6.837 — FALL '00

Midterm Quiz

Time allotted: 1 hour 15 minutes

Open book, open notes

(100 points + 5 extra points)

Instructions: Answer all questions using the space provided. If you need more paper, raise your hand, and it will be provided for you. If you have a question during the test, do not leave your seat. Instead, raise your hand until someone comes to assist you. If any question seems unclear or appears to lack sufficient information, state a reasonable assumption (in writing on your quiz) and proceed. The point value of each problem is indicated in parentheses. Make sure that you answer each part of each question, even if you believe you have already answered the question in an earlier part.

Name Solutions
(Print)

I understand that this quiz is subject to the conditions and procedures stated in MIT Policies and Procedures handbook in Section 10.2. Furthermore, I neither gave, nor received, any unsanctioned assistance during this examination.

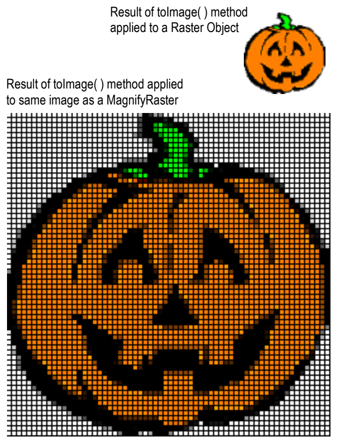
Signed ______________________ Date ________

Problem 1. RASTER ADDRESSING: You are given the following familiar object:

```
public class Raster {
  ///////////////////////// Constructors ////////////////////
  public Raster();                // allows class to be extended
  public Raster(int w, int h);    // specify size
  public Raster(Image img);       // set to size and contents of image

  ///////////////////////// Accessors ////////////////////////
  public int getSize( );          // pixels in raster
  public int getWidth( );         // width of raster
  public int getHeight( );        // height of raster
  public int[] getPixelBuffer( ); // get array of pixels

  ///////////////////////// Methods //////////////////////////
  public void fill(int argb);     // fill with packed argb
  public void fill(Color c);         // fill with Java color
  public Image toImage( );
  public int getPixel(int x, int y);
  public Color getColor(int x, int y);
  public boolean setPixel(int pix, int x, int y);
  public boolean setColor(Color c, int x, int y);
}
```

Suppose you are asked to extend the given Raster class to create the new class called MagnifyRaster, which has two additional methods, setMagnification( ) and getMagnification( ), as shown below:

```
public class MagnifyRaster extends Raster {
  protected int magFactor = 4;

  public MagnifyRaster( ) {
  }

  public MagnifyRaster(int w, int h) {
    super(w, h);
  }

  public MagnifyRaster(Image img) {
    super(img);
  }

  public void setMagnification(int f) {
    magFactor = f;
  }

  public int getMagnification( ) {
    return magFactor;
  }

  public Image toImage( ) {
    // YOUR CODE HERE
  }
}
```


Result of toImage( ) method applied to a Raster Object
Result of toImage( ) method applied to same image as a MagnifyRaster

The MagnifyRaster toImage( ) method returns an image where each pixel of the raster is displayed as a magFactor by magFactor sized block, with a one pixel wide, black border to the right and below each box. In all other respects, a MagnifyRaster's behavior is indistinguishable from that of a Raster. An example of an image returned by a MagnifyRaster is shown above.

Part A: (15 points) Write a code fragment to implement the toImage( ) method of MagnifyRaster.

My original version:

```
public Image toImage( ) {
      int zoom = magFactor + 1;
      int w = this.width*zoom, h = this.height*zoom;
      Raster bigRaster = new Raster(w, h);
      int bigPixel[] = bigRaster.getPixels( );
      int i = 0, k = 0;
      int borderColor = 0xff000000;

      for (int y = 0; y < height; y++) {
         for (int v = 0; v < zoom; v++) {
            for (int x = 0; x < width; x++) {
               int pix = pixel[i++];
               for (int u = 0; u < zoom; u++) {
                  bigPixel[k++] = ((u == magFactor) || (v == magFactor)) ? borderColor : pix;
               }
            }
            i -= width;
         }
         i += width;
      }
      return bigRaster.toImage();
   }
```

Using tricks I learned while grading:

```
public Image toImage( ) {
      int zoom = magFactor + 1;
      int w = this.width*zoom, h = this.height*zoom;
      Raster bigRaster = new Raster(w, h);
      int pix;

      for (int y = 0; y < h; y++) {
         for (int x = 0; x < w; x++) {
            if (((x+1) % zoom == 0) || ((y+1) % zoom == 0)) {
               pix = 0xff000000;                // black border
            } else {
               pix = getPixel(x/zoom, y/zoom);
            }
            bigRaster.setPixel(pix, x, y);
         }
      }
      return bigRaster.toImage();
   }
```

Part B: (10 points) After writing your MagnifyRaster class, you decide to incorporate it into an applet with the following approximate structure in an effort to test the new class:

```
public class TestMagnify extends Applet {
    MagnifyRaster mRaster;

    public void init( ) {
        mRaster = new MagnifyRaster(getImage(getDocumentBase(), "myimage.jpg"));
    }

    public void paint(Graphics g) {
        g.drawImage(mRaster.toImage(), 0, 0, this);
    }

    public void update(Graphics g) {
        paint(g);
    }

    public boolean mouseDown(Event e, int x, int y) {
        /*
          If the pixel at (x, y) falls on a non-gridline then the
          color of the "clicked" pixel box is permuted as follows:
             the new red channel is set equal to the old green channel,
             the new green channel is set equal to the old blue channel, and
             the new blue channel is set equal to the old red channel.

          This color change will be reflected on the screen as well as
          by contents of mRaster.
         */

        // MISSING CODE FRAGMENT

        repaint();
        return true;
    }
}
```

Write the missing code fragment of the mouseDown method as described by the block comment. You are welcome to declare any necessary variables.

```
public boolean mouseDown(Event e, int x, int y) {
      int mag = mRaster.getMagnification( );
      int tilesize = mag + 1;
      if (((x % tilesize) == mag) || ((y % tilesize) == mag)) {
        return true;
      }
      x = x / tilesize;
      y = y / tilesize;
      int pix = mRaster.getPixel(x, y);
      pix = 0xff000000 + ((pix & 0x0000ffff) << 8) + ((pix >> 16) & 255);
      mRaster.setPixel(pix, x, y);
      repaint( );
      return true;
   }
```

Problem 2. TWO DIMENSIONAL TRANSFORMATIONS: The three-parameter family of image transformations called Euclidean have the form:

$$E(t_x, t_y, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & t_x \\ \sin(\theta) & \cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

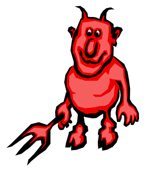
This family of transforms forms an algebraic group whose function can be described as a rotation by θ about the origin followed by a translation by $[t_x, t_y]$.

Part A: (15 points) Derive the matrix formulation for a three-parameter family of image transforms, *$R(p_x, p_y, \theta)$*, whose function is to rotate coordinates about the point $[p_x, p_y]$ by the angle θ.

$$R(p_x, p_y, \theta) = \begin{bmatrix} 1 & 0 & p_x \\ 0 & 1 & p_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -p_x \\ 0 & 1 & -p_y \\ 0 & 0 & 1 \end{bmatrix}$$

$$R(p_x, p_y, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & -p_x \bullet \cos(\theta) + p_y \bullet \sin(\theta) + p_x \\ \sin(\theta) & \cos(\theta) & -p_y \bullet \cos(\theta) - p_x \bullet \sin(\theta) + p_y \\ 0 & 0 & 1 \end{bmatrix}$$

Part B: (5 points) Does this family of transforms, *$R(p_x, p_y, \theta)$*, form an algebraic group? Explain why or why not.



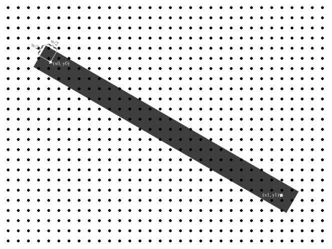
This was the devilish question. There is no reason to expect you to have figured this one out. The answer is no. While is obeys the three axioms of a group (the associative law, the set includes the inverse of every member, and the set includes identity), it is not closed under composition. The fact is these rules are *necessary*, but not *sufficient* for defining a group. My treatment in class might have misled you on this detail. To prove that a set is closed under composition can be tricky. In order to prove that a set is not closed you need merely state a contradiction.

Consider the composition of $R(x_o, y_o, `\pi)R(x_o/2, y_o/2, `\pi)$. The net result is a translation, $T(x_o, y_o)$. However, one cannot select x, y, and `θ such that $R(x, y, `\theta) = T(x_o, y_o)$. Thus, 2D rotations about an arbitrary point are not closed. Everyone got 5 points for this question.

Part C: (5 points) How does the *$R(p_x, p_y, \theta)$* family of transforms relate to the Euclidean family (i.e. are they a subset, a superset, or equivalent)? Explain.

The set of 2D rotations about an arbitrary point is a subset of the Euclidean family. We know this since, all members of R(px, py, θ) can be decomposed into a series of Euclidean transforms, as shown in part A, and the set of Euclidean transforms is closed, thus R(px, py, θ) is at least a subset. However, the class of transforms R(px, py, θ) does not include pure translations without rotations. Notice R(px,py,0) = I, regardless of px, py, thus you cannot make the diagonal of R(px, py, θ) all 1' s (as is the case with a pure translation) without also making the two upper elements of the left-most column 0' s. This makes the origin a very special place.

Problem 3. LINE DRAWING ALGORITHMS AND SCAN CONVERSION: All of the line drawing algorithms described in class have assumed that the endpoints are specified as integers and that the line width is fixed to one pixel. In this problem, we will investigate a more general class of line drawing algorithms that allow arbitrary endpoints (specified as float values) and variable widths.



An example of the type of line that we seek to render is illustrated in the figure to the right. The following fragment of Java code implements this scan converter:

```
public void Draw(Raster r) {
    int width = r.getWidth();
    int height = r.getHeight();
    int pixel[] = r.getPixelBuffer();
    int xMin, xMax, yMin, yMax;
    int x, y;

    // compute the bounding box
    // CODE OMITTED

    // compute the discriminator along the length of the line
    float Al = y0 - y1;
    float Bl = x1 - x0;

    // scale the discriminator to measure distance from the line
    float d = (float) Math.sqrt(Al*Al + Bl*Bl);
    Al = Al/d;
    Bl = Bl/d;
    d = 0.5f*(d + w);

    // compute the discriminator along the width of the line
    float Aw = -Bl;
    float Bw = Al;

    // initialize the discriminator
    float tl = Al*xMin + Bl*yMin - 0.5f*(Al*(x0 + x1) + Bl*(y0 + y1));
    float tw = Aw*xMin + Bw*yMin - 0.5f*(Aw*(x0 + x1) + Bw*(y0 + y1));

    yMin *= width;
    yMax *= width;

    //     .... scan convert line ....
    for (y = yMin; y <= yMax; y += width) {
        float el = tl;
        float ew = tw;
        boolean beenInside = false;
        for (x = xMin; x <= xMax; x++) {
            if ((el > -0.5f*w) && (el < 0.5f*w) && (ew > -d) && (ew < d)) {
                pixel[y+x] = argb;
                beenInside = true;
            } else if (beenInside) break;
            el += Al;
            ew += Aw;
        }
        tl += Bl;
        tw += Bw;
    }
}
```

This Draw( ) method is defined within a FatLine Object with the following floating point instance variables: x0, y0, x1, y1, and w. These are defined according to the figure above.

Part A: (5 points) Give expressions for the constant terms of the two discriminators, Cl and Cw (i.e. the C term in the expression A x + B y + C = 0).

The constant terms are just the right-most terms of the two expressions that initialize the discriminators:

Cl = -0.5*(Al*(x0 + x1) + Bl*(y0 + y1))
Cw = -0.5*(Aw*(x0 + x1) + Bw*(y0 + y1))

Alternatively, you could derive the expression as:

$$Cl = \frac{x0\,y1 - x1\,y0}{\sqrt{(x0-x1)^2-(y0-y1)^2}} \quad Cw = -\frac{(x0+x1)(x0-x1)+(y0+y1)(y0-y1)}{2\sqrt{(x0-x1)^2-(y0-y1)^2}}$$

but, for reasons discussed in lecture, this form in not as well behaved numerically.

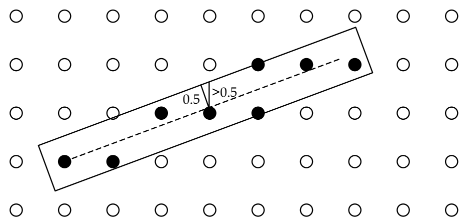
Part B: (10 points) Another approach to scan-converting variable-width lines would use 4 oriented-edge equations to represent the boundaries of the line, similar to the way we represented triangles. Discuss any advantages or disadvantages of this alternative method when compared to the method given.

A clear disadvantage of using 4 edge equations is that you would have to update 4 discriminators in the inner loop of the rasterizer rather than 2 as in this version. You would also have to compute those extra coefficients, which could be a little tricky.

One advantage of the 4 edge-equation approach is it avoids normalizing the discriminator coefficients, thus possibly avoiding the square-root.

Part C: (10 points) If the line width of the given line drawing method is set to 1 and the endpoints are limited to integer values, would it always generate the same result as the DDA line drawing algorithm given in class? Explain why or why not.

No, it will not generate the same result as a DDA. A DDA line-drawing algorithm will set the *one* closest pixel on each row or column (depending upon which is the dominant axis). A pixel-sampling approach, like the one given, will set all pixels that satisfy the discriminators. The closest pixel of the DDA is determined at integer values along the dominant axis and is no more than 0.5 a pixel from the ideal line as measured in the minor axis direction. The given algorithm will turn on all pixels within 0.5 a pixel of the ideal line as measured in the direction perpendicular to the line. This distance is generally greater than 0.5 a pixel measured in the minor axis direction as shown. This results in an occasional minor axis column or row with two pixels set as shown below.



Problem 4. THREE DIMENSIONAL TRANSFORMS: Suppose we define a transformation called *pivot*(a,b) that maps the unit vector a to the desired unit vector b via a rotation of π radians about the vector that bisects the two given vectors, $m = \frac{a+b}{|a+b|}$.

Part A: (10 points) Express pivot as a matrix operator in terms of the three vectors given. Assume the vector elements are indexed as follows $a = [a_1, a_2, a_3]^T$.

$$pivot(a,b) = (1-\cos(\pi))\begin{bmatrix} m_1^2 & m_1m_2 & m_1m_3 \\ m_1m_2 & m_2^2 & m_2m_3 \\ m_1m_3 & m_2m_3 & m_3^2 \end{bmatrix} + \sin(\pi)\begin{bmatrix} 0 & m_3 & -m_2 \\ -m_3 & 0 & m_1 \\ m_2 & -m_1 & 0 \end{bmatrix} + \cos(\pi)\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$pivot(a,b) = 2\begin{bmatrix} m_1^2 & m_1m_2 & m_1m_3 \\ m_1m_2 & m_2^2 & m_2m_3 \\ m_1m_3 & m_2m_3 & m_3^2 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2m_1^2-1 & 2m_1m_2 & 2m_1m_3 \\ 2m_1m_2 & 2m_2^2-1 & 2m_2m_3 \\ 2m_1m_3 & 2m_2m_3 & 2m_3^2-1 \end{bmatrix}$$

Part B: (5 points) Describe the class of input vectors for which the pivot operation is undefined.

When the vectors a and b are antipodes (i. e. $a = -b$).

Part C: (5 points) Suppose a pivot transform was used to modify the current viewing transform view. In particular, suppose it was used to map the original look-at vector, a to the new look-at vector b (the matrix composition would be $[\,view(\dot{eye}, \dot{eye}+a, u)\,][\,pivot(b,a)\,]$ where u is the up vector). Describe the impact of this mapping on the rendered image.

In addition to the final viewing direction being along b rather than a, the up direction will be flipped. Thus, the image will appear upside down relative to the original.

Part D: (5 points) Suppose that two sequential pivot transforms are composed with a viewing transform as follows $[\,view(\dot{eye}, \dot{eye}+a, u)\,][\,pivot(m,a)\,][\,pivot(b,m)\,]$. Assume that coordinates are multiplied as column vectors on the right. Describe the impact of this mapping on the rendered image.

Here the final viewing direction will be along b rather than a, as before, but the up direction will be consistent with the original up vector.