```c
#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;
```

```
    // 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);
```

```
    /* 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;
}
```