Kinect Chapter 2.3. Changing the Background

In this chapter, I get a chance to go outside, without leaving the safety of my cubicle (see Figure 1).

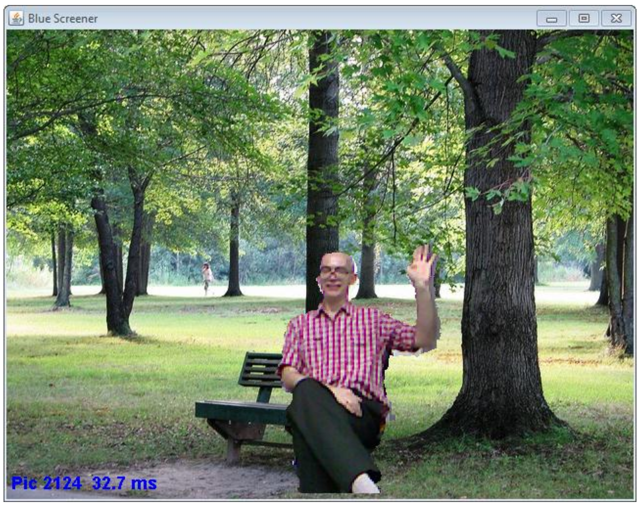

Figure 1. Sitting on a Park Bench, or am I?

I use Kinect to identify the background of my work area, and replace it with a picture of a park. This might remind you of *blue screen compositing*, a standard element of TV weather forecasting, but I'm *not* using that color-based approach. Instead I'm utilizing the Kinect's ability to identify body shapes to decide which pixels to make transparent. Bodies are drawn unchanged, but all the other pixels in the image are made transparent, allowing a static background image (e.g. of a park) to be seen.

The process involves three OpenNI generator nodes – one for the camera image, another for the depth map, and a user generator node. The user generator returns a labeled user map, where each pixel holds a user ID (1, 2, etc.), or 0 to mean it's part of the background. My code uses '0' positions in the user map to set the corresponding pixel positions in the image map to be transparent. The main steps, which are explained in more detail below, are summarized in Figure 2.

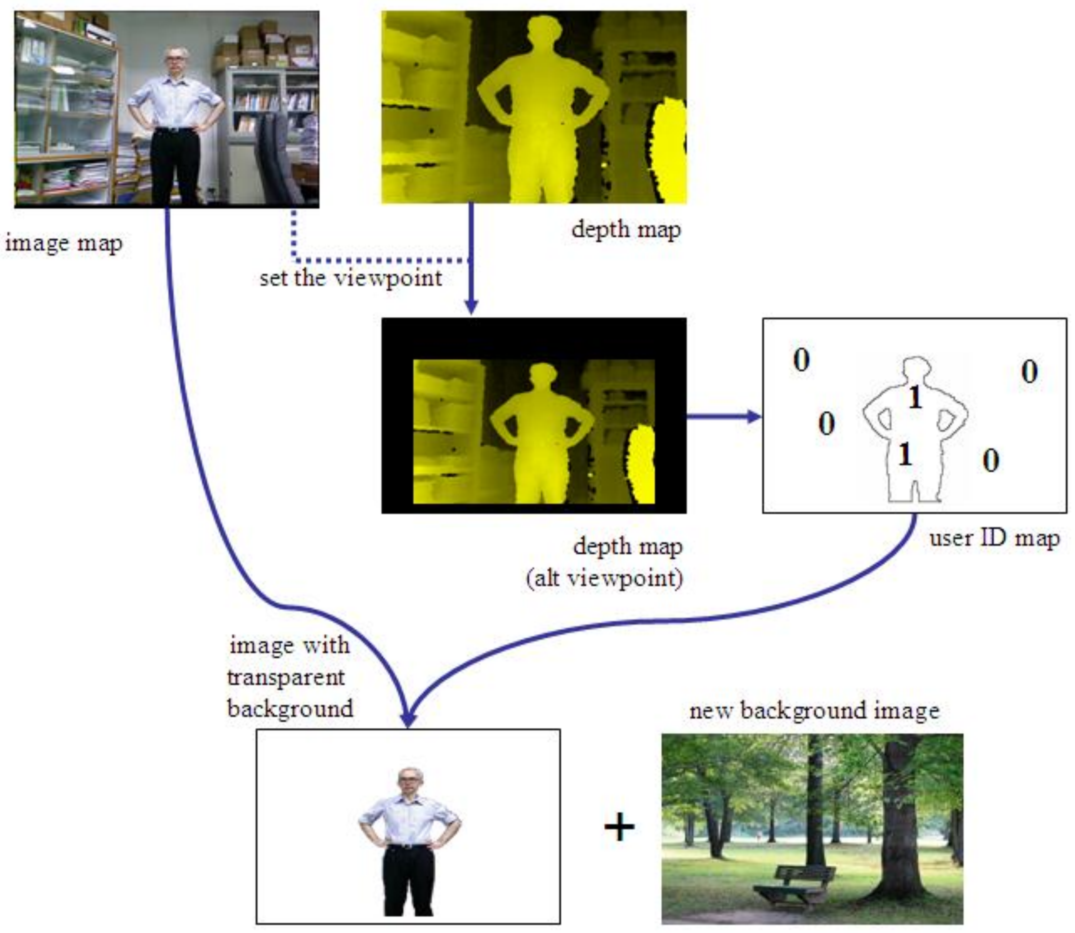

Figure 2. Changing the Background.

An important step is making sure that the image and depth generators utilize the same viewpoint. The Kinect's depth map data is derived from its IR sensor, which is positioned a few millimeters away from the camera, and so doesn't see the same view of the scene.

Version 6 of the imaging application of chapter 2 addressed this viewpoint issue by calling the AlternativeViewPointCapability API. Actually, that's not quite true since the OpenNI version I was using back then had some typos which meant I couldn't get my code to work. This chapter's application is the first working example of that API. (It's working now because I'm using a more recent version of the library.)

The adjusted depth map is the data source for the construction of the user ID map, so that map will also be correctly aligned with the camera image.

A body-based approach to identifying the background means that there's no need for me to drape a blue sheet around my office, but there are some drawbacks.

One problem, which is visible in Figure 2, is that the depth map covers less space than the camera image. After it's been scaled and repositioned by the viewpoint change, user detection is limited to the central of the camera image. Even if the user stands directly in front of the Kinect, parts of his legs won't be detected because the depth map doesn't reach to the lower edge of the camera image. This problem can be seen in Figure 3, where my legs disappear just above the bottom edge of the panel.

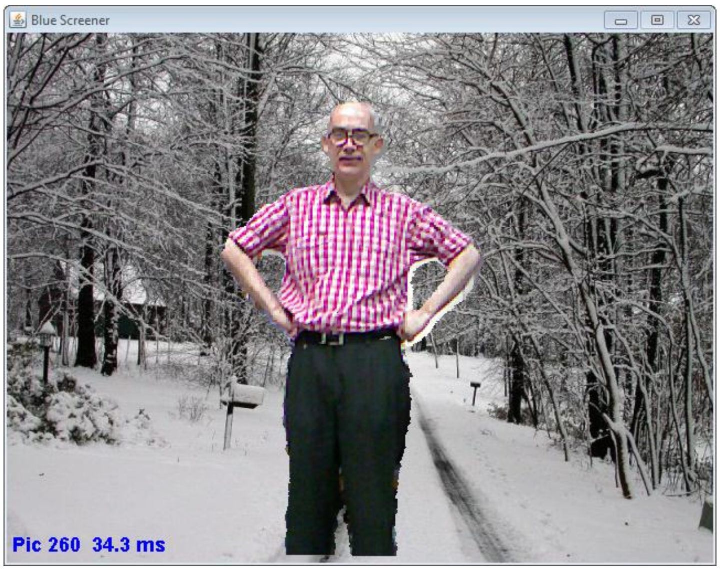

Figure 3. Out in the Snow.

Another drawback is that the Kinect doesn't do a particularly accurate job of identifying a body outline, which shows itself as thick white edges around parts of my body in Figure 3. The edges tend to flicker since each update slightly changes them.

The UML class diagrams for this application, called ChangeBG, are shown in Figure 4.

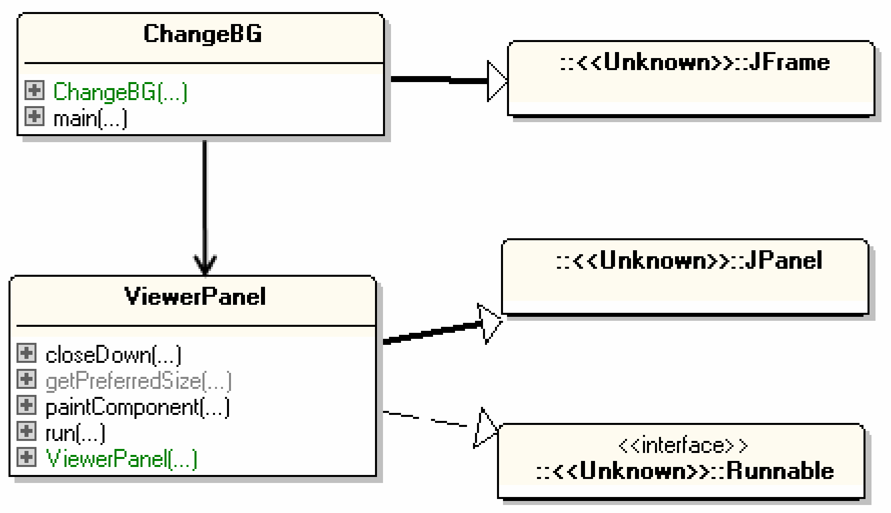

Figure 4. Class Diagrams for the ChangeBG Application.

The ChangeBG class is little more than a JFrame wrapper around a rendering panel implemented by ViewerPanel, and many of its elements will be familiar from the examples in chapter 2. I'll concentrate on explaining the novel elements of ViewerPanel, in particular its use of a user generator node and the AlternativeViewPointCapability API

1. Setting up OpenNI

The ViewerPanel class calls configOpenNI() to create image, depth, and user generator nodes. It also handles the adjustment of the depth node's viewpoint.

```
// globals
private Context context;
private ImageGenerator imageGen;
private DepthGenerator depthGen;
private DepthMetaData depthMD;
private SceneMetaData sceneMD;


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

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

    imageGen = ImageGenerator.create(context);
    depthGen = DepthGenerator.create(context);

    // set viewpoint of DepthGenerator to match ImageGenerator
    boolean hasAltView =
       depthGen.isCapabilitySupported("AlternativeViewPoint");
    if (hasAltView) {
      AlternativeViewpointCapability altViewCap =
               depthGen.getAlternativeViewpointCapability();
      altViewCap.setViewpoint(imageGen);
    }
    else {
      System.out.println("Alternative ViewPoint not supported");
      System.exit(1);
    }

    MapOutputMode mapMode = new MapOutputMode(640, 480, 30);
    depthGen.setMapOutputMode(mapMode);
    imageGen.setMapOutputMode(mapMode);
    imageGen.setPixelFormat(PixelFormat.RGB24);

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

    depthMD = depthGen.getMetaData();

    UserGenerator userGen = UserGenerator.create(context);
    sceneMD = userGen.getUserPixels(0);
       // used to return a map containing user IDs (or 0)
       // at each depth location

    context.startGeneratingAll();
    System.out.println("Started context generating...");
  }
  catch (Exception e) {
    System.out.println(e);
    System.exit(1);
  }
}  // end of configOpenNI()
```

After the creation of the image and depth generators, Generator.getAlternativeViewpointCapability() obtains the viewpoint capability for the depth generator, and sets it's viewpoint to be that of the image generator with a call to AlternativeViewpointCapability.setViewpoint().

The UserGenerator node produces data relating to bodies in the scenes, such as the number of users and their centers of mass (CoMs). I want to create a scene map whose pixels hold user IDs. The first step is to initialize a global SceneMetaData object:

```
UserGenerator userGen = UserGenerator.create(context);
sceneMD = userGen.getUserPixels(0);
```

The 0 argument to UserGenerator.getUserPixels() specifies that the ID map should contain the IDs of all the users in the scene at each depth location. The IDs are positive numbers, or 0 for the background.

2. Updating the Nodes

The run() method in the threaded part of ViewerPanel uses Context.waitAndUpdateAll() to wait for all the nodes to have new data, before updating them. This helps to keep the camera and IR sensors in rough synchronization.

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


public void run()
{
  isRunning = true;
  while (isRunning) {
    try {
      context.waitAndUpdateAll();
    }
    catch(StatusException e)
    {  System.out.println(e);
       System.exit(1);
    }
    long startTime = System.currentTimeMillis();
    screenUsers();
    totalTime += (System.currentTimeMillis() - startTime);
    repaint();
  }

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

3. Screening Users

The screenUsers() method carries out the removal of the background shown as the outcome of Figure 2. It begins by converting the standard Kinect camera image buffer into a pixel array called cameraPixels.

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


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, imWidth, imHeight, cameraPixels,
                                        0, imWidth );
  imageCount++;
}  // end of screenUsers()
```

I described convertToPixels() in detail in chapter 2, so won't go through it again. hideBackground() sets the background pixels in the array to be transparent, and the result is converted into a BufferedImage back in screenUsers().

hideBackground() uses information from the user ID maps to turn the background pixels transparent in the cameraPixels array.

```
// globals
private DepthMetaData depthMD;
private DepthGenerator depthGen;
private SceneMetaData sceneMD;
private int hideBGPixel;    // the "hide the background" pixel


private void hideBackground(int[] cameraPixels)
{
  depthMD = depthGen.getMetaData();
        // reassignment to avoid the viewpoint changing

  // update the user ID map
  ShortBuffer usersBuf = sceneMD.getData().createShortBuffer();

  while (usersBuf.remaining() > 0) {
    int pos = usersBuf.position();
    short userID = usersBuf.get();       // get and incr position
    if (userID == 0) // if not a user (i.e. part of background)
      cameraPixels[pos] = hideBGPixel;   // make pixel transparent
  }
}  // end of hideBackground()
```

The function is complicated by having to avoid a bug in OpenNI's support for the alternative viewpoint mechanism. Although the depth map's viewpoint was changed at initialization time in configOpenNI(), it sometimes switches back to its original viewpoint. I happened upon a fix for this by reassigning the depth meta data to depthMD at the start of the function.

Note that hideBackground() does not need to use the depth map since the user ID map (usersBuf) contains all the required information.

The hideBGPixel value assigned to cameraPixels[] is set to be a transparent blue in ViewPanel's constructor:

```
// in ViewPanel()
hideBGPixel =  new Color(0, 0, 255, 0).getRGB();
```

There's no reason why it’s transparent blue as opposed to a different invisible color.

4. Rendering the Scene

Rendering involves drawing the background image (whose filename is supplied when the application is started), followed by the modified Kinect camera image. The image's transparent parts will allow the background to show through.

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


public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D) g;

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

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

writeStats() draws the image count and processing time at the bottom left of the panel.