Sprites - Collision Detection - Computing Collisions

- Java’s *Rectangle* class provides methods for checking intersections
  - 1. `boolean intersects(Rectangle r)`
  - 2. `boolean contains(Rectangle r)`
  - 3. `boolean contains(Point p)`
  - 4. `boolean contains(int x, int y)`
  - 5. `boolean contains(int x, int y, int w, int h)`
  - 6. `boolean contains(Rectangle r)`

- Analytic computation for arbitrary shapes
  - For upright rectangle, checking to see if a point \((x, y)\) is within a bounding box \((x_0, y_0) \rightarrow (x_1, y_1)\) is simple:
    \[
    x_0 \leq x \leq x_1 \text{ and } y_0 \leq y \leq y_1
    \]
  - We can determine which side of a line a point \(P(x_0, y_0)\) is on using the implicit equation for the line:
    \[
    y - mx - b = 0
    \]
    * Substituting \((x_0, y_0)\) for \((x, y)\) in the above
      1. If the result == 0, \(P\) is on the line
      2. If the result > 0, \(P\) is on one side of the line
      3. If the result < 0, \(P\) is on the other side of the line
  - For arbitrary polygon intersection, we can check each vertex of one poly against ALL the edges of the second using the above
    · If a vertex is inside ALL of the second poly’s edges, the vertex is inside the polygon
    · You must check against every edge of the second polygon, since \(y - mx - b = 0\) defines infinite lines

- Embedded diagrams illustrating the concept of vertices inside and outside edges.
Chapter 11: Sprites - Collision Detection - Computing Collisions (2)

· You may need to perform the calculations for each polygon

- Performing the calculations for the vertices of the hexagon indicate no intersection
- Performing the calculations for the triangle indicate an intersection
  – Another approach is to cast rays from each vertex and determine the number of intersections that occur

  * An even number indicates the vertex is outside
  * An odd number indicates the vertex is inside

• When a curve is involved, explicit intersections must be computed