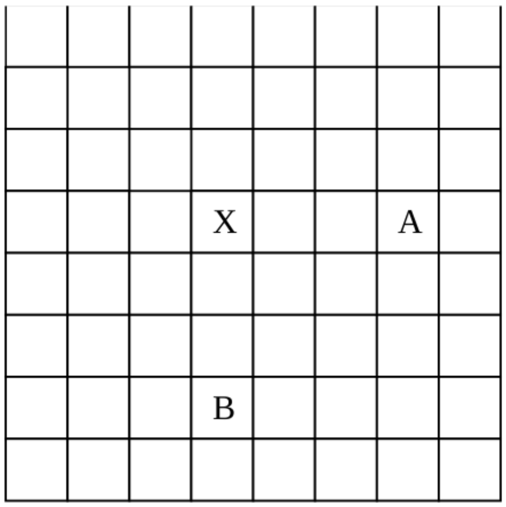
Name: Rodrigo Ignacio Rojas Garcia
Course: ECE 4310
Lab #8

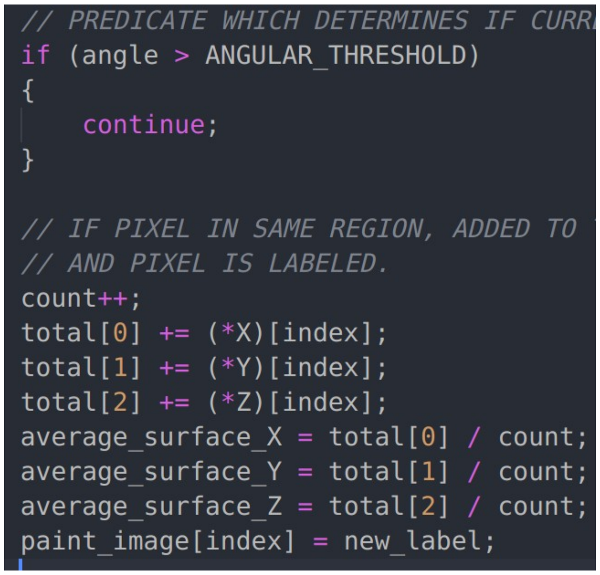
Range Image Segmentation

In this project the student was to segment a range image based upon surface normals. The student was to use a PPM image called “chair-range.ppm” and C code regarding conversion of pixels into 3D coordinates and Region Grow. The segmentation process used the image grid for grouping of pixels, but used the 3D coordinates for calculating the surface normals for region predicates. The laboratory was successfully completed by completing the following steps:

1. Threshold Image
    1. The image “chair-range.ppm” was thresholded at a value of **137 (value of THRESHOLD on C Code)** with the purpose of removing the background and only leaving the floor and chair in the image.
2. Obtain 3D Coordinates
    1. The C-code provided by Dr. Hoover was modified and implemented in order to obtain the 3D coordinates from pixels from the “chair-range.ppm” image. It should be noted that the slant type was assumed to be of scan-direction downward.
3. Calculate Surface Normals
    1. Surface normals were obtained by using the “chair-range.ppm” image and calculating it by taking the cross product of (B – X) x (A – X), where A and B are both 3D coordinates of those pixels (reference image below). The distance chosen between pixels for cross products were of value **3 (value of PIXEL_WIDTH on C code).**



4. Region Growing
    1. The C-code provided by Dr. Hoover regarding queue-based region growing was modified and used to segment regions of the thresholded PPM image. The C-code was modified in order to be able to join a pixel based on the predicate that is within a threshold of the average orientation of pixels already on the region.
    2. The seed pixels for region growing were found by identifying a complete 5x5 window of unlabeled (and not masked out) of still-unlabeled region. The process followed that if any pixel within the 5x5 window was masked out already or already labeled in the 5x5 region, then the pixel could not be considered a seed for a new region. Region growing stopped when there were no more possible seed pixels in the image.
    3. The process of finding a seed pixel was broken down into following steps:
        1. Look through whole thresholded image **(starting at pixel at ROW 2, COLUMN 2).**
        2. Search for seed pixel in 5x5 window
            1. Calculate Angular difference using the dot product
            2. **Predicate: If the angular difference is less than Angular Difference threshold then:**
                1. **Add value to the total Surface Normal for each X, Y, and Z**
                2. **Calculate new Average Surface Normal for each X, Y, and Z**
                3. **Label Pixel**
                4. **Code:**



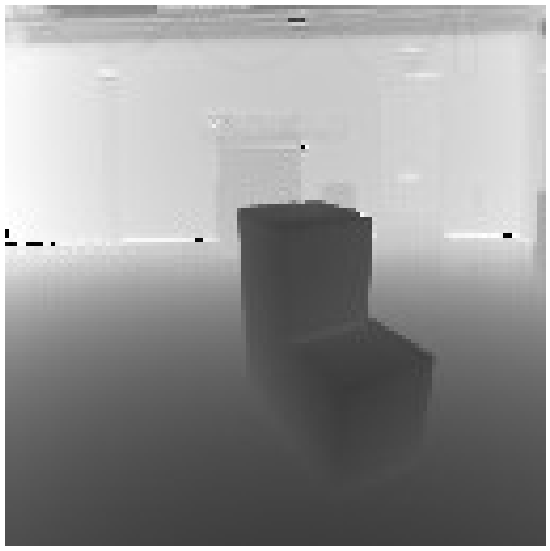
```
// PREDICATE WHICH DETERMINES IF CURR
if (angle > ANGULAR_THRESHOLD)
{
    continue;
}

// IF PIXEL IN SAME REGION, ADDED TO
// AND PIXEL IS LABELED.
count++;
total[0] += (*X)[index];
total[1] += (*Y)[index];
total[2] += (*Z)[index];
average_surface_X = total[0] / count;
average_surface_Y = total[1] / count;
average_surface_Z = total[2] / count;
paint_image[index] = new_label;
```

                5. **It should be noted that the ANGULAR_THRESHOLD value is of 0.65**
        3. Repeat Process until all image is searched

All C Code can be seen at the end of the report.

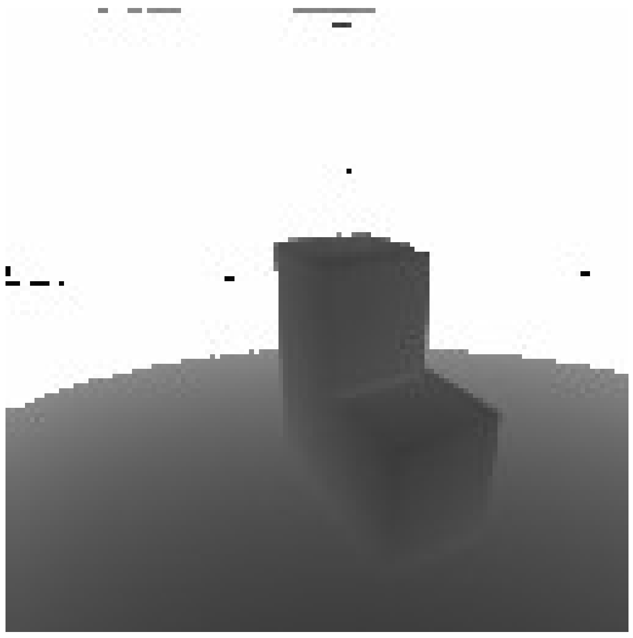
RESULTS

Original "chair-range.ppm" Image:



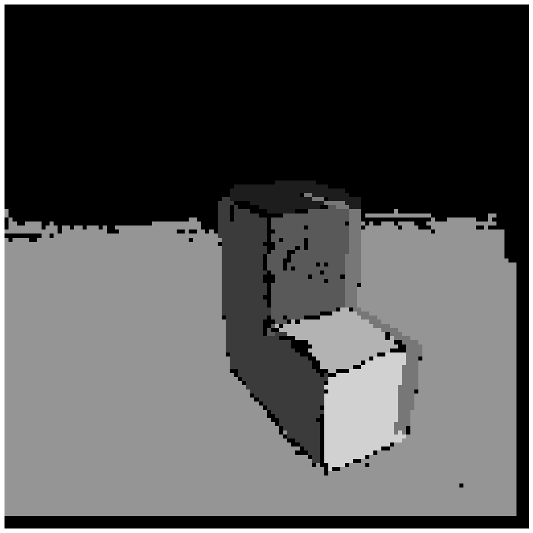
Thresholded "chair-range.ppm" Image:

Threshold value: 137



Segmented & Thresholded "chair-range.ppm" Image:

Threshold value: 137
Cross Product Distance: 3
Angular Difference Threshold: 0.65



Region	**Number of Pixels**	**Average Surface Normal (X)**	**Average Surface Normal (Y)**	**Average Surface Normal (Z)**
1	168	-2.912282	327.747653	-57.789212
2	786	-50.107734	0.919432	-8.377012
3	479	2.569452	-2.398113	-4.477929
4	440	50.852073	-1.701171	-15.616532
5	6844	-1.519162	28.897201	-8.941283
6	256	-0.973660	8.602109	-2.396540

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define MAXLENGTH 256
#define THRESHOLD 137
#define PIXEL_WITDH 3
#define MAX_QUEUE 10000
#define ANGULAR_THRESHOLD 0.65

// RETURNS PPM IMAGE
unsigned char *read_in_image(int rows, int cols, FILE *image_file)
{
    // VARIABLE DECLARATION SECTION
    unsigned char *image;

    image = (unsigned char *)calloc(rows * cols, sizeof(unsigned char));

    fread(image, sizeof(unsigned char), rows * cols, image_file);

    fclose(image_file);

    return image;
}

// CREATE AND SAVE FILE AS PPM IMAGE
void save_image(unsigned char *image, char *file_name, int rows, int cols)
{
    // VARIABLE DECLARATION SECTION
    FILE * file;
    file = fopen(file_name, "w");
    fprintf(file, "P5 %d %d 255\n", cols, rows);
    fwrite(image, rows * cols, sizeof(unsigned char), file);
    fclose(file);
}

// THRESHOLD IMAGE TO GET RID OF BACKGROUND
unsigned char *threshold_image(int rows, int cols, unsigned char *image)
{
    // VARIABLE DECLARATION SECTION
    int i;
    unsigned char *output_image;

    // ALLOCATE MEMORY
    output_image = (unsigned char *)calloc(rows * cols, sizeof(unsigned char));

    // THRESHOLD IMAGE
    for (i = 0; i < (rows * cols); i++)
    {
        if (image[i] > THRESHOLD)
        {
            output_image[i] = 255;
        }
        else
        {
            output_image[i] = image[i];
        }
    }

    save_image(output_image, "thresholded.ppm", rows, cols);
    return output_image;
}

// CALCULATES THE X,Y, AND Z COORDINATES OF IMAGE
void calc_3Dpoints(unsigned char *image, int rows, int cols, double **X, double **Y,
double **Z)
{
    // VARIABLE DECLARATION SECTION
    int i, j;
    double x_angle, y_angle, distance;
    double cp[7];
    double slant_correction;
    int index = 0;

    // ALLOCATE MEMORY
    *X = calloc(rows * cols, sizeof(double *));
    *Y = calloc(rows * cols, sizeof(double *));
    *Z = calloc(rows * cols, sizeof(double *));

    // COORDINATES ALGORITHM
    cp[0]=1220.7;       /* horizontal mirror angular velocity in rpm */
    cp[1]=32.0;     /* scan time per single pixel in microseconds */
    cp[2]=(cols/2)-0.5;     /* middle value of columns */
    cp[3]=1220.7/192.0; /* vertical mirror angular velocity in rpm */
    cp[4]=6.14;     /* scan time (with retrace) per line in milliseconds */
    cp[5]=(rows/2)-0.5;     /* middle value of rows */
    cp[6]=10.0;     /* standoff distance in range units (3.66cm per r.u.) */

    cp[0]=cp[0]*3.1415927/30.0; /* convert rpm to rad/sec */
    cp[3]=cp[3]*3.1415927/30.0; /* convert rpm to rad/sec */
    cp[0]=2.0*cp[0];        /* beam ang. vel. is twice mirror ang. vel. */
    cp[3]=2.0*cp[3];        /* beam ang. vel. is twice mirror ang. vel. */
    cp[1]/=1000000.0;       /* units are microseconds : 10^-6 */
    cp[4]/=1000.0;          /* units are milliseconds : 10^-3 */

    for (i = 0; i < rows; i++)
    {
        for (j = 0; j < cols; j++)
        {
            slant_correction = cp[3] * cp[1] * ((double)j - cp[2]);
            x_angle = cp[0] * cp[1] * ((double)j - cp[2]);
            y_angle = (cp[3] * cp[4] * (cp[5] - (double)i)) + (slant_correction *
1); /*  + slant correction */
            index = (i * cols) + j;
            distance = (double)image[index] + cp[6];
            (*Z)[index] = sqrt((distance * distance) / (1.0 + (tan(x_angle) *
tan(x_angle)) + (tan(y_angle) *tan(y_angle))));
            (*X)[index] = tan(x_angle) * (*Z)[index];
            (*Y)[index] = tan(y_angle) * (*Z)[index];
        }
    }
}

void calc_surface_normal(unsigned char *image, unsigned char *threshold_image, int
rows, int cols, double **S_X, double **S_Y, double **S_Z)
{
    // VARIABLE DECLARATION SECTION
    int i, j;
    int index1, index2, index3;
    double *X, *Y, *Z;
    double x1, x2, y1, y2, z1, z2;
```

```c
    // Obtains 3D points in image
    calc_3Dpoints(image, rows, cols, &X, &Y, &Z);

    // ALLOCATION OF MEMORY
    *S_X = calloc(rows * cols, sizeof(double *));
    *S_Y = calloc(rows * cols, sizeof(double *));
    *S_Z = calloc(rows * cols, sizeof(double *));

    for (i = 0; i < (rows - PIXEL_WITDH); i++)
    {
        for (j = 0; j < (cols - PIXEL_WITDH); j++)
        {
            index1 = (i * cols) + j;
            index2 = ((i + PIXEL_WITDH) * cols) + j;
            index3 = (i * cols) + (j + PIXEL_WITDH);

            // VECTOR A
            x1 = X[index3] - X[index1];
            y1 = Y[index3] - Y[index1];
            z1 = Z[index3] - Z[index1];

            // VECTOR B
            x2 = X[index2] - X[index1];
            y2 = Y[index2] - Y[index1];
            z2 = Z[index2] - Z[index1];

            // CROSS PRODUCT CALCULATION
            (*S_X)[index1] = (y1 * z2) - (z1 * y2);
            (*S_Y)[index1] = ((x1 * z2) - (z1 * x2)) * -1;
            (*S_Z)[index1] = (x1 * y2) - (y1 * x2);
        }
    }
}

// REGION GROW
int queue_paint_full(unsigned char *image, unsigned char *paint_image, int rows, int
cols,
                    int current_row, int current_col,
                    int paint_over_label, int new_label,
                    double **X, double **Y, double **Z)
{
    // VARIABLE DECLARATION SECTION
    int count;
    int r2,c2;
    int queue[MAX_QUEUE],qh,qt;
    int index;
    double dot_product;
    double average_surface_X, average_surface_Y, average_surface_Z;
    double angle = 0;
    double distance_A = 0;
    double distance_B = 0;
    double total[3] = {0, 0, 0};

    count = 0;

    // STARTING AVERAGE SURANCE NORMALS (AT CURRENT PIXEL VALUE)
    index = (current_row * cols) + current_col;
    average_surface_X = (*X)[index];
    average_surface_Y = (*Y)[index];
    average_surface_Z = (*Z)[index];

    // STARTING TOTAL SUM VALUES (AT CURRENT PIXEL VALUE)
    total[0] = (*X)[index];
    total[1] = (*Y)[index];
    total[2] = (*Z)[index];

    queue[0] = index;
    qh = 1; /* queue head */
    qt = 0; /* queue tail */
    count = 1;

    while (qt != qh)
    {
        for (r2 = -1; r2 <= 1; r2++)
        {
            for (c2 = -1; c2 <= 1; c2++)
            {

                index = (queue[qt] / cols + r2) * cols + queue[qt] % cols + c2;

                // PREDICATES TO SKIP
                if (r2 == 0  &&  c2 == 0)
                {
                    continue;
                }
                if ((queue[qt] / cols + r2) < 0  ||  (queue[qt] / cols + r2) >= rows
- PIXEL_WITDH  ||
                (queue[qt] % cols + c2) < 0  ||  (queue[qt] % cols + c2) >= cols -
PIXEL_WITDH)
                {
                    continue;
                }
                if (paint_image[index] != 0)
                {
                    continue;
                }

                // CALCULATES DOT PRODUCT
                dot_product = (average_surface_X * (*X)[index]) + (average_surface_Y
* (*Y)[index]) + (average_surface_Z * (*Z)[index]);

                // CALCULATES ANGLE
                distance_A = sqrt( pow(average_surface_X, 2) + pow(average_surface_Y,
2) + pow(average_surface_Z,2) );
                distance_B = sqrt( pow((*X)[index], 2) + pow((*Y)[index],2 ) +
pow((*Z)[index], 2) );
                angle = acos(dot_product / (distance_A * distance_B));

                // PREDICATE WHICH DETERMINES IF CURRENT PIXEL IS IN THE SAME REGION
                if (angle > ANGULAR_THRESHOLD)
                {
                    continue;
                }

                // IF PIXEL IN SAME REGION, ADDED TO THE REGION, AVERAGE SURFANCE FOR
X, Y, AND Z CALCULATED,
                // AND PIXEL IS LABELED.
                count++;
                total[0] += (*X)[index];
                total[1] += (*Y)[index];
                total[2] += (*Z)[index];
                average_surface_X = total[0] / count;
                average_surface_Y = total[1] / count;
                average_surface_Z = total[2] / count;
                paint_image[index] = new_label;

                queue[qh] = (queue[qt] / cols + r2) * cols+ queue[qt] % cols + c2;

                qh = (qh + 1) % MAX_QUEUE;

                if (qh == qt)
                {
                    printf("Max queue size exceeded\n");
                    exit(0);
                }
            }
        }
        qt = (qt + 1 )% MAX_QUEUE;
    }

    printf("X: %lf, Y: %lf, Z: %lf\n", average_surface_X, average_surface_Y,
average_surface_Z);
    return count;
}


int main(int argc, char *argv[])
{
    // VARIABLE DECLARATION SECTION
    FILE *image_file;
    int IMAGE_ROWS, IMAGE_COLS, IMAGE_BYTES, i, j, r, c, regions, k;
    int new_label = 30;
    int index = 0;
    int count = 0;
    regions = 0;
    char file_header[MAXLENGTH];
    unsigned char *input_image, *thresholded_image, *paint_image;
    double *S_X, *S_Y, *S_Z;
    int valid = 0;

    if (argc != 2)
    {
        printf("Usage: ./executable image_file\n");
        exit(1);
    }

    image_file = fopen(argv[1], "rb");
    if (image_file == NULL)
    {
        printf("Error, could not read PPM image file\n");
        exit(1);
    }
    fscanf(image_file, "%s %d %d %d\n", file_header, &IMAGE_COLS, &IMAGE_ROWS,
&IMAGE_BYTES);
    if ((strcmp(file_header, "P5") != 0) || (IMAGE_BYTES != 255))
    {
        printf("Error, not a grey-scale 8-bit PPM image\n");
        fclose(image_file);
        exit(1);
    }

    /* ALLOCATES MEMORY AND READS IN INPUT IMAGE */
    input_image = read_in_image(IMAGE_ROWS, IMAGE_COLS, image_file);

    /* THRESHOLD IMAGE */
    thresholded_image = threshold_image(IMAGE_ROWS, IMAGE_COLS, input_image);
```

```c
    /* CALCULATE SURFACE NORMALS */
    calc_surface_normal(input_image, thresholded_image, IMAGE_ROWS, IMAGE_COLS, &S_X,
&S_Y, &S_Z);

    /* ALLOCATE MEMORY FOR OUTPUT IMAGE WHICH WILL BE USED FOR REGION GROW */
    paint_image = calloc(IMAGE_ROWS * IMAGE_COLS, sizeof(unsigned char));

    /* REGION GROW  */
    for (i = 2; i < IMAGE_ROWS - PIXEL_WITDH; i++)
    {
        for (j = 2; j < IMAGE_COLS - PIXEL_WITDH; j++)
        {
            valid = 1;
            for (r = -2; r < 3; r++)
            {
                for (c = -2; c < 3; c++)
                {
                    index = ((i + r) * IMAGE_COLS) + (j + c);
                    if (thresholded_image[index] == 255 || paint_image[index] != 0)
                    {
                        valid = 0;
                    }
                }
            }
            if (valid == 1)
            {
                count = queue_paint_full(input_image, paint_image, IMAGE_ROWS,
IMAGE_COLS, i, j, 255, new_label, &S_X, &S_Y, &S_Z);
                // IF REGION NOT BIG ENOUGH, LABELING IS RESETTED AND NOT COUNTED
                if (count < 100)
                {
                    for (k = 0; k < (IMAGE_ROWS * IMAGE_COLS); k++)
                    {
                        if (paint_image[k] == new_label)
                        {
                            paint_image[k] = 0;
                        }
                    }
                }
                else
                {
                    regions++;
                    new_label += 30;
                    printf("Region: %d, Number of Pixels: %d\n", regions, count);
                }
            }

        }
    }

    save_image(paint_image, "paint.ppm", IMAGE_ROWS, IMAGE_COLS);

    return 0;
}
```