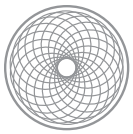


IAT 455 - Final Assignment
Automated Rotoscoper

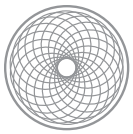
Prepared by:

Ethan Johanson - 301076978

Alannah Darnel - 301087087

Peter Gao -

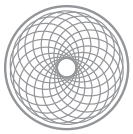
November 28, 2010



Overview

Project Description

The Automated Rotoscoper program was inspired by such works as the film *A Scanner Darkly,* and aims to create a rendered/cel-shaded aesthetic out of any various input image with some user controls to perform adjustments. While not expecting the output quality of the inspiration film, (Done with a lot of input from animators over multiple years) the project aims to create a pleasing output nonetheless.



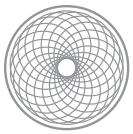
Research

Current Approach #1

The first approach to automated rotoscoping that our team encountered in this project was not actually an entirely automated one. While much of the technique did come down to computer processing of image data, it utilizes animators inputting contour keyframes to be followed and tweened. (As this technique is primarily for video, not every frame is keyed). Nonetheless, we found this approach far more user involved (and complicated with spline generation) than we were able to implement

Current Approach #2

The second researched approach utilized Image segmentation via entropy coding. The results of this technique were highly impressive, but required a level of mathematical knowledge beyond that of the team members, with a variety of techniques such as sphere mapping and multiple sets of vectors. Attempts were initially made to replicate this technique, but all research sources utilized undisclosed MATLAB code in addition to the algorithms they made public, so this was unattainable.



Work

Our Approach

The technique we decided to attempt instead of the unattainable researched techniques utilized a quite simple plan that proved to be a somewhat tricky and imperfect implementation, though many source images were found to work quite well with the technique.

The first step was to generate, via edge detection, a sort of colouring book outline image. This image would serve as a guideline for creating filled regions of solid colour in the output image.

Once this outline was created, fill/space searching algorithms were to be implemented to locate unified regions and mark indexes of these areas.

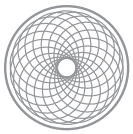
These areas were then sampled in the original image and the colour values of those pixels averaged to generate a colour with which to fill the region.

Regions were then filled and outlines reapplied to generate the final output image.

The various components, all taking in and outputting images as arrays of numbers, were able to be coded independently and combined at the end in a simple Java Applet without too much trouble.

Team Member Work Breakdown

Ethan - Edge Detect, Applet Layout, Research
Peter - Region Detect, Research
Alannah - Colour Average, Applet Layout, Research



Final Result

Problems Encountered

Our main issues with our implementation are:

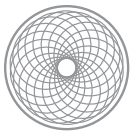
-Edge Detection having difficulties on blurry edges/image bounds. This results in extremely large regions that average out vastly different colours (such as hair and a background)

-Scalability of region finder. Images with many small regions can take very long to process

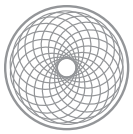
Code

-Sobel.Java

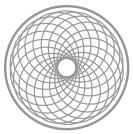
```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
import java.net.*;
import java.io.*;
import java.lang.Math;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.JApplet;
import javax.imageio.*;
import javax.swing.event.*;

public class Sobel {

              //store values
              int[] input;
              int[] output;
              //sobel template
              float[] template={-1,0,1,-1,0,1,-1,0,1}; //Sobel:
{-1,0,1,-2,0,2,-1,0,1}; //Roberts:{0,0,0,0,-1,1,0,0,0};
              int progress;
              int templateSize=3;
```



```
		int width;
		int height;
		double[] direction;

		//constructor
		public Sobel() {
			progress=0;
		}

		//initialize with input image
		public void init(int[] original, int widthIn, int heightIn) {
			width=widthIn;
			height=heightIn;
			input = new int[width*height];
			output = new int[width*height];
			direction = new double[width*height];
			input=original;
		}

		//invert process image result
		public int[] inverted(){
			int[] temp = process();
			for (int x =0; x < temp.length; x++){
				int sum = 0xffffffff - temp[x];
				temp[x] = 0xff000000 | ((int)sum << 16 | (int)sum << 8 |
(int)sum);
			}
			return temp;
		}

		//to process image
		public int[] process() {
			//Y Sobel Operation results
			float[] GY = new float[width*height];
			//X Sobel Operation results
			float[] GX = new float[width*height];
			//final results
			int[] total = new int[width*height];
			progress=0;
			int sum=0;
			int max=0;

			//avoid edges, iterate through width+height
			for(int x=(templateSize-1)/2; x<width-(templateSize+1)/2;x++) {
				progress++;
				for(int y=(templateSize-1)/2; y<height-(templateSize+1)/2;y
++) {

					//reset sum
					sum=0;
```



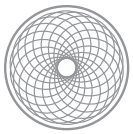
```
                                    //convolve with template
                                    for(int x1=0;x1<templateSize;x1++) {
                                          for(int y1=0;y1<templateSize;y1++) {
                                                int x2 = (x-(templateSize-1)/2+x1); //
current width location minus half of template, plus current x in template
                                                int y2 = (y-(templateSize-1)/
2+y1); //current height location minus half of template, plus current y in template
                                                float value = (input[y2*width+x2] &
0xff) * (template[y1*templateSize+x1]); //multiply nearby input pixel by corresponding
sobel value
                                                sum += value; //add result to sum
                                          }
                                    }
                                    GY[y*width+x] = sum; //sum of current pixel for Y
Sobel

                                    //same system as convolving for Y value
                                    for(int x1=0;x1<templateSize;x1++) {
                                          for(int y1=0;y1<templateSize;y1++) {
                                                int x2 = (x-(templateSize-1)/2+x1);
                                                int y2 = (y-(templateSize-1)/2+y1);
                                                float value = (input[y2*width+x2] &
0xff) * (template[x1*templateSize+y1]); //inverted template x/y
                                                sum += value;
                                          }
                                    }
                                    GX[y*width+x] = sum;

                              }
                        }

                        //go through pixels
                        for(int x=0; x<width;x++) {
                              for(int y=0; y<height;y++) {
                                    //gradient approximation (pythagoras)
                                    total[y*width+x]=(int)Math.sqrt(GX[y*width+x]*GX
[y*width+x]+GY[y*width+x]*GY[y*width+x]);
                                    //direction of gradient
                                    direction[y*width+x] = Math.atan2(GX[y*width+x],GY
[y*width+x]);
                                    //assign new max gradient value if necessary
                                    if(max<total[y*width+x])
                                          max=total[y*width+x];
                              }
                        }

                        float ratio=(float)max/255;
                        //go through pixels
                        for(int x=0; x<width;x++) {
                              for(int y=0; y<height;y++) {
```



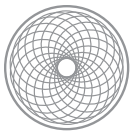
```
                                    //sum is pixel's gradient vector with repect to its
ratio to the maximum
                                    sum=(int)(total[y*width+x]/ratio);
                                    //bitwise create output greyscale pixel of intensity
of gradient edge with full alpha
                                    output[y*width+x] = 0xff000000 | ((int)sum << 16 |
(int)sum << 8 | (int)sum);
                              }
                        }
                        //finish
                        progress=width;
                        return output;
                }

                public double[] getDirection() {
                        return direction;
                }
                public int getProgress() {
                        return progress;
                }

        }
```

-Fill.Java

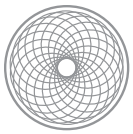
```
import java.awt.image.*;
import java.awt.Color;
import java.util.*;
import java.awt.Point;

public class Fill {
        ColorModel m_colorModel=null;
        public Fill(){

        }

        public int[] queueFill(int[] edge, int[] orig){
                boolean ticker = false;
                Queue<Point> Q = new LinkedList<Point>();
                int[] cellshade = new int[edge.length];;
                int[] temp = new int[edge.length];
                boolean[] checked = new boolean[edge.length];
                for (int i=0; i<checked.length-1;i++){
                        checked[i]=false; //This array holds which pixels have already
been checked
                        cellshade[i]=0xff000000;
                        temp[i]=0;
                }
                for (int y = 1; y<499; y++){
                        for (int x = 1; x<499; x++){
```



```
                        if ((checked[y*500 + x]==false)&&(new Color(edge
[y*500+x]).getRed()!=0)){
                                Q.add(new Point(x,y));
                                ticker=true;
                        }
                        while(!Q.isEmpty()){
                                Point p = Q.remove();
                                //System.out.println(p.x + ":"+p.y);
                                checked[p.y*500 + p.x]=true;
                                if ((p.x>0) && (p.x < 499 && p.y >0) && (p.y < 499)){
                                        temp[p.y*500+p.x]=-1;
                                        if (checked[p.y*500 + p.x+1]==false && (new
Color(edge[p.y*500+p.x+1]).getRed()!=0)){
                                                Q.add(new Point(p.x+1, p.y));
                                                checked[p.y*500 + p.x+1]=true;
                                        }
                                        if (checked[p.y*500 + p.x-1]==false && (new
Color(edge[p.y*500+p.x-1]).getRed()!=0)){
                                                Q.add(new Point(p.x-1 ,p.y));
                                                checked[p.y*500 + p.x-1]=true;
                                        }
                                        if (checked[(p.y+1)*500 + p.x]==false && (new
Color(edge[(p.y+1)*500+p.x]).getRed()!=0)){
                                                Q.add(new Point(p.x, p.y+1));
                                                checked[(p.y+1)*500 + p.x]=true;
                                        }
                                        if (checked[(p.y-1)*500 + p.x]==false && (new
Color(edge[(p.y-1)*500+p.x]).getRed()!=0)){
                                                Q.add(new Point(p.x, p.y-1));
                                                checked[(p.y-1)*500 + p.x]=true;
                                        }
                                }
                        }
                        if (Q.isEmpty() && ticker==true){
                                averageFill(orig, edge, temp, cellshade);
                                ticker=false;
                        }
                }
        }



        return cellshade;
    }

    private void averageFill(int[] orig, int[] orig_pix, int[] temp_pix, int[]
cellshade){
        int r = 0;
        int g = 0;
        int b = 0;
```



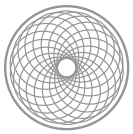
```
		int[] RGB = new int[temp_pix.length];
		int counter = 0; //Counters the amount of pixels we found
		for (int i=0; i<temp_pix.length-1;i++){
			if (temp_pix[i]==-1){
				counter++;
				r+=new Color(orig[i]).getRed();
				g+=new Color(orig[i]).getGreen();
				b+=new Color(orig[i]).getBlue();
			}
		}

		for (int i=0; i<temp_pix.length-1;i++){
			if (temp_pix[i]==-1){
				//Averages the colour of all the pixels and store it in the
"RGB" array;
				RGB[i]=new Color(r/counter,g/counter,b/counter,255).getRGB
();
				cellshade[i]=RGB[i];
				temp_pix[i]=0;
			}
		}
	}
	/*
	public int[] Fill(int[] pix, int[] orig){
		int[] cellshade = new int[pix.length];;
		int[] temp = new int[pix.length];
		boolean[] checked = new boolean[pix.length];
		for (int i=0; i<checked.length-1;i++){
			checked[i]=false; //This array holds which pixels have already
been checked
			cellshade[i]=0xff000000;
		}
		//500 is the width and height of the image, scaled when the image was
loaded
		//Iterate through every single pixel
		for (int y=0; y<500; y++){
			for(int x=0;x<500;x++){
				//Finding non-checked pixels and non-black pixel regions to
"fill"
				if ((checked[y*500 + x]==false && new Color(pix[y*500 +
x]).getRed()!=0)){
					temp=fillSelect(pix, temp, x, y, new Color(pix[y*500
+ x]).getRGB(), checked);
					averageFill(orig, pix, temp, cellshade);
					checked[y*500 + x]=true;
				} else { checked[y*500 + x]=true; }
			}
		}
		return cellshade;
	}
```



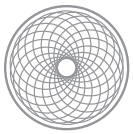
```
	private int[] fillSelect(int[] orig_pix, int[] temp_pix, int tx, int ty, int
RGB, boolean[] checked){

		//Fills all the way to the left until it touches an edge
		int fillL = tx;
		do{
			temp_pix[ty*500 + fillL]=-1;
			checked[ty*500 + fillL]=true;
			fillL--;
		} while (fillL >=0 && new Color(orig_pix[ty*500 + fillL]).getRGB()==RGB);
		fillL++;

		//Fill all the way to the right until it touches an edge
		int fillR = tx;
		do{
			temp_pix[ty*500 + fillR]=-1;
			checked[ty*500 + fillR]=true;
			fillR++;
		} while (fillR < 500 && new Color(orig_pix[ty*500 + fillR]).getRGB()
==RGB);
		fillR--;
		System.out.println("fillR: " + fillR + " fillL: "+fillL);
		//fills up and down (with more recursion)
		for (int i=fillL; i<=fillR; i++){
			//System.out.println("TX: " +i + " TY: "+ty);
			if (ty > 0 && new Color(orig_pix[(ty-1)*500 + i]).getRGB()==RGB){
				fillSelect(orig_pix, temp_pix, i, ty-1, RGB, checked);
			}
			if (ty < 500 && new Color(orig_pix[(ty+1)*500 + i]).getRGB()==RGB)
{
				fillSelect(orig_pix, temp_pix, i, ty+1, RGB, checked);
			}
		}
		return temp_pix;
	}*/
}
```

-SobelDemo.Java

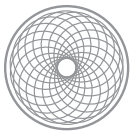
```
import java.awt.*;
import java.awt.image.*;
import java.applet.*;
import java.net.*;
import java.io.*;
import java.lang.Math;
import java.util.*;
import java.awt.event.*;
import javax.swing.*;
```



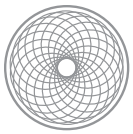
```
import javax.imageio.*;
import javax.swing.event.*;

public class SobelDemo extends JApplet {

	// Initializations
	Image edgeImage, accImage, outputImage;
	MediaTracker tracker = null;
	PixelGrabber grabber = null;
	int width = 0, height = 0;
	// filelist
	String files[] =
{
"70s.jpg","blastoise.gif","optimus.jpg","woman2.gif","iphone.jpg","cocacola.jpg","luon
go.jpg","artoo.jpg","snes.jpg", "butterfly.jpg","baconator.jpg","oprah.jpg",
"buzz.jpg","van.jpg","dinosaur.jpg","harper.jpg","church.jpg", "whiteStripes.jpg",
                    "bird.jpg", "Mario_and_Luigi.jpg" };
	javax.swing.Timer timer;
	// slider constraints
	static final int TH_MIN = 0;
	static final int TH_MAX = 255;
	static final int TH_INIT = 245;
	int threshold = TH_INIT;
	boolean thresholdActive = false;

	// more inits
	int imageNumber = 0;
	static int progress = 0;
	public int orig[] = null;

	// picture storage for all preloads
	Image pictures[] = new Image[files.length];

	// layout components
	JProgressBar progressBar;
	JPanel controlPanel, imagePanel, progressPanel;
	JLabel origLabel, outputLabel, comboLabel, sigmaLabel, thresholdLabel,
	processing;
	JSlider thresholdSlider;
	JButton thresholding;
	JComboBox imSel;

	// Sobel class
	static Sobel edgedetector;

	// Applet init function
	public void init() {
		this.setSize(1200, 700);
		// tracker for images
		tracker = new MediaTracker(this);
```



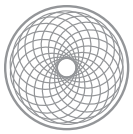
```
// get all specified files and add to image holding array
for (int i = 0; i < files.length; i++) {
      pictures[i] = getImage(this.getCodeBase(), files[i]);
      pictures[i] = pictures[i].getScaledInstance(500, 500,
                  Image.SCALE_SMOOTH);
      tracker.addImage(pictures[i], i);
}
// wait for loads
try {
      tracker.waitForAll();
} catch (InterruptedException e) {
      System.out.println("error: " + e);
}

// setup applet organization
Container cont = getContentPane();
cont.removeAll();
cont.setBackground(Color.black);
cont.setLayout(new BorderLayout());

controlPanel = new JPanel();
controlPanel.setLayout(new GridLayout(2, 3, 15, 0));
controlPanel.setBackground(new Color(55, 55, 55));
imagePanel = new JPanel();
imagePanel.setBackground(new Color(55, 55, 55));
progressPanel = new JPanel();
progressPanel.setBackground(new Color(150, 150, 150));
progressPanel.setLayout(new GridLayout(2, 1));

comboLabel = new JLabel("IMAGE");
comboLabel.setForeground(Color.white);
comboLabel.setHorizontalAlignment(JLabel.CENTER);
controlPanel.add(comboLabel);
sigmaLabel = new JLabel("");
sigmaLabel.setHorizontalAlignment(JLabel.CENTER);
controlPanel.add(sigmaLabel);
thresholdLabel = new JLabel("Threshold Value = " + TH_INIT);
thresholdLabel.setForeground(Color.white);
thresholdLabel.setHorizontalAlignment(JLabel.CENTER);
controlPanel.add(thresholdLabel);

processing = new JLabel("Processing...");
processing.setHorizontalAlignment(JLabel.LEFT);
progressBar = new JProgressBar(0, 100);
progressBar.setValue(0);
progressBar.setStringPainted(true); // get space for the string
progressBar.setString(""); // but don't paint it
progressPanel.add(processing);
progressPanel.add(progressBar);
```



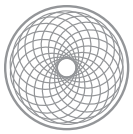
```
// get data about images
width = pictures[imageNumber].getWidth(null);
height = pictures[imageNumber].getHeight(null);

imSel = new JComboBox(files);
imageNumber = imSel.getSelectedIndex();
// listen for new selections from combo box and update when it happens
imSel.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		imageNumber = imSel.getSelectedIndex();
		origLabel.setIcon(new ImageIcon(pictures[imageNumber]));
		processImage();
	}
});
controlPanel.add(imSel, BorderLayout.PAGE_START);

// updater for progress bar on timer
timer = new javax.swing.Timer(100, new ActionListener() {
	public void actionPerformed(ActionEvent evt) {
		progressBar.setValue(edgedetector.getProgress());
	}
});

// original image section
origLabel = new JLabel("Original Image", new ImageIcon(
		pictures[imageNumber]), JLabel.CENTER);
origLabel.setVerticalTextPosition(JLabel.BOTTOM);
origLabel.setHorizontalTextPosition(JLabel.CENTER);
origLabel.setForeground(Color.white);
imagePanel.add(origLabel);

// edge detected image section
outputLabel = new JLabel("Edge Detected", new ImageIcon(
		pictures[imageNumber]), JLabel.CENTER);
outputLabel.setVerticalTextPosition(JLabel.BOTTOM);
outputLabel.setHorizontalTextPosition(JLabel.CENTER);
outputLabel.setForeground(Color.white);
imagePanel.add(outputLabel);

// activate threshold
thresholding = new JButton("Thresholding Off");
// thresholding.setVerticalTextPosition(AbstractButton.BOTTOM);
// thresholding.setHorizontalTextPosition(AbstractButton.CENTER);
thresholding.setBackground(Color.RED);
thresholding.addActionListener(new ActionListener() {
	public void actionPerformed(ActionEvent e) {
		if (thresholdActive == true) {
			thresholdActive = false;
			thresholding.setBackground(Color.RED);
			thresholding.setText("Thresholding Off");
```



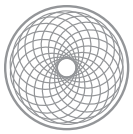
```
                        thresholdSlider.setEnabled(false);
                    } else {
                        thresholdActive = true;
                        thresholding.setBackground(Color.GREEN);
                        thresholding.setText("Thresholding ON");
                        thresholdSlider.setEnabled(true);
                    }
                    processImage();
                }
        });
        controlPanel.add(thresholding);

        // slider for theshold value
        thresholdSlider = new JSlider(JSlider.HORIZONTAL, TH_MIN, TH_MAX,
                    TH_INIT);
        thresholdSlider.addChangeListener(new thresholdListener());
        thresholdSlider.setMajorTickSpacing(40);
        thresholdSlider.setMinorTickSpacing(10);
        thresholdSlider.setPaintTicks(true);
        thresholdSlider.setPaintLabels(true);
        thresholdSlider.setBackground(new Color(192, 204, 226));
        controlPanel.add(thresholdSlider);

        // add elements to content
        cont.add(controlPanel, BorderLayout.NORTH);
        cont.add(imagePanel, BorderLayout.CENTER);
        cont.add(progressPanel, BorderLayout.SOUTH);

        // create initial transformed image
        processImage();

}

// listener updates processed image on change to theshold slider
class thresholdListener implements ChangeListener {
        public void stateChanged(ChangeEvent e) {
                JSlider source = (JSlider) e.getSource();
                if (!source.getValueIsAdjusting()) {
                        System.out.println("threshold=" + source.getValue());
                        threshold = source.getValue();
                        thresholdLabel
                        .setText("Threshold Value = " + source.getValue());
                        processImage();
                }
        }
}

// bitwise operation compares input to threshold value. if over, pixel =
// white, if not, pixel = black
public int[] threshold(int[] original, int value) {
```



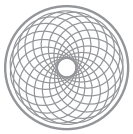
```
	for (int x = 0; x < original.length; x++) {
		if ((original[x] & 0xff) >= value)
			original[x] = 0xffffffff;
		else
			original[x] = 0xff000000;
	}
	return original;
}

// generates output image
private void processImage() {
	// get values from original input image
	orig = new int[width * height];
	PixelGrabber grabber = new PixelGrabber(pictures[imageNumber], 0, 0,
			width, height, orig, 0, width);
	try {
		grabber.grabPixels();
	} catch (InterruptedException e2) {
		System.out.println("error: " + e2);
	}
	// ready progress bar
	progressBar.setMaximum(width - 4);
	// state progress text
	processing.setText("Processing...");
	// inputs not enabled during processing
	thresholdSlider.setEnabled(false);
	thresholding.setEnabled(false);
	imSel.setEnabled(false);

	// new Sobel class
	edgedetector = new Sobel();

	// begin update timer for progress bar
	timer.start();

	new Thread() {
		public void run() {
			// initialize edge detector with input image
			edgedetector.init(orig, width, height);
			// get result of sobel process
			int[] res = edgedetector.inverted();
			// transform result by threshold operation if threshold is
			// activated
			if (thresholdActive == true) {
				res = threshold(res, threshold);
			}
			//res = SubPix(orig, res);
			// make final image
			//int[] storedEdges = edgedetector.process();
		//	Fill filler = new Fill();
```



```
                        //res = fill.Gamma(res);

                //      res = filler.queueFill(res, orig);
                        /*****************************************************
                         *   THIS IS WHERE THE OTHER CHUNK WOULD BE APPLIED   *
                         *****************************************************/

                        final Image output = createImage(new MemoryImageSource
(width,
                                        height, res, 0, width));
                        // run updates and return functionality to inputs
                        SwingUtilities.invokeLater(new Runnable() {
                                public void run() {
                                        outputLabel.setIcon(new ImageIcon(output));
                                        processing.setText("Done");
                                        if (thresholdActive == true) {
                                                thresholdSlider.setEnabled(true);
                                        }
                                        thresholding.setEnabled(true);
                                        imSel.setEnabled(true);
                                }
                        });
                }
        }.start();
    }

    public int[] SubPix(int[] img1, int[] img2) {
            int[] pix = null;
            PixelGrabber grabber = new PixelGrabber(pictures[imageNumber], 0, 0,
                            width, height, pix, 0, width);
            try {
                    grabber.grabPixels();
            } catch (InterruptedException e2) {
                    System.out.println("error: " + e2);
            }
            ColorModel m_colorModel = grabber.getColorModel();

            int[] temp = new int[img1.length];
            for (int cnt = 0; cnt < img1.length; cnt++) {
                    int red = m_colorModel.getRed(img1[cnt]); // get red color data
                    int green = m_colorModel.getGreen(img1[cnt]); // get green color
                    // data
                    int blue = m_colorModel.getBlue(img1[cnt]); // get blue color data
                    int edginess = m_colorModel.getRed(img2[cnt]);

                    int subRed = Math.max(0, red - edginess);

                    int subGreen = Math.max(0, green - edginess);

                    int subBlue = Math.max(0, blue - edginess);
```



```
                    temp[cnt] = 0xff000000 | ((int) subRed << 16 | (int) subGreen << 8
| (int) subBlue);
                }
                return temp;
        }

}
```



Conclusion

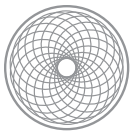
Alannah

I always thought that the artwork in the movie "A Scanner Darkly" was pretty interesting. As a result, I thought the topic our group chose was relevant to my interests. Although our project required slightly more work than expected, our group managed to pull it off. Our project is not animated, but it uses cel-shading to convert an image into the artwork similar to that in the movie. I was able to contribute with research, Java layouts, and the fill(averaging) method in the code. Our code had five main steps - edge detection, inversion, white detection, colour filling-in, and border adding at the end. The main problem I had with the fill method was control of the colours and where in the image they went. For example, if the previous steps like edge detection and white detection weren't very accurate, then the additional colour filling would most likely lead to spilling of the wrong colours outside or inside of the wrong objects. This project helped me realize the importance of accuracy in every single step in the digital compositing process.

Peter

I was in charge of implementing a flood fill algorithm that would fill a region of pixels, mark that as a selection, and then pass the selection to the averageFill function to have colours of the selection averaged and painted back onto a new “cellshade” array. The flood fill algorithm was not as easy as I originally thought, because the original recursive method I approached the problem with was not a very good solution as Java applets have a very small stack size, so even with our relatively small (500 x 500) images, it would get stack overflow errors. This means that I had to use Queues, but since Java doesn’t have explicit Queues, I had to use LinkedLists to emulate Queues instead.

The way queueFill function (the flood fill) works is that there is a for look traversing through the entire image one pixel at a time, and there is a secondary Boolean array of the same size as the image that stores whether or not the current pixel has been checked. If the traversal loop finds a white area, we add that to the Queue stack, and if the Queue stack is NOT full, we check the neighbours of all the nodes in the queue stack in 4 directions. So sooner or later the entire region will be “filled”, and as soon as the region is found, the result is passed to averageFill function as mentioned above. Since the traversal for loop will go through every single pixel, the algorithm will fill all white-spots until every single white pixel has been filled with a colour pixel.



Ethan

Being in charge of the edge detection algorithm provided me with a variety of new knowledge during my research for potential solutions. The amount of edge detection algorithms and the pros and cons of each was highly impressive, as I previously had no idea just how much research and breadth of attempts there were on the topic.

Through implementing our final choice of a Sobel operator and collaborating on the various other portions I learned an extensive amount about vector, matrix, and searching mathematic techniques. It also made me appreciate much more the work that goes into any rotoscoped piece of work. The simplicity of rendering in cel-shading from a 3d modelling program (which I had past experience with) has become much less of a mystery and fascination compared to the effort involved in generating similar results out of pre-existing images. I am hopeful I will be able to look further in the future into techniques such as keyframe rotoscoping in video production.

https://netfiles.uiuc.edu/ssundarm/www/roto/roto_seg.html

http://grail.cs.washington.edu/projects/rotoscoping/

http://en.wikipedia.org/wiki/Sobel_operator

http://en.wikipedia.org/wiki/Canny_edge_detector