Kinect Chapter 2.4. Kinect Snow

In the previous chapter I showed how it was possible to use the OpenNI generator nodes for imaging, depth data, and user IDs to replace the background of the camera picture. The end result looks a bit like the blue screening used on TV, but the quality is poorer, with jagged edges around the user's outline.

If changing the background was all this Kinect technique offers then there would be little point to preferring it over blue screening, which can be implemented without depth and user ID processing (although you need a colored backcloth). However, the extra information supplied by the Kinect allows me to augment the visuals in interest ways, which I'll discuss in this chapter and the next (chapter 2.5).

This chapter looks at having the user interact with the 'virtual' scene. Often this requires skeletal information from the Kinect (e.g. the position of the user's hands or head). I'll start utilizing skeletons in chapter 4, but there's plenty of interaction forms that don't need that sort of detail.

The KinectSnow application places the user on a country road in a snowstorm (see Figure 1).

Figure 1. The User in a Snow Storm.

The falling snow gradually piles up on top of the person, until he moves. As the screenshots in Figure 2 show, the heaped snow briefly retains the outline of the user's old position, and then starts dropping again.

Figure 2. The User Moves.

1. An Overview of the KinectSnow Application

KinectSnow builds upon the features of ChangeBG from chapter 2.3, as indicated by the class diagrams for the application in Figure 3.

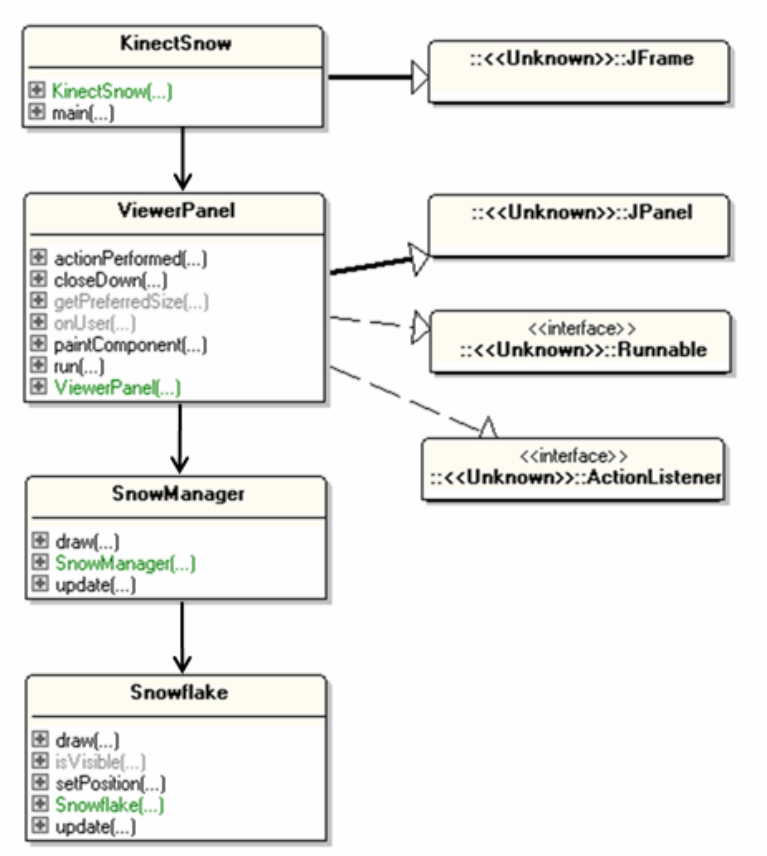

Figure 3. Class Diagrams for the KinectSnow Application.

The KinectSnow class implements the JFrame, which renders the scene in a panel created with ViewerPanel. ViewerPanel performs the same operations as the same-

named class in the ChangeBG application, with additional code for making the snow fall.

The updating and rendering of the snowstorm are handled by a SnowManager object, which represents each snowflake by a Snowflake object.

2. Hiding the Background

A large part of ViewerPanel's work involves subtracting the background from the Kinect camera image, leaving only the users visible. It does this in exactly the same way as in the last chapter, so I won't repeat all the details again. Before continuing, you should read chapter 2.3 (if you haven't already done so).

The current camera image is modified repeatedly by the ViewerPanel's run() method:

```
// globals
private volatile boolean isRunning;
private Context context;

// for snow animation
private SnowManager snowMan;
private volatile boolean moveSnow;


public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAndUpdateAll();
          // wait for all nodes to have new data, then updates them
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    screenUsers();

    if (moveSnow) {    // time to animate the snow
      moveSnow = false;
      snowMan.update();
    }

    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }

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


This code is almost identical to the run() method in the last chapter, with a few small additions that I'll explain below. run() calls screenUsers() to remove the background from the camera image:

```
// globals
private ImageGenerator imageGen;
private int[] cameraPixels;
private BufferedImage cameraImage;


private void screenUsers()
{
  // store the Kinect RGB image as a pixel array in cameraPixels
  try {
    ByteBuffer imageBB = imageGen.getImageMap().createByteBuffer();
    convertToPixels(imageBB, cameraPixels);
  }
  catch (GeneralException e) {
    System.out.println(e);
  }

  hideBackground(cameraPixels);

  // change the modified pixels into an image
  cameraImage.setRGB( 0, 0, pWidth, pHeight, cameraPixels,
                                                  0, pWidth );
  imageCount++;
}  // end of screenUsers()
```

The camera image is maintained in two globals – the cameraPixels byte array and the cameraImage BufferedImage. The hideBackground() method is unchanged from before, assigning all the background pixels a transparent blue color stored in the hideBGPixel integer:

```
private int hideBGPixel;
   /* the "hide the background" pixel: this could be any color
      so long as its alpha value is 0 */

// in the ViewerPanel constructor
hideBGPixel =  new Color(0, 0, 255, 0).getRGB();
                                        // transparent blue
```

Rendering the scene involves drawing the snowy road, then the partly transparent cameraImage, and the falling snow:

```
// globals
private BufferedImage backIm, cameraImage;
      // the background image and final camera image
private SnowManager snowMan;


public void paintComponent(Graphics g)
// draw background, the camera image, and the snow
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;
```



```
  g2.drawImage(backIm, 0, 0, this);
  g2.drawImage(cameraImage, 0, 0, this);

  snowMan.draw(g2);

  writeStats(g2);
} // end of paintComponent()
```

The new element is the call to SnowManager.draw().

3. Adding Snow Fall

The simplest way to animate the snow is inside the loop in run(), but there's a slight hitch. The loop executes as quickly as possible, waiting only long enough for the OpenNI generator nodes to be refreshed. In practice this means that each iteration takes about 30-50 ms (I know this from the statistics printed at the bottom left of the panel). This is too fast an update rate for snow that's meant to be drifting downwards.

My solution is to use a timer and a boolean flag to animate the snow at a slower rate. The timer and flag are set up in the ViewerPanel constructor:

```
// globals
private javax.swing.Timer animatorTimer;
private volatile boolean moveSnow;
     // used to flag that it's time for a snow update


// in the ViewerPanel constructor
     :
// create a timer for animating the falling snow
moveSnow = false;
animatorTimer = new javax.swing.Timer(75, this);    // refresh rate
animatorTimer.setInitialDelay(500);       // wait time before start
animatorTimer.setCoalesce(true);
animatorTimer.start();                    // start the timer
```

The timer fires roughly every 75ms, triggering a call to the actionPerformed() method defined in ViewerPanel:

```
public void actionPerformed(ActionEvent e)
{  moveSnow = true;  }
```

The moveSnow variable is now set to true, which affects the execution of the next iteration of the run()'s loop:

```
// inside run()
  :
while (isRunning) {
 // some code not shown...

  if (moveSnow) {    // time to animate the snow
```



```
    moveSnow = false;
    snowMan.update();
  }

  // some code not shown...
  }
  :
```

Inside the if-block, the moveSnow flag is toggled back to false, and the SnowManager object updates the snow.

This approach means that although the run() loop executes very speedily, the updating of the snow occurs less frequently.

4. User Collision Detection

An important feature of a Snowflake object is its ability to detect a collision with the top of a user, and stop moving. This relies on a ViewerPanel.onUser() method which tests if a given (x, y) coordinate is a pixel that's used to draw a user:

```
// globals
private int[] cameraPixels;
               // holds the pixels that fill the cameraImage image
private int hideBGPixel;        //  the "hide the background" pixel
private int pWidth, pHeight;    // of Kinect panel


public boolean onUser(int x, int y)
// is the pixel at (x,y) used to draw a user?
{
  if ((x < 0) || (x >= pWidth) ||
      (y < 0) || (y >= pHeight))
    return false;

  int pixel = cameraPixels[(pWidth*y) + x];
  return (pixel != hideBGPixel);
}  // end of onUser()
```

The trick is to convert the (x, y) coordinate into an index into the cameraPixels[] array, and then test if that location contains the transparent color. If it doesn't then the pixel must be part of the user image, and so the method returns true.

The coding of onUser() assumes that the rendering panel is a fixed size whose width and height are stored in the pWidth and pHeight globals.

5. Managing the Snow

The SnowManager object maintains an array of Snowflake objects, which it updates and draws. The array is initialized in the SnowManager constructor:

```
// globals
```



```
private static final int NUM_FLAKES = 3000;
                                  // total number of flakes
private Snowflake[] snow;
private Random rand;


public SnowManager(int pWidth, int pHeight, ViewerPanel vp)
{
  rand = new Random();
  snow = new Snowflake[NUM_FLAKES];   // create all snow flakes
  for(int i=0; i < NUM_FLAKES; i++)
    snow[i] = new Snowflake(rand, pWidth, pHeight, vp);
}  // end of SnowManager()
```

Updating the snow simply involves calling the update() method for each Snowflake object in the array. However, it's complicated by a need to make the snow fall look random, and to reuse Snowflake objects which have dropped below the bottom edge of the panel. These complications are dealt with by the startSomeFlakes() method at the beginning of update():

```
public void update()
{
  startSomeFlakes();
  for(int i=0; i < NUM_FLAKES; i++)
    snow[i].update();
}  // end of update()
```

Initially every Snowflake() object is positioned above the top-edge of the panel, and so is invisible. Similarly, any flakes that have dropped off the bottom of the panel will also be invisible. All the flakes' visibility are checked by startSomeFlakes(), and a randomly selected group are made visible at the top of the panel.

```
// globals
private static final int NUM_FLAKES = 3000;  //total no. of flakes
private static final int START_BATCH = 20;
                                   // number of flakes in a batch
private Snowflake[] snow;
private Random rand;


private void startSomeFlakes()
{
  int numStarted = 0;
  int i = 0;
  while ((numStarted < START_BATCH) && (i < NUM_FLAKES)) {
    if (!snow[i].isVisible()) {     // if flake invisible
      if (rand.nextBoolean() == true) {   // maybe
         snow[i].setPosition();
             // position at top of panel (so becomes visible)
         numStarted++;
      }
    }
    i++;
  }
}  // end of startSomeFlakes()
```


The call to Snowflake.setPosition() gives a flake a visible position in the panel. Now that it's visible, the flake will start moving downwards when Snowflake.update() is called back in SnowManager.update().

SnowManager draws all the snowflakes by calling Snowflake.draw() for each one. A flake will only be rendered if it is visible in the panel.

```
public void draw(Graphics2D g2)
{ for(int i=0; i < NUM_FLAKES; i++)
    snow[i].draw(g2);
}
```

6. Creating a Snowflake

A Snowflake object maintains a (x, y) coordinate for a flake. If this lies off the top or bottom edges of the panel then the flake is deemed invisible, and so neither updated or drawn. However, if it is visible on the panel's surface, then Snowflake.update() will move it down and Snowflake.draw() will render it as a small white circle.

The unusual aspect of update() is that it must determine if the flake is touching the top of a user and then stop it from moving.

```
// globals
private static final int Y_DROP = 10;     // max down step
private static final int X_DRIFT = 6;     // max side-to-side step

private Random rand;
private int x, y;   // position of flake


public void update()
{
  if (isVisible()) {
    if (isOnTopEdgeOfUser())
      return;   // do not update (i.e. stop moving)
    else {
      x += rand.nextInt(X_DRIFT) - X_DRIFT/2;
                                  // small sideways movement
      y += rand.nextInt(Y_DROP);                 // move down
    }
  }
}  // end of update()
```

The random number-based calculations in update() modify the flake's (x, y) coordinate so it moves downwards with a small drift to the left or right. However, if the flake is invisible (not on the panel) or touching the top-edge of a user, then the update is skipped, and the flake will not move.

Note, that if the user moves, the top-edge touching condition may no longer be true when the flake is next updated, and so it will resume falling.

The isVisible() method uses the fact that the panel is a fixed size to determine if the flake is visible or not:


```
public boolean isVisible()
// a flake is visible if within the panel's boundaries
{
  if ((x < 0) || (x >= pWidth) ||
      (y < 0) || (y >= pHeight))
    return false;
  return true;
}  // end of isVisible()
```

isOnTopEdgeOfUser() utilizes a simple test based on calling ViewerPanel.onUser() twice. The current location of the flake is tested, and a location a small distance higher up the panel (see Figure 4).

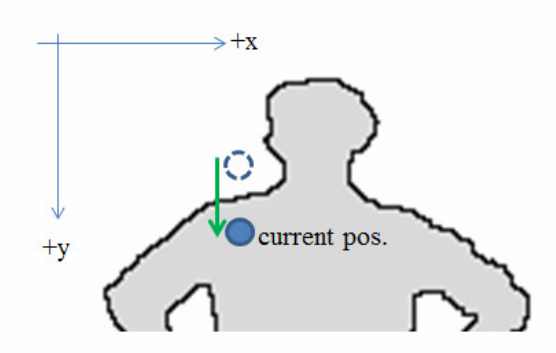

Figure 4. Testing for the Top-edge Condition.

A flake is deemed to be touching the top-edge of the user if it is currently on a user but the coordinate a short distance higher (the dotted circle in Figure 4) is *not* on the user:

```
// globals
private static final int SIZE = 6;   // of the flake

private int x, y;   // position of flake
private ViewerPanel vp;


private boolean isOnTopEdgeOfUser()
// test if the flake is on the top edge of a user
{
  if (vp.onUser(x,y) && !vp.onUser(x,y-SIZE*2))
     return true;
  return false;
}  // end of isOnTopEdgeOfUser()
```

The SIZE constant is also used when rendering the flake, and corresponds to the flake's diameter.

The isOnTopEdgeOfUser() test is simple and fast to carry out, but may also fail. For example, if the flake is moving very quickly, then it may travel too far into the user's space in a single update. Then the "short distance higher" offset used by isOnTopEdgeOfUser() will be on the user, and so the edge test will fail In fact, this

does occasionally happen, with a few flakes passing over the user without stopping. However, this looks like snow dropping in front of the user, so is easy to ignore.

