Kinect Chapter 2.5. Transforming the User

In the previous two chapters I looked at how to distinguish the user from the background of a Kinect camera image. Chapter 2.3 explained the coding technique (involving imaging, depth data, and user ID generator nodes). The essential idea is to convert each camera frame into a BufferedImage with the background pixels made transparent. A new 'virtual' background is then drawn behind the image. Chapter 2.4 implemented user/scene interaction using the fact that the non-user pixels in the image are invisible.

This chapter focuses on how the user image can be changed without affecting the virtual background. This is surprisingly easy because standard Java image processing techniques, such as blurring and pixel color effects, can be utilized. There are several imaging libraries that offer such effects, and I've used Jerry Huxtable's JH Labs image filters (http://www.jhlabs.com/ip/filters/) for the examples here.

Library methods usually apply their effects to all the pixels in an image, but since the background pixels in the Kinect image are transparent, changes to those parts will typically remain invisible.

Figure 1 shows two screenshots of the KTransformer program, with the "Chrome" effect selected from the menu.

Figure 1. The Chrome Effect.

The effect is applied to consecutive frames coming from the Kinect camera, so the user remains chrome-plated as he walks around.

KTransformer allows a filter's parameters to be changed at run time. For example, Figure 2 shows the "Dissolve" filter in action – its "density" parameter cycles from 0 to 1 and back again over the course of a few seconds, making the user repeatedly fade away and reappear.

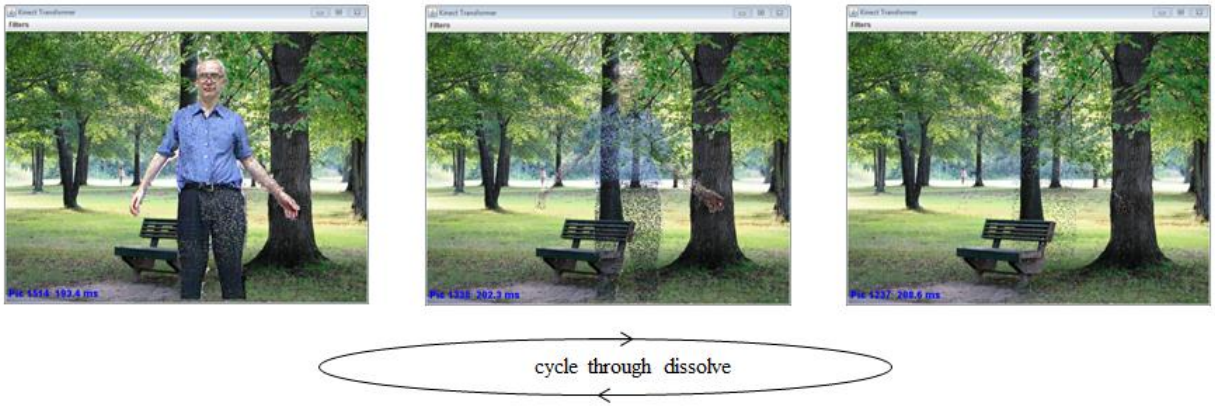

Figure 2. The Varying Dissolve Effect.

The class diagrams for the application are shown in Figure 3.

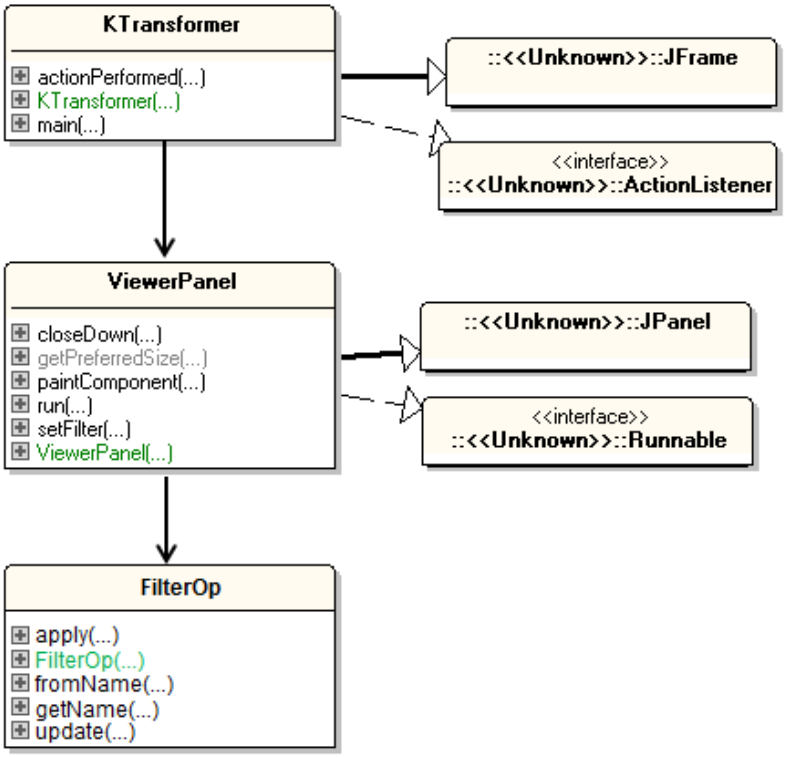

Figure 3. The KTransformer Class Diagrams.

The KTransformer class implements the JFrame, which renders the scene in a panel created with ViewerPanel. KTransformer also includes a menu bar with a single "Filters" menu for all the filters. When the user selects a filter menu item, the scene changes to show its effect. A close-up of the menu is shown in Figure 4.

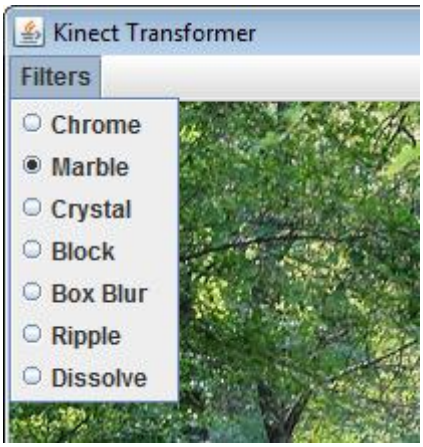

Figure 4. The KTransformer Menu.

ViewerPanel performs the same operations as the same-named class in the ChangeBG application in chapter 2.3, with additional code for updating and applying the current filter to the camera image. A large part of ViewerPanel's work involves subtracting the background from the Kinect camera image, leaving only the user visible. It does this in exactly the same way as previously, so I won't repeat the details again.

FilterOp is an enum type which encapsulates the details of creating, updating, and applying filters, which come from the JH Labs image library (http://www.jhlabs.com/ip/filters/). The API offers over 100 operations, but KTransformer employs only seven (as listed in Figure 4's menu). Mostly this is to reduce FilterOp's size, but some of the JH Labs operations aren't suitable, for reasons I'll explain at the end.

1. Creating the Filter Menu

KTransformer creates the Filter menu by calling its buildFilterMenu() method. The menu items are radio buttons, grouped so that only one can be enabled at a time.

```
private void buildFilterMenu(FilterOp startFop)
{
  JMenuBar mb = new JMenuBar();
  JMenu fMenu = new JMenu("Filters");
  mb.add(fMenu);

  // build the menu items
  FilterOp[] values = FilterOp.values();
                        // get all FilterOp constants
  JRadioButtonMenuItem mi;
  ButtonGroup group = new ButtonGroup();
  for(FilterOp fop : values) {
    mi = new JRadioButtonMenuItem(fop.getName());
               // create menu item from FilterOp name
    if (fop == startFop)
      mi.setSelected(true);   // set startFop menu item to be "on"
    mi.addActionListener(this);
    fMenu.add(mi);
    group.add(mi);
```



```
  }

  setJMenuBar(mb);
}  // end of buildFilterMenu()
```

The names used in the menu items are obtained from the FilterOp enum by looping through all its enumeration values. FilterOp.getName() retrieves each constant's name string.

buildFilterMenu() is called from the KTransformer() constructor with the FilterOp constant FilterOp.MARBLE as the starting filter. The marble effect is shown in Figure 5.

Figure 5. The Marble Effect.

KTransformer implements the ActionListener interface, and actionPerformed() is called whenever a filter menu item is selected:

```
// global
private ViewerPanel viewerPanel;


public void actionPerformed(ActionEvent e)
{
  JRadioButtonMenuItem mi =(JRadioButtonMenuItem)e.getSource();
  mi.setSelected(true);
  String filterName = mi.getText();   // get selected item's name
  System.out.println("Selected: " + filterName);

  FilterOp fop = FilterOp.fromName(filterName);  //name --> FilterOp
  viewerPanel.setFilter(fop);
}  // end of actionPerformed()
```

The selected menu item's name is used by FilterOp.fromName() to look up its associated FilterOp constant. This constant is passed to the panel to change the current filter.

2. Updating and Redrawing the Panel

The update-draw loop in ViewerPanel.run() is very similar to earlier versions. The main changes are that the update steps include an update to the current FilterOp and the filter's application to the camera image.

```
// in ViewerPanel
// globals
private volatile boolean isRunning;
private int imageCount = 0;
private long totalTime = 0;
private BufferedImage cameraImage;

private FilterOp fop;   // current filter


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();

    fop.update(totalTime);
    screenUsers();  // make the background transparent
    cameraImage = fop.apply(cameraImage);
        // modify the camera image with the current filter
    imageCount++;

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

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

The FilterOp variable is initially assigned FilterOp.MARBLE so the user image is displayed with a marble effect (as shown in Figure 5). This fop value can be changed by the user selecting a menu item, which triggers a call to ViewerPanel.setFilter():

```
public void setFilter(FilterOp fp)
// called from the top-level to change the current filter
```



```
{ fop = fp;  }
```

3. Timing the Filters

The totalTime global in ViewerPanel stores the application's execution time, and imageCount the number of images processed. These are used by writeStats() to calculate an average iteration time for the run() loop, which is drawn at the bottom left of the panel. This information is very useful for judging a filter's speed.

Table 1 lists average iteration times for the filter operations, rounded to the nearest 10 ms value.

Filter	Avg. Iteration Time (ms)
None	30
Chrome	100 and v.slowly increases
Marble	90
Crystal	150 and increases
Block	90
Box Blur	80
Ripple	120 and slowly increases
Dissolve	80

Table 1. Average Iteration Filter Times.

The "None" row gives the average iteration time when there's no filter applied to the image, so shows the average time required to refresh the Kinect image and make its background invisible.

Four of the filters ("Marble", "Block", "Box Blur", and "Dissolve") execute at reasonable speeds. The "Block" and "Box Blur" effects are shown in Figures 6 and 7.

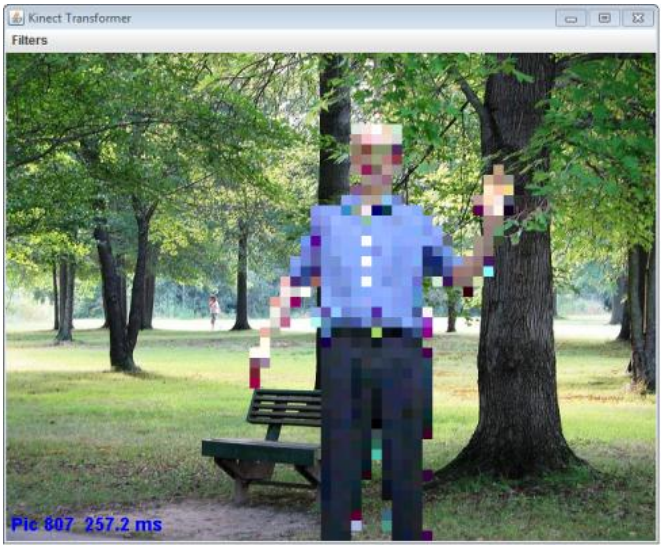

Figure 6. The Block Effect.

The "Box Blur" effect has one of its parameters updated at run time – the blurring amount cycles between 0 (no blur) and 12.

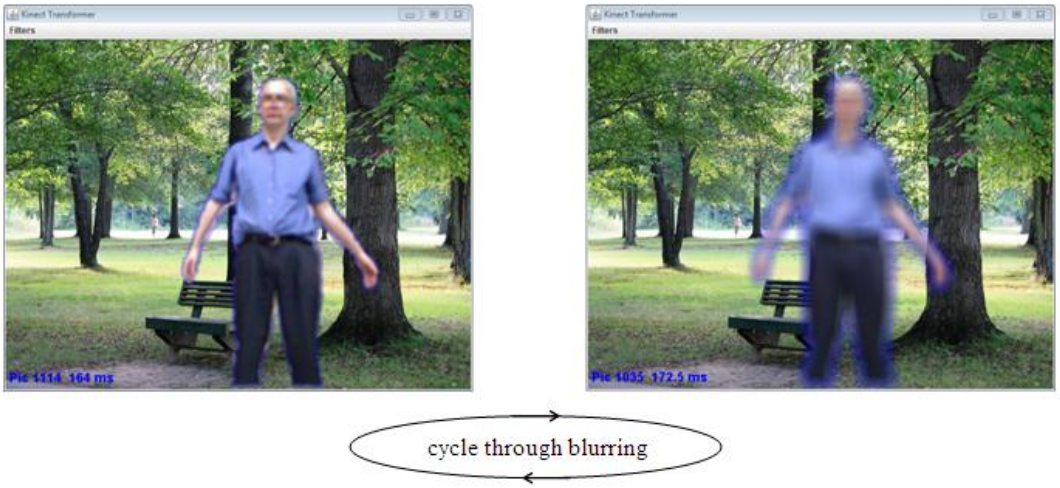

Figure 7. The Box Blur Effect.

The other three filters in Table 1 ("Chrome", "Crystal", and "Ripple") perform poorly, although they start off with good iteration times. "Crystal" is the slowest, perhaps because I set its edge color parameter to be transparent. Figure 8 shows the "Crystal" effect, and Figure 9 the "Ripple".

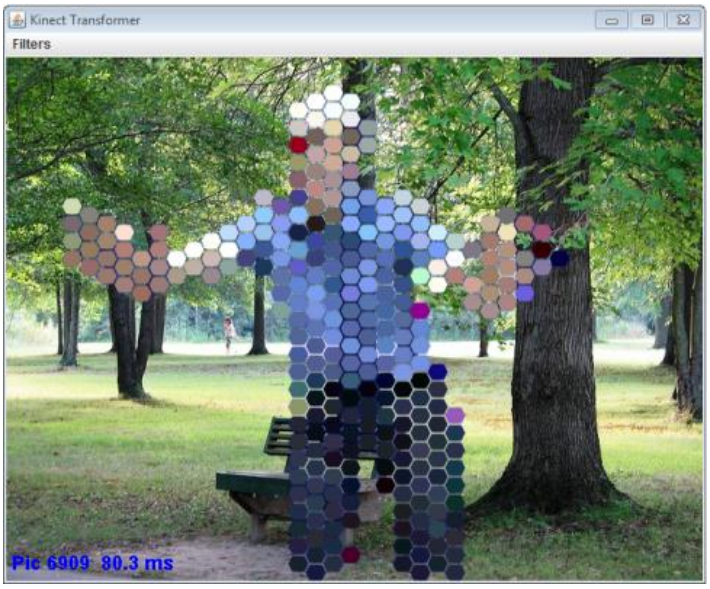

Figure 8. The Crystal Effect.

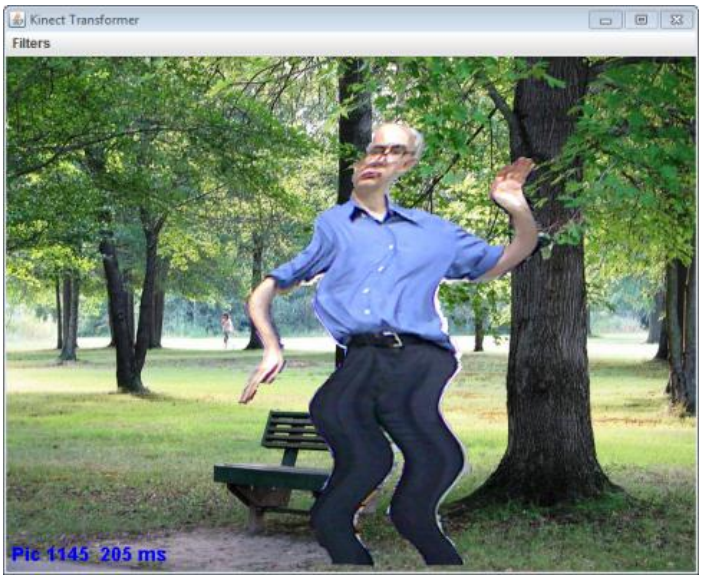

Figure 9. The Ripple Effect.

The "Chrome", "Crystal", and "Ripple" effects progressively get slower the longer they execute, with "Crystal" slowing the most quickly. This suggests that their coding may have some memory problems.

4. The FilterOp Enumerated Type

Each filter in the FilterOp enum has three parts – an enumeration constant, a name string, and its BufferedImageOp operation. Each operation is initialized using a static method called op().

```
// enum globals
// declare constants, name strings, and BufferedImage ops
CHROME("Chrome", op("Chrome")),
MARBLE("Marble", op("Marble")),
CRYSTAL("Crystal", op("Crystal")),
BLOCK("Block", op("Block")),
BOX_BLUR("Box Blur", op("Box Blur")),
RIPPLE("Ripple", op("Ripple")),
DISSOLVE("Dissolve", op("Dissolve"));

private static int hideBGPixel = new Color(0, 0, 255, 0).getRGB();
                                  // transparent blue

// enum parameters
private String name;
private BufferedImageOp op;


FilterOp(String name, BufferedImageOp op)
{  this.name = name;
   this.op = op;
}
```



```
private static BufferedImageOp op(String name)
{
  BufferedImageOp op = null;
  System.out.println("Creating op for " + name);

  if (name.equals("Chrome")) {
    op = new ChromeFilter();
    ((ChromeFilter)op).setBumpHeight(3f);
  }
  else if (name.equals("Marble")) {
    op = new MarbleFilter();
    ((MarbleFilter)op).setXScale(8f);
    ((MarbleFilter)op).setYScale(8f);
  }
  else if (name.equals("Crystal")) {
    op = new CrystallizeFilter();
    ((CrystallizeFilter)op).setEdgeColor(hideBGPixel);
  }
  else if (name.equals("Block")) {
    op = new BlockFilter();
    ((BlockFilter)op).setBlockSize(10);
  }
  else if (name.equals("Box Blur"))      // can be updated later
    op = new BoxBlurFilter();
  else if (name.equals("Ripple")) {
    op = new RippleFilter();
   ((RippleFilter)op).setXAmplitude(12f);
  }
  else if (name.equals("Dissolve"))      // can be updated later
    op = new DissolveFilter();
  else
    System.out.println("Did not recognize op name");

  return op;
}  // end of op()
```

The classes used in op() come from Jerry Huxtable's filter library, available from http://www.jhlabs.com/ip/filters/ as a JAR file. The website includes helpful screenshots of all the filters (about 100), but documentation on the filters' parameters and methods is rather sparse. I used the excellent JD-GUI decompiler (from http://java.decompiler.free.fr/?q=jdgui) to examine the decompiled JAR file, and then I experimented with each classes methods.

4.1. Updating a Filter

FilterOp.update() shows how a filter's behavior can be modified at run time by adjusting its parameters. To keep things relatively simple, the parameter change is hardwired to be cyclic, using a variable that moves in fractional steps from 0 to 1 and back again. The cycle is calculated using the current execution time of the application, modulo a constant called CYCLE_TIME (3000 ms). This approach means that a single cycle takes about 3 seconds to complete. update() is defined as follows:

```
// globals
private static final int CYCLE_TIME = 3000;
                    // time (in ms) for an update cycle
private BufferedImageOp op;
```



```
public void update(long totalTime)
{
  if (op == null)
    return;

  float cycle = (float)(totalTime % CYCLE_TIME)*2/CYCLE_TIME;
            // produces a value between 0f and 2f
  if (cycle > 1f)
    cycle = 2f - cycle;     // so goes 0 - 1 - 0

  if (this == BOX_BLUR) {
    cycle = cycle*12f;    // so cycles 0 - 12 - 0
    ((BoxBlurFilter)op).setRadius((int)cycle);
  }
  else if (this == DISSOLVE)
    ((DissolveFilter)op).setDensity(cycle);
}  // end of update()
```

The amount of blurring in the "Box Blur" effect (see Figure 7) is controlled by adjusting its radius parameter with BoxBlurFilter.setRadius(). The amount of dissolving (se Figure 2) is controlled with DissolveFilter.setDensity().

4.2. Applying a Filter to an Image

The filter modifies the camera image in apply():

```
// global
private BufferedImageOp op;


public BufferedImage apply(BufferedImage im)
{
  if (op == null)
    return im;
  else
    return op.filter(im, null);
}  // end of apply()
```

This code may seem inefficient because the filter is applied to the entire image, including the invisible background. One optimization might be to crop the image so that it excludes the invisible parts. However, care must be taken not to crop too much since some operations cause transparent pixels to become visible. For example, the blurring in Figure 7 extends the edges of the user image.

A related issue is that some filters change the size of the image, making it difficult to draw a cropped picture in the same place on the screen. This shows itself as a jittering of the image about the panel, but is less of a problem when the visible parts are surrounded by invisible pixels.

5. Choosing a Filter

Many filters specify a coordinate where the effects are centered. For example, the JH Labs "Twirl" uses the center of the image as a rotation point. The twirled result looks quite amusing until the user steps away from the center, because the effect doesn't follow him. "Twirl" and other coordinate-based effects have methods for moving their locations, but choosing a coordinate would require more analysis of the Kinect image (for example, to find the user's center-of-mass). I've taken the easier alternative of not using any filters that work relative to a position.

Another issue is that some effects are too slow to be used in real-time, taking over 200 ms to be processed. For that reason, I wouldn't use the "Chrome", "Crystal", or "Ripple" effects in an application.

Although I employed JH Lab filters here, there are many other Java image processing libraries that offer interesting effects. They include:

- ImageJ (http://rsbweb.nih.gov/ij/). ImageJ claims to perform the fastest pure Java image processing. It comes with a large range of effects and additional plug-ins.
- Marvin (http://marvinproject.sourceforge.net/). It offers multithreaded image processing, and plug-ins.
- JMagick (http://www.jmagick.org/). A thin JNI layer above the popular ImageMagick API (http://www.imagemagick.org/).
- im4java (http://im4java.sourceforge.net/). im4java also utilizes ImageMagick but by generating command line calls.
- NeatVision (http://www.neatvision.com/). Nearly 300 image manipulation, processing and analysis operations are available, but the library is no longer being developed.

Another solution is to create your own filters. An excellent book that discusses BuffereImageOp is "Java 2D Graphics" by Jonathan Knudsen (O'Reilly 1999).

