Kinect Chapter 9. Kinect Breakout

[**Note**: all the code for this chapter is available online at http://fivedots.coe.psu.ac.th/~ad/kinect/; only important fragments are described here.]

It's game time! I'll be reusing the hand tracking code of the previous chapter as the user interface for a simple version of the classic Breakout game. Kinect Breakout is shown in Figure 1.

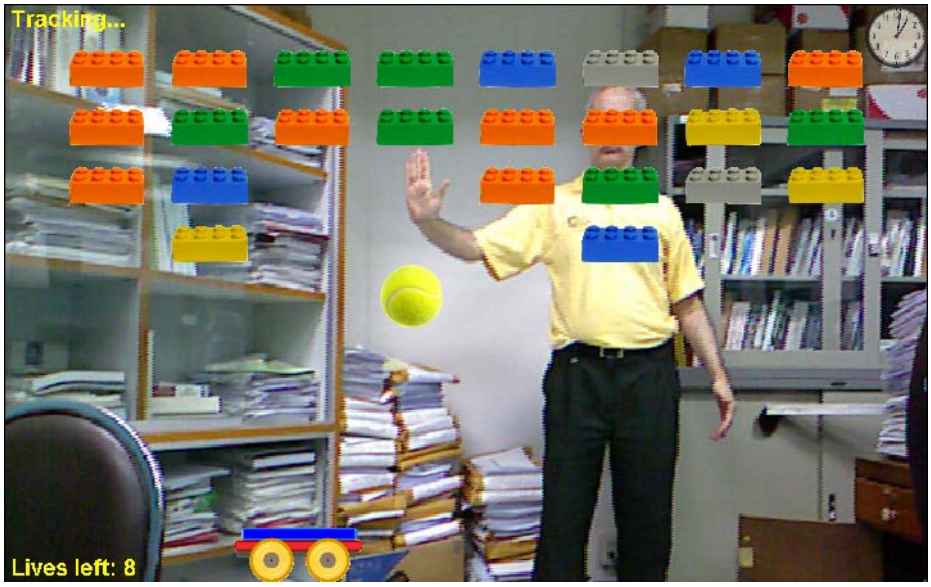

Figure 1. Kinect Breakout.

The aim is to move the wheeled carriage at the bottom of the screen to the left and right, so the tennis ball bounces off it and into the bricks. When the ball hits a brick, that brick disappears. The ball rebounds off bricks, the top of the screen, and the sides, but drops through the floor. When that happens, the user loses a 'life', and the ball reappears in the center of screen, moving downwards to the left or right. The ball's speed and direction can be adjusted by how you hit it with the carriage.

The Kinect's connection to all this is that it converts my hand movements into left and right shifts of the carriage. No mouse or keyboard input is necessary.

The game's background is the Kinect's camera display, enlarged to fill the screen. This gives me some feedback on my hand position although the carriage deliberately doesn't move in exact unison with my hand. If it did, I'd have to run backwards and forwards in my room to reach the left and right boundaries of the display area, which is too much like serious exercise.

There's a lot missing from this application to make it into a real game, but it illustrates how the Kinect can be utilized as a gaming input device.

The application, called Breakout, consists of several classes, shown in Figure 2.

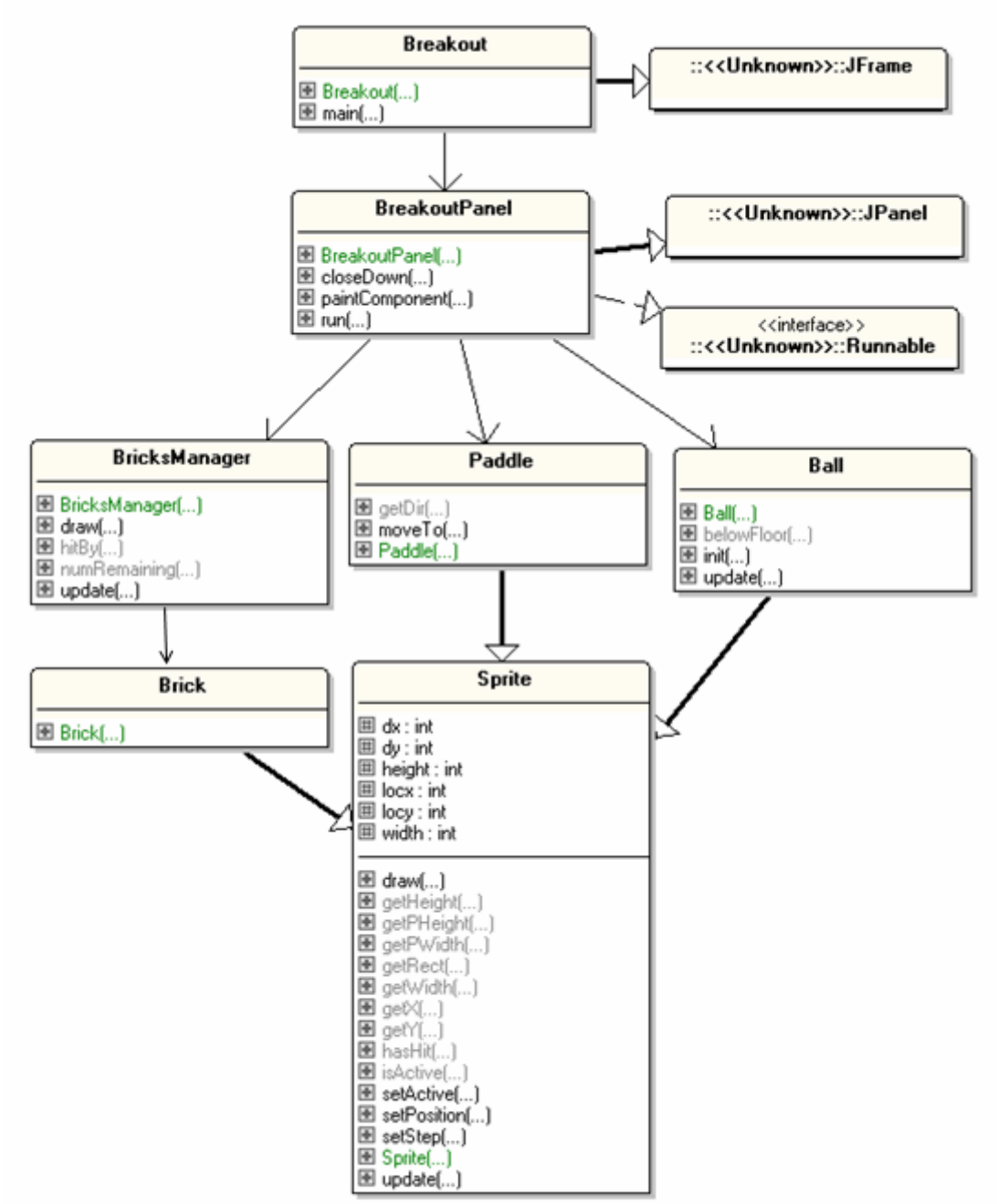

Figure 2. The Breakout Classes.

The top-level Breakout class creates an undecorated JFrame (i.e. one with no menu bar or borders), and turns the cursor invisible, but the real work is left to BreakoutPanel. It creates a panel that fills the screen, initializes the Kinect, sets up session and hand point listeners, creates the game elements, and enters a game update-draw-sleep loop.

The game elements are represented by three classes: Brick (for creating bricks), Paddle (for the carriage that the user moves), and Ball (for the ball, of course). These classes are fairly short because they inherit a great deal of functionality from the

Sprite superclass. Sprite handles updates to the position and speed of a game element, and renders it to the screen.

The BricksManager class supervises the bricks – placing them in a grid pattern initially, and controlling their redrawing as the game progresses.

1. Creating the Breakout Panel

The BreakoutPanel() constructor performs numerous initialization tasks: it sets the panel size, sets up a key listener, creates game objects (e.g. the bricks, ball, paddle), configures the Kinect, and starts a thread that executes the game update-draw-sleep loop. The code:

```
// globals
private static final int MAX_LIVES = 10;

private int pWidth, pHeight;  // panel size

// for game messages
private Font font;
private String gameStatusMsg = null;

// game elements
private Ball ball;
private Paddle paddle;
private BricksManager bricksMan;
private int livesLeft = MAX_LIVES;


public BreakoutPanel()
{
  // set panel/game dimensions to be those of the screen
  Dimension scrDim = Toolkit.getDefaultToolkit().getScreenSize();
  setPreferredSize(scrDim);
  pWidth = scrDim.width;
  pHeight = scrDim.height;

  // create game message font
  font = new Font("SansSerif", Font.BOLD, 36);

  // listen for key presses
  setFocusable(true);
  requestFocus();  // JPanel now has focus, so receives key events
  initKeyListener();

  // create game elements
  ball = new Ball(scrDim);
  paddle = new Paddle(scrDim);
  bricksMan = new BricksManager(scrDim);
  gameStatusMsg = "Lives left: " + livesLeft;

  configKinect();

  new Thread(this).start();   // start the game
}  // end of BreakoutPanel()
```


Bricks creation is handled by the BricksManager object, relieving BreakoutPanel of some tricky positional math (that I'll show you later).

Having a key listener may seem a bit useless since the user interacts with the game via the Kinect. Nevertheless, it's handy to have keyboard controls for pausing, resuming, and terminating the game. During the game's development, I also utilized a mouse listener to move the paddle:

```
public void mouseMoved(MouseEvent e)
{  paddle.moveTo(e.getX()); }
```

Once the Kinect code was fully working I removed the mouse motion listener.

1.1. Listening for Keys

initKeyListener() only looks for keys that stop, pause, or resume the game, although its functionality could obviously be extended. The game's state is affected using two global booleans, isPaused and isRunning, which I'll explain when I get to the game loop.

```
// globals
private boolean isRunning = false;   // used to stop the game loop
private boolean isPaused = true;     // pauses the game


private void initKeyListener()
// define keys for stopping, pausing, resuming the game
{
  addKeyListener( new KeyAdapter() {
    public void keyPressed(KeyEvent e)
    {
      int keyCode = e.getKeyCode();
      if ((keyCode == KeyEvent.VK_ESCAPE) ||
          (keyCode == KeyEvent.VK_Q) ||
          ((keyCode == KeyEvent.VK_C) && e.isControlDown()) )
            // ESC, q, ctrl-c to stop the game
          isRunning = false;
        else if (keyCode == KeyEvent.VK_P)
            // toggle pausing/resume with 'p'
          isPaused = !isPaused;
      }
    });
}  // end of initKeyListener()
```

ESC, 'q', or ctrl-c terminate the game, while 'p' either pauses or resumes the game.

1.2. Configuring the Kinect

configKinect() is a lengthy function that sets up the OpenNI and NITE generators and listeners. It's quite similar to the configKinect() method used by the HandsTracker example in the last chapter.

I attach four generators to the Kinect Context: an ImageGenerator for displaying the scene, a DepthGenerator for converting real-world hand coordinates into screen

values, a HandsGenerator for observing hand points, and a GestureGenerator for employing gestures to start and resume the game.

```
// globals
// OpenNI and NITE variables
private Context context;
private ImageGenerator imageGen;
private DepthGenerator depthGen;

private SessionManager sessionMan;
private SessionState sessionState;

// Kinect camera specific
private double scaleFactor = 1.0;
private int scaledWidth, scaledHeight;
private int mapWidth, mapHeight;


private void configKinect()
{
  try {
    context = new Context();

    // add the NITE License
    License license = new License("PrimeSense",
                          "0KOIk2JeIBYClPWVnMoRKn5cdY4=");
    context.addLicense(license);

    // set up image and depth generators
    imageGen = ImageGenerator.create(context);
         // for displaying the scene
    depthGen = DepthGenerator.create(context);
         // for converting real-world coords to screen coords

    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
    imageGen.setMapOutputMode(mapMode);
    depthGen.setMapOutputMode(mapMode);

    imageGen.setPixelFormat(PixelFormat.RGB24);

    ImageMetaData imageMD = imageGen.getMetaData();
    mapWidth = imageMD.getFullXRes();
    mapHeight = imageMD.getFullYRes();

    // calculate scaling for Kinect image and hand coords
    scaleFactor = ((double)pWidth)/mapWidth;
    scaledWidth = pWidth;
    scaledHeight = (int) (mapHeight * scaleFactor);

    // set Mirror mode for all
    context.setGlobalMirror(true);

    // set up hands and gesture generators
    HandsGenerator hands = HandsGenerator.create(context);
    hands.SetSmoothing(0.1f);

    GestureGenerator gesture = GestureGenerator.create(context);

    context.startGeneratingAll();
    System.out.println("Started context generating...");
```



```
    // set up session manager and points listener
    sessionMan = new SessionManager(context, "Click,Wave",
                                                "RaiseHand");
    setSessionEvents(sessionMan);
    sessionState = SessionState.NOT_IN_SESSION;

    // increase timeout from 15 to 60 secs (15000 --> 60000 ms)
    sessionMan.setQuickRefocusTimeout(60000);
    System.out.println("New Session refocus timeout: " +
                          sessionMan.getQuickRefocusTimeout() );

    PointControl pointCtrl = initPointControl();
    sessionMan.addListener(pointCtrl);
  }
  catch (GeneralException e) {
    e.printStackTrace();
    System.exit(1);
  }
}  // end of configKinect()
```

One new element in this version of configKinect() is the scaling of the camera image to fill the screen. The Kinect's camera image width (mapWidth) is divided into the panel's width (pWidth) to get a scaling factor. This is used to enlarge the Kinect image, and to scale the hand point coordinates.

I've changed the length of the Kinect's session refocus timeout. By default, 15 seconds of no hand detection is allowed before a session is terminated. I increased this to 60 seconds to give the user more time to resume a game before the session (and therefore the game) ended.

The setSessionEvents() and initPointControl() methods handle the creation of several listeners for the session and hand points, which are summarized in Figure 3.

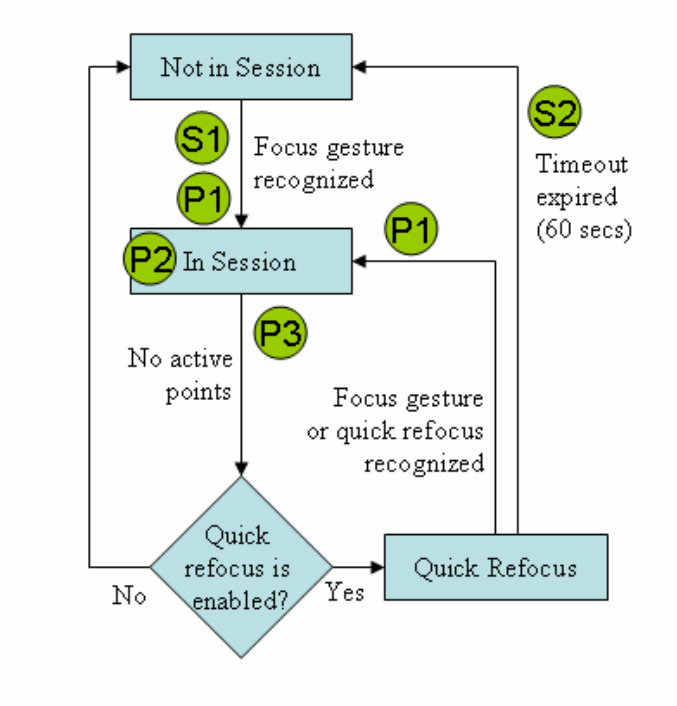

Figure 3. NITE Session States

The circles denote listeners: S1 and S2 are session listeners, while P1-P3 are for the hand points.

setSessionEvents() is unchanged from the last chapter, so I won't repeat the code here – it sets up a listener for when the session starts (state S1 in Figure 3), and a listener for when the session ends (S2 in Figure 3).

The hand point listeners created in initPointControl() are similar to those in the last chapter as well, but I don't need to monitor hand point destruction in this application:

```
// globals
private boolean isPaused = true;     // pauses the game
private SessionState sessionState;


private PointControl initPointControl()
// create 4 hand point listeners
{
  PointControl pointCtrl = null;
  try {
    pointCtrl = new PointControl();

    // create new hand point, and move paddle (P1)
    pointCtrl.getPointCreateEvent().addObserver(
                            new IObserver<HandEventArgs>() {
      public void update(IObservable<HandEventArgs> observable,
                         HandEventArgs args)
      { isPaused = false;            // start/resume the game
        movePaddle(args.getHand());  // hand moves paddle
      }
    });

    // hand point has moved; move paddle (P2)
    pointCtrl.getPointUpdateEvent().addObserver(
                            new IObserver<HandEventArgs>() {
      public void update(IObservable<HandEventArgs> observable,
                         HandEventArgs args)
      {  movePaddle(args.getHand());  }    // hand moves paddle
    });

    // no active hand point, which triggers refocusing (P3)
    pointCtrl.getNoPointsEvent().addObserver(
                             new IObserver<NullEventArgs>() {
      public void update(IObservable<NullEventArgs> observable,
                         NullEventArgs args)
      {
        if (sessionState != SessionState.NOT_IN_SESSION) {
          System.out.println("  Lost hand point, so refocusing");
          sessionState = SessionState.QUICK_REFOCUS;
          isPaused = true;    // pause the game while refocusing
          // if refocusing takes too long (> 60 secs)
          // then session will automatically end
        }
      }
    });

  }
  catch (GeneralException e) {
```



```
    e.printStackTrace();
  }
  return pointCtrl;
}  // end of initPointControl()
```

The three listeners are labeled P1-P3 in the code above, and in Figure 3.

A new hand point causes the P1 listener to start the game (or resume it), and the point's coordinate is utilized to move the paddle. As the user keeps moves their hand, the P2 listener will be invoked repeatedly, thereby adjusting the paddle position.

If HandsGenerator can't find the hand point (perhaps because the user is hiding his hand, as in Figure 4), then quick refocusing is enabled, and the game pauses.

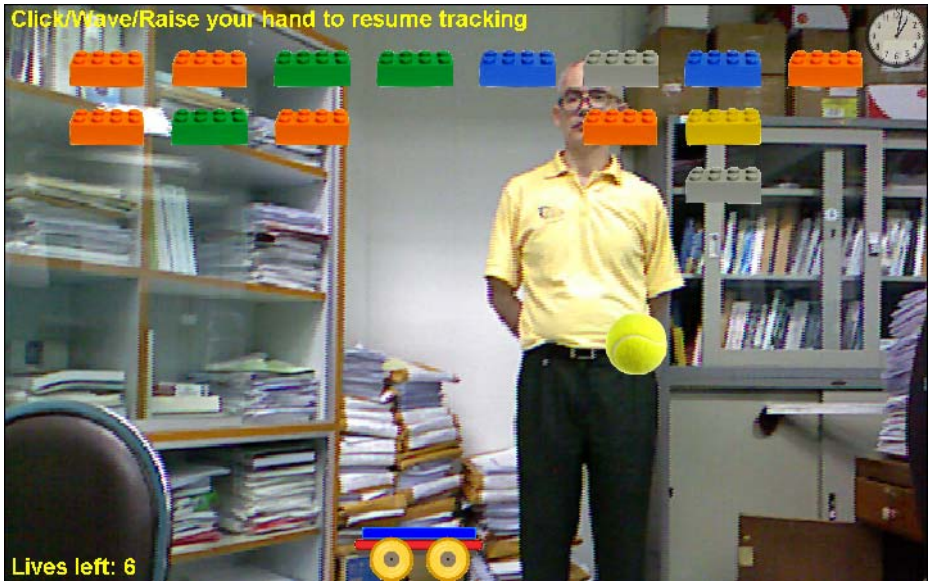

Figure 4. A Paused Game.

The session can be restarted by the user performing a quick refocus gesture ("RaiseHand") or a focus gesture ("Click" or "Wave"). If a refocus is detected before the session timeout (60 secs, set in configKinect()), then the P1 listener is fired. Pausing is switched off (i.e. isPaused is set to false) and the paddle moved.

1.3. Moving the Paddle

The paddle (the wheeled carriage that can move left and right at the bottom of the screen) is 'driven' by makePaddle(). The function is supplied with the user's hand position, and its x-coordinate is mapped into scaled screen space.

The coordinate undergoes two transformations, once from real-world space into Kinect camera image space, then scaled so it matches the enlarged image shown on the screen.

```
// globals
private SessionState sessionState;
private Paddle paddle;
private DepthGenerator depthGen;
```



```
private double scaleFactor;


private void movePaddle(HandPointContext handContext)
{
  sessionState = SessionState.IN_SESSION;
  int x = getScrX( handContext.getPosition() );
  paddle.moveTo(x);
}  // end of movePaddle()


private int getScrX(Point3D realPt)
/* convert real-world hand coordinate to Kinect camera image
   dimensions (640x480), and then scale the x-value to
   full-screen size.  */
{
  try {
    Point3D pt = depthGen.convertRealWorldToProjective(realPt);
      // convert real-world coords to Kinect camera image coords
    if (pt != null)
      return (int)(pt.getX() * scaleFactor);
             // scale x-coord to full-screen size
  }
  catch (StatusException e)
  {  System.out.println("Problem converting point"); }
  return 0;
}  // end of getScrX()
```

getScrX() performs the real-world-to-screen space transformation with the help of the DepthGenerator, then scales the x-value with scaleFactor calculated in configKinect().

1.4. The Game Loop

The BreakoutPanel thread implements a simple game loop consisting of four steps, which repeat until the game stops:

wait for Kinect updates;

update the game state;

draw the modified game state;

sleep so that each iteration of the game loop takes a fixed amount of time.

The run() method:

```
// globals
private static final int CYCLE_TIME = 25;
                              // time for one game iteration, in ms

private boolean isRunning = false;   // used to stop the game loop
private boolean isPaused = true;     // pauses the game

private Context context;
private SessionManager sessionMan;


public void run()
{
```



```
  long duration;
  isRunning = true;

  while(isRunning) {
    try {    // wait for Kinect input
      context.waitAnyUpdateAll();
      sessionMan.update(context);
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }

    long startTime = System.currentTimeMillis();
    updateCameraImage();

    if (!isPaused && !gameOver)
      updateGame();

    duration = System.currentTimeMillis() - startTime;
    repaint();

    if (duration < CYCLE_TIME) {
      try {
        Thread.sleep(CYCLE_TIME-duration);
           // wait until CYCLE_TIME time has passed
      }
      catch (Exception ex) {}
    }
  }

  // close down Kinect
  try {
    context.stopGeneratingAll();
  }
  catch (StatusException e) {}
  context.release();
  System.exit(0);
}  // end of run()
```

The waiting for the Kinect is done by calling Context.waitAnyUpdateAll(), then the Kinect camera image is updated, and the rest of game state advanced in updateGame(). The game is repainted by run() making a repaint() request.

The execution time for an iteration (apart from the repainting) is recorded in the duration variable, which is used to calculate the sleep time that each cycle takes to make an iteration occupy CYCLE_TIME (25) milliseconds. This makes the application run with a frame rate of about 1000/25 == 40 frames/second, which is reasonably speedy.

1.5. Updating the Game

Breakout's game state is quite simple because only the ball and paddle move, and a brick can only turn invisible (when hit by the ball). These state changes are performed by updateGame():

```
// globals
```



```
private boolean gameOver = false;   //has user finished the game?
private Ball ball;
private Paddle paddle;
private BricksManager bricksMan;


private void updateGame()
{
  gameOver = isAtGameEnd();
  if (!gameOver) {
    ball.update(paddle, bricksMan);
    paddle.update();
  }
  else
    ball.setActive(false);   // make ball invisible
}  // end of updateGame()
```

The possibility that a brick may become invisible is tested inside Ball.update(), so a BricksManager reference is passed to the Ball object.

When the game is over, the ball is turned invisible because the game seems to look better to have all the bricks *and* the ball disappear at the end.

isAtGameEnd() performs several tests to determine if the user has won or lost. A player can win if he removes all the bricks by hitting them with the ball, and can lose if his number of lives drops to 0. The only way a user can lose a life is if the ball drops below the 'floor' (the lower edge of the screen)

```
// globals
private String gameStatusMsg = null;
private int livesLeft;
private Ball ball;
private BricksManager bricksMan;


private boolean isAtGameEnd()
{
  if (ball.belowFloor()) {     // means user has lost a life
    livesLeft--;
    if (livesLeft == 0) {
      gameStatusMsg = "All lives lost";
      return true;    // game is over
    }
    else {
      gameStatusMsg = "Lives left: " + livesLeft;
      ball.init();     // reposition ball back on screen
    }
  }

  if (bricksMan.numRemaining() == 0) {   // all bricks deleted
    gameStatusMsg = "Victory!";
    return true;
  }

   return false;
}  // end of isAtGameEnd()
```


The global gameStatusMsg string is drawn in the bottom left corner of the screen at render time to give the user some information about the game's status. For example, Figure 5 shows gameStatusMsg reporting the number of lives remaining.

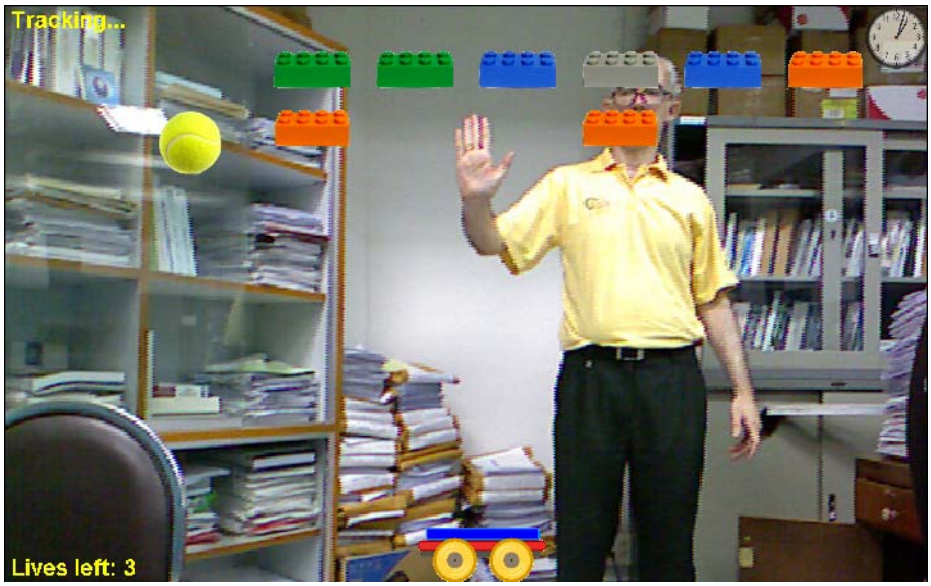

Figure 5. A User With Only Three Lives.

1.6. Painting the Game

BreakoutPanel.run() makes a paint request by calling repaint(), which eventually calls the panel's paintComponent() method.

The game visuals are drawn so that foreground images appear after background ones. The drawing order is:

first, the enlarged Kinect camera image;

the game elements: ball, paddle, and the visible bricks;

Kinect session state information (at the top-left corner)

game status information (bottom left corner);

a game over message in the middle of the screen (if the game is over).

The paintComponent() method:

```
// globals
private boolean gameOver;
private int pWidth, pHeight;         // panel size

// game elements
private Ball ball;
private Paddle paddle;
private BricksManager bricksMan;
private String gameStatusMsg;

// Kinect camera specific
private BufferedImage camImage = null;
private int scaledWidth, scaledHeight;
```



```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);

  // fill the background with black
  g.setColor(Color.BLACK);
  g.fillRect(0, 0, pWidth, pHeight);

  // draw scaled camera image as background
  if (camImage != null)
    g.drawImage(camImage, 0, 0, scaledWidth, scaledHeight, null);

  // draw game elements
  ball.draw(g);
  paddle.draw(g);
  bricksMan.draw(g);

  // draw status messages
  g.setFont(font);
  g.setColor(Color.YELLOW);

  drawSessionInfo(g);
  if (gameStatusMsg != null)
    g.drawString(gameStatusMsg, 10, pHeight-10);

  if (gameOver)
    showGameOver(g);
} // end of paintComponent()
```

Much of the painting work is delegated to the paddle, ball, and bricks manager, by calling their draw() methods.

2. A Sprite

The bricks, ball, and paddle share a lot of functionality, which is placed in a Sprite superclass. For instance, every sprite has a position, a velocity (in terms of step increments in the x- and y- directions), an image for rendering, and can be made invisible (deactivated). Common sprite behaviors involves changing a sprite's position and adjusting its step increments (to make it move faster or slower, or change direction). Two sprites can be tested to see if they have 'collided', a useful feature when the ball hits the paddle or a brick.

The Sprite class I describe here is quite basic, and could certainly possible be made more powerful. For example, it could display different images depending on a sprite's internal state, or even short animations. The class could permit sound clips to be connected to important states, such as collisions. A version of Sprite which does these things can be found in chapter 11 of my book *Killer Game Programming in Java*, or in draft version online at http://fivedots.coe.psu.ac.th/~ad/jg/ch06/.

2.1. The Sprite Constructor

A sprite is initialized with its position, the size of the enclosing panel, and the name of its image:

```
//globals
// default step sizes (how far to move in each update)
private static final int STEP = 3;

private int pWidth, pHeight;    // panel size

// protected vars
protected int locx, locy;        // location of sprite
protected int dx, dy;            // step to move in each update
protected int width, height;     // sprite dimensions


public Sprite(String fnm, Dimension scrDim)
{
  pWidth = scrDim.width;     // panel size
  pHeight = scrDim.height;

  loadImage(fnm);    // load image and set width/height

  // start in center of panel by default
  locx = (pWidth-width)/2;
  locy = (pHeight-height)/2;

  dx = STEP; dy = STEP;
} // end of Sprite()
```

The sprite's coordinate (locx, locy) and its step values (dx, dy) are stored as integers. This simplifies certain tests and calculations, but restricts positional and speed precision. For instance, the ball can't move 0.5 pixels at a time.

The locx, locy, dx, and dy variables are protected rather than private due to their widespread use in Sprite subclasses. They also have getter and setter methods, so can be accessed and changed by objects outside the Sprite hierarchy.

Sprite only stores (x, y) coordinates—there's no z-coordinate; such functionality is unnecessary in Breakout. Simple z-axis ordering is achieved by ordering the calls to draw() in BreakoutPanel.paintComponent().

loadImage () loads a specified image, and records its dimensions:

```
// globals
private static final int SIZE = 8;  // default sprite size
private static final String IM_DIR = "images/";

private Image image =  null;
protected int width, height;     // sprite dimensions


private void loadImage(String fnm)
{
  image = new ImageIcon(IM_DIR + fnm).getImage();
  if (image != null) {
    width = image.getWidth(null);
    height = image.getHeight(null);
```



```
  }
  else {
    System.out.println("Could not load image from " + fnm);
    width = SIZE;  height = SIZE;
  }
}  // end of loadImage()
```

If the image can't be found, then default dimensions are assigned to the sprite.

2.2. Collision Detection

The Sprite class offers a simple form of collision detection based on comparing the bounding boxes of two sprites to see if they overlap. The bounding box is available through getRect ():

```
// globals
protected int locx, locy;        // location of sprite
protected int width, height;     // sprite dimensions

public Rectangle getRect()
{  return  new Rectangle(locx, locy, width, height);  }
```

The collision detection method, hasHit(), utilizes getRect() to compare two sprites:

```
public boolean hasHit(Sprite sprite)
/* this sprite has hit the other one if their
   bounding rectangles intersect  */
{
  Rectangle thisRect = getRect();
  Rectangle spriteRect = sprite.getRect();
  if (thisRect.intersects(spriteRect))
    return true;
  return false;
}  // end of hasHit()
```

2.3. Updating a Sprite

A sprite is updated by adding its step values (dx, dy) to its current location (locx, locy):

```
// global
protected int locx, locy;        // location of sprite
protected int dx, dy;            // step to move in each update
private boolean isActive = true;
               // a sprite is updated and drawn only when active


public void update()
{ if (isActive()) {
    locx += dx;
    locy += dy;
  }
}
```


The isActive boolean allows a sprite to be (temporarily) removed from the game since the sprite won't be updated or drawn when isActive is false. There are public isActive() and setActive() methods for manipulating the boolean.

No attempt is made in update() to test for collisions with other sprites, obstacles, or the edges of the gaming pane. These must be added by the subclasses when they override update().

BreakoutPanel.run()'s animation loop tries to maintain a fixed frame rate so that the sprites' update() methods are called close to 40 times per second. This allows me to calculate a sprite's speed. For instance, if its x-axis step value (dx) is 10, then the sprite will be moved 10 pixels in each update. This corresponds to a speed of 10*40 = 400 pixels/second along that axis.

2.4. Drawing a Sprite

The panel's paintComponent() method calls the sprite's draw() method to repaint it:

```
public void draw(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
      g.drawImage(image, locx, locy, null);
  }
} // end of draw()
```

If the sprite's image is null, then its default appearance is a small yellow circle.

3. The Paddle

The paddle is moved left and right along the bottom of the screen, controlled by left and right swipes of the user's hand.

The mapping from hand position to paddle position isn't exact since this would require me to move my hand large distances. Instead a restricted x-axis hand range around the middle of the screen is converted into a range spanning the entire display, as illustrated by Figure 6.

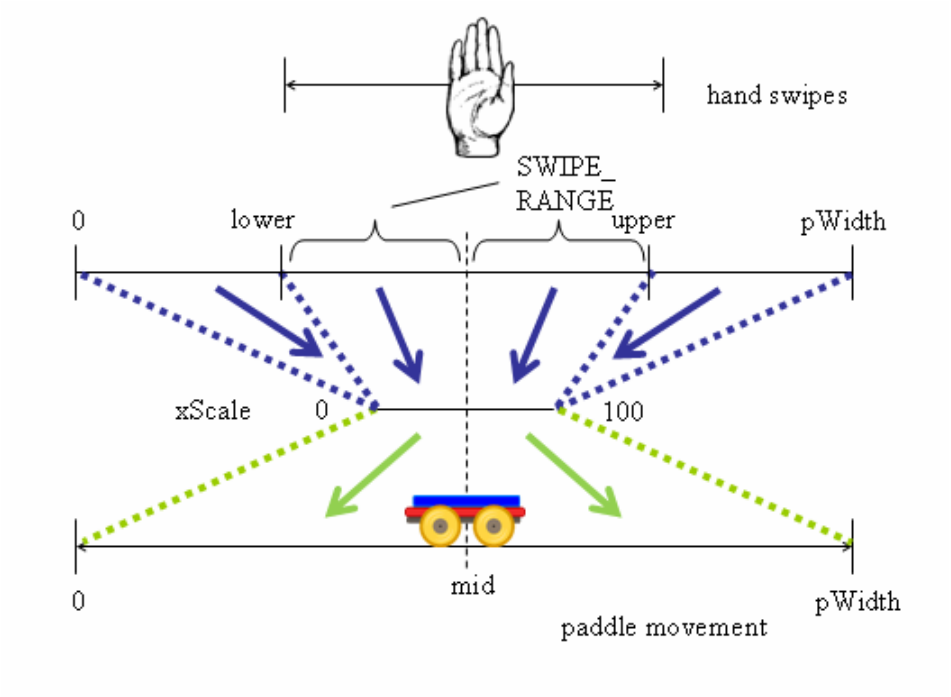

Figure 6. From Hand Swipes to Paddle Movement.

Hand positions around the mid point (+/- SWIPE_RANGE) are scaled to between 0 and 100. Positions to the left of the lower limit are mapped to 0, while those above the upper limit go to 100. This range is then scaled to span the width of the screen.

The Paddle() constructor calculates the lower, mid, and upper values, and also initializes the paddle's starting position.

```
// globals
private static final int SWIPE_RANGE = 200;

private int mid, upperLimit, lowerLimit;


public Paddle(Dimension scrDim)
{
  super("carriage.png", scrDim);

  setPosition( (getPWidth()-width)/2, getPHeight()-height);
    // positioned at the bottom of the panel, at the center

  setStep(0,0);  // no movement

  mid = getPWidth()/2;     // the middle of the panel

  // calculate lower-to-upper limits
  lowerLimit = mid - SWIPE_RANGE;
  if (lowerLimit < 0)
    lowerLimit = 0;

  upperLimit = mid + SWIPE_RANGE;
  if (upperLimit > getPWidth())
    upperLimit = getPWidth();
```



```
} // end of Paddle()
```

The paddle's x- and y- axis steps are set to 0 since all the paddle's movement is supplied by the user's hand position.

3.1. Moving the Paddle

The hand position listeners in BreakoutPanel call Paddle.moveTo() to move the paddle, and pass it the hand's x-axis screen coordinate:

```
// global
private int xPrev = -1;     // previous paddle position


public void moveTo(int x)
{
  if ((x >= 0) && (x+width < getPWidth())) {
    xPrev = locx;
    locx = scaleX(x);    // new x-axis location of paddle
  }
}  // end of moveTo()
```

scaleX() executes the two-stage transformation shown in Figure 6:

```
private int scaleX(int x)
{
  // truncate lower-upper so xScale is in the range 0-100
  int xScale = 0;
  if ((lowerLimit < x) && (x < upperLimit))
    xScale = ((x - mid) * 100/(2 * SWIPE_RANGE)) + 50;
  else if (x >= upperLimit)
    xScale = 100;

  return (xScale * (getPWidth()-width)/100);
}  // end of scaleX()
```

The position in the 0-100 range is stored in the local xScale variable, then scaled to span the screen's width.

Since the paddle should stay fully visible on the right side of the screen, the scale factor uses (pWidth – width). This ensures that the paddle can only move as far right as (pWidth – width).

3.2. Using the Paddle's Direction

The ball's velocity is affected by the paddle's direction of movement when the ball rebounds off the paddle. Its current direction can be accessed via the public Paddle.getDir() method which returns a value from the PaddleDir enumeration:

```
enum PaddleDir {     // direction of paddle movement
    LEFT, RIGHT, NONE
}
```


getDir() calculates the direction by comparing the paddle's current location (in locx) with its previous location (in xPrev):

```
// global
private int xPrev = -1;     // previous paddle position


public PaddleDir getDir()
{
  if ((xPrev == -1) || (xPrev == locx))
    return PaddleDir.NONE;
  else if (xPrev < locx)
    return PaddleDir.RIGHT;
  else
    return PaddleDir.LEFT;
}  // end of getDir()
```

3. Managing the Bricks

The BricksManager class creates the bricks, randomly assigning them differently colored Lego images. The bricks are evenly spaced out across the screen in NUM_ROWS rows.

The BricksManager constructor:

```
// globals
private static final int NUM_ROWS = 4;

private static final String IM_DIR = "images/";

private static final String[] LEGO_FNMS = {   // lego brick images
     "orangeBrick.png", "redBrick.png", "yellowBrick.png",
     "blueBrick.png",  "grayBrick.png", "greenBrick.png" };

private Brick[] bricks;
private int numRemaining;
       // the no of active bricks (i.e. ones that are visible)


public BricksManager(Dimension scrDim)
{
  Dimension brickDim = getBrickSize(LEGO_FNMS[0]);
  if (brickDim == null) {
    System.out.println("No brick found in " + LEGO_FNMS[0]);
    System.exit(1);
  }
  int brickWidth = brickDim.width;
  int brickHeight = brickDim.height;

  int xDist = brickWidth*4/3;
         // x-axis dist from start of one brick to start of next

  int numCols = (scrDim.width/xDist)-1;   // number of columns

  int xGap = (scrDim.width - (numCols * xDist) + brickWidth/3)/2;
        // gap between bricks, so columns are
        // equally spaced out from the screen sides
```



```
  // store NUM_ROWS*numCols bricks in an array
  bricks = new Brick[NUM_ROWS*numCols];

  numRemaining = NUM_ROWS*numCols;

  Random rand = new Random();   //for randomly choosing image fnm
  int count = 0;
  for (int i = 0; i < NUM_ROWS; i++)
    for (int j = 0; j < numCols; j++) {
      int idx = rand.nextInt(LEGO_FNMS.length);
      bricks[count] = new Brick(LEGO_FNMS[idx], xGap + j*xDist,
                                70 + i*(brickHeight*3/2), scrDim);
      count++;
    }
} // end of BricksManager()
```

The Lego image filenames are randomly assigned to each Brick object as it is created.

The size of a brick is obtained by calling getBrickSize(), which loads one of the Lego images (all the images are the same size):

```
// global
private static final String IM_DIR = "images/";


private Dimension getBrickSize(String fnm)
{
  try {
    BufferedImage im = ImageIO.read( new File(IM_DIR + fnm));
    return new Dimension(im.getWidth(), im.getHeight());
  }
  catch (IOException e)
  {  return null;  }
}  // end of getBrickSize()
```

3.1. Collision Detection

BricksManager handles collision detection between the ball and the bricks by passing the ball's reference to each of them. The first brick that reports that it has been hit is deactivated, and so becomes invisible.

```
// global
private Brick[] bricks;


public Brick hitBy(Ball ball)
{
  for(Brick b : bricks)
    if (b.isActive() && ball.hasHit(b)) {
      b.setActive(false);    // brick will no longer be drawn
      numRemaining--;
      return b;    // return reference to brick that was hit
    }
  return null;
}  // end of hitBy()
```


3.2. Drawing the Bricks

BricksManager draws the bricks by calling Brick.draw() on each one:

```
public void draw(Graphics g)
{ for(Brick b : bricks)
    b.draw(g);
}
```

3.3. Defining a Brick

The Brick class is extremely short, inheriting all of its functionality from Sprite.

```
public class Brick extends Sprite
{
  public Brick(String fnm, int x, int y, Dimension scrDim)
  {
    super(fnm, scrDim);
    setPosition(x, y);
    setStep(0, 0);    // not moving
  }
}
```

4. The Ball

The ball starts moving from the center of the screen, bouncing off the walls, ceilings, paddle, and bricks. When it hits a brick, the brick disappears. If the ball goes below floor level, then the user loses a life, and the ball is reinitialized, starting back at the screen's center.

When the ball hits the paddle, the ball's velocity increases if the paddle direction matches the ball's direction. Alternatively, it slows down if the ball and paddle are moving in opposite directions.

The ball's initialization is split between a constructor and a public init() method, which is called whenever the ball needs to be restarted after dropping off the screen.

```
// globals
private static final int STEP = 15;

private Random rand;


public Ball(Dimension scrDim)
{
  super("tennis.png", scrDim);
  rand = new Random();
  init();
}


public void init()
{
  setPosition((getPWidth()-width)/2, getPHeight()/2);
```



```
                              // always start near screen center

  int xStep = (rand.nextBoolean()) ? STEP : -STEP;
  setStep(xStep, STEP);     // move down, but to left or right
}  // end of init()
```

The ball always starts at the center of the screen (actually a little below the center), and moves downwards. However, it may head either to the left or right depending on the sign of the x-axis step value.

4.1. Updating the Ball

The ball's update will adjust its step values if it has bounced off a wall or the ceiling, rebounded off the paddle, or hit a brick. In the case of hitting the paddle, the ball's speed will also vary depending on the paddle's direction of movement.

```
// global
private static final int INCR = 10;
            // for speeding up/slowing down the sprite


public void update(Paddle paddle, BricksManager bricksMan)
{
  bounceOffWalls();

  // if the ball has hit the paddle, make it bounce up
  if (hasHit(paddle)) {
    dy = -dy;
    locy += 5*dy;     // move away from paddle,
                      // so won't hit on next frame
    // adjust speed of ball based on paddle's dir of movement
    PaddleDir dir = paddle.getDir();
    if (dir == PaddleDir.LEFT)
      dx =- INCR;
    else if (dir == PaddleDir.RIGHT)
      dx += INCR;
  }

  // rebound off a brick that's been hit
  Brick brick = bricksMan.hitBy(this);
  if (brick != null)
    reboundFrom(brick);

  super.update();
} // end of update()
```

Rebounding typically involves reversing the sign of a step variable (dx or dy). This is determined for a wall or the ceiling by looking at the position of the ball in bounceOffWalls():

```
private void bounceOffWalls()
// Rebound when the ball hits a wall or ceiling.
{
  if ((locy <= 0) && (dy < 0))    // touching top and moving up
    dy = -dy;
```



```
  if ((locx <= 0) && (dx < 0))    // touching lhs and moving left
    dx = -dx;   // move right
  else if ((locx+width >= getPWidth()) && (dx > 0))
    // touching rhs and moving right
    dx = -dx;   // move left
}  // end of bounceOffWalls()
```

Rebounding from a brick requires collision detection between the ball and the brick. Rather than use intersection, four 'edge points' just outside the ball area are tested for containment by reboundFrom() in brick's bounding rectangle:

```
private void reboundFrom(Brick brick)
// make the ball bounce away from the brick
{
  Rectangle brickRect = brick.getRect();   // bounding box

  // calculated edge points of ball
  Point ptRight = new Point(locx + width+1, locy);
  Point ptLeft = new Point(locx-1, locy);
  Point ptTop = new Point(locx, locy-1);
  Point ptBottom = new Point(locx, locy + height+1);

  // test edge points against brick's bounded box
  if (brickRect.contains(ptRight) ||
      brickRect.contains(ptLeft) )
    dx = -dx;     // rebound from left or right edges

  if (brickRect.contains(ptTop)  ||
      brickRect.contains(ptBottom) )
    dy = -dy;     // rebound from top or bottom edges
} // end of reboundFrom()
```

