**William Benton**

**Lab 8: Range Image Segmentation**

**12/05/2019**

**Background and Implementation:**

The purpose of this lab was to take a gray scale range image in and segment the image into the different surfaces. This was done by first thresholding the original gray scale image. The thresh hold value I used was 137. If the pixel was above the value 137, the pixel was set to 255, otherwise, the pixel did not change. Then I took the original gray scale image and calculated the X, Y, and Z coordinates. This was done using code provided. The code provided used details specific to the camera used to take the picture. These values were stored in an array of structs. The struct stored the X, Y, and Z values. With the X, Y, and Z values were found, the surface normal calculations were done. This was done by looping through every pixel and looking at a pixel to the right and a pixel below the current pixel. The surface normal vector was calculated by subtracting the current pixel from the pixel below and subtracting the current pixel from the pixel to the right and then taking the cross product of each of those resulting vectors. This result was stored in another array of structs. The distance I chose to use was 3 pixels to the right and 3 pixels below. From here, the segmentation began. This was done via region growing. Previously, we implemented region growing, so I took this previous code and modified it. It began by looping through every pixel and looking at a 5x5 window around every pixel. If there was no pixel in the window that was taken out by thresholding and no pixel had already been written over, then the pixel could be used to start the region growing. The region growing code utilizes a queue to perform the region growing. The portion that was modified to handle this specific task was the predicate to join the region. The predicate to join the region was that the angle between the current surface normal vector and the average of the surface normal vectors currently in the region, had to be within a threshold. I chose to use 0.72 as the threshold. The angle was determined using the magnitude of the average vector and the magnitude of the current vector and calculating the dot product between both vectors. The formula to determine the angle is the dot product divided by the product of the magnitudes. The result of this was the angle in radians, that’s why the threshold is not in degrees. If it was within the threshold, it was added to the region. The average vector was printed out and the number of pixels in the region. If the region was less than 100 pixels, the region was erased. The resulting image was printed out in gray scale as well as a colored image.

**Results:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Region Number** | **X Average** | **Y Average** | **Z Average** | **Number of Pixels** |
| Region Removed | 0.112017 | 1.193703 | -4.314375 | 3 |
| 1 | -6.487198 | -331.474135 | -58.264375 | 165 |
| 2 | -51.327779 | -0.914586 | -8.544396 | 763 |
| 3 | 2.662467 | 2.420322 | -4.482366 | 466 |
| 4 | 103.44437 | 2.337080 | -27.563593 | 207 |
| 5 | -1.618351 | -28.782052 | -8.864264 | 6801 |
| 6 | -1.050807 | -8.514511 | -2.389239 | 254 |
| 7 | 2.750688 | 1.627939 | -4.829699 | 423 |

A screenshot of a cell phone

Description automatically generated

Figure : Threshold Image. Threshold Value: 137

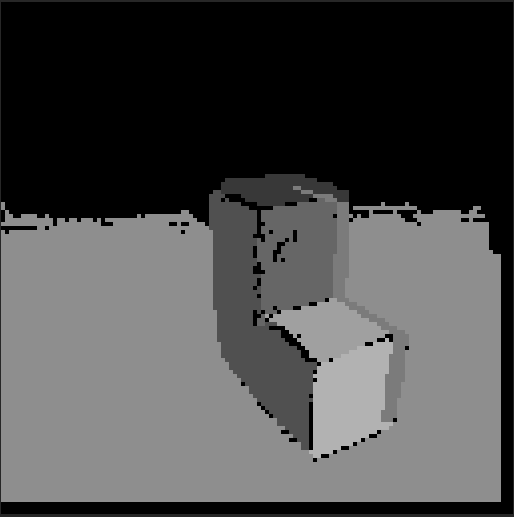


Figure : Grayscale Output Image

A close up of a logo

Description automatically generated

Figure : Colored Output Image

The segmentation turned out very well. It missed a few pixels but overall it was able to find all the surfaces very well. I spent some time testing different angle thresholds and the one I settled on looked the best.

**Code:**

*RangeSegmentation.h*

typedef struct cart\_coord\_s

{

double X;

double Y;

double Z;

} cart\_coord\_t;

#define MAX\_QUEUE 10000 /\* max perimeter size (pixels) of border wavefront \*/

#define SQR(x) ((x)\*(x))

#define ANGLE\_THRESH 0.72

void convert2XYZ(unsigned char \*RangeImage, cart\_coord\_t \*Coords, int ROWS, int COLS);

void WriteGreyScaleImage(unsigned char \*image, char \*filename, int ROWS, int COLS);

void SurfaceNormalCalc(cart\_coord\_t \*Coords, cart\_coord\_t \*SurfaceNormal, int ROWS, int COLS);

void threshold(unsigned char \*image, unsigned char \*output\_image, int thresh\_val, int ROWS, int COLS);

void RegionGrow(unsigned char \*image,

unsigned char \*labels,

int ROWS, int COLS,

int r, int c,

int paint\_over\_label,

int new\_label,

cart\_coord\_t \*SurfaceNormal,

int \*indices,

int \*count);

void WriteColorImage(unsigned char \*grayscale, int ROWS, int COLS);

*RangeSegmentation.c*

#include <stdlib.h>

#include <stdio.h>

#include <math.h>

#include <stdbool.h>

#include <string.h>

#include "RangeSegmentation.h"

int main ()

{

FILE \*fpt;

int ROWS, COLS, BYTES;

char header[320];

cart\_coord\_t \*Coords;

unsigned char \*RangeImage;

unsigned char \*ThreshholdImage;

cart\_coord\_t \*SurfaceNormal;

unsigned char \*OutputImage;

char \*output\_filename;

// Region Growing Variables

int i, j, k;

int r, c;

int \*indices;

bool valid\_seed;

int RegionSize, TotalRegions;

/\* read image \*/

if ((fpt=fopen("chair-range.ppm","rb")) == NULL)

{

printf("Unable to open range image for reading\n");

exit(0);

}

fscanf(fpt,"%s %d %d %d",header,&COLS,&ROWS,&BYTES);

if (strcmp(header,"P5") != 0 || BYTES != 255)

{

printf("Not a greyscale 8-bit PPM image\n");

exit(0);

}

RangeImage = (unsigned char \*)calloc(ROWS\*COLS,sizeof(unsigned char));

header[0] = fgetc(fpt); /\* read white-space character that separates header \*/

fread(RangeImage, 1, COLS\*ROWS, fpt);

fclose(fpt);

Coords = (cart\_coord\_t \*)calloc(ROWS\*COLS, sizeof(cart\_coord\_t));

ThreshholdImage = (unsigned char \*)calloc(ROWS\*COLS, sizeof(unsigned char));

SurfaceNormal = (cart\_coord\_t \*)calloc(ROWS\*COLS, sizeof(cart\_coord\_t));

OutputImage = (unsigned char \*)calloc(ROWS\*COLS, sizeof(unsigned char));

threshold(RangeImage, ThreshholdImage, 137, ROWS, COLS);

output\_filename = "threshold.ppm";

WriteGreyScaleImage(ThreshholdImage, output\_filename, ROWS, COLS);

convert2XYZ(RangeImage, Coords, ROWS, COLS);

SurfaceNormalCalc(Coords, SurfaceNormal, ROWS, COLS);

// Begin Region Growing

indices = (int \*)calloc(ROWS\*COLS, sizeof(int));

TotalRegions = 20;

for (i = 2; i < ROWS-2; i++)

{

for (j = 2; j < COLS-2; j++)

{

valid\_seed = true;

for (r = -2; r <= 2; r++)

{

for (c = -2; c <= 2; c++)

{

// Remove possibility of pixel becoming a seed if it has already been marked as a region

// or if it did not pass the threshold

if ((ThreshholdImage[(i+r)\*COLS+(j+c)] == 255) || OutputImage[(i+r)\*COLS+(j+c)] != 0)

{

valid\_seed = false;

}

}

}

if (valid\_seed)

{

TotalRegions += 20;

RegionGrow(RangeImage, OutputImage, i, j, ROWS, COLS, 0, TotalRegions, SurfaceNormal, indices, &RegionSize);

if (RegionSize < 100)

{

for (k = 0; k < RegionSize; k++)

{

OutputImage[indices[k]] = 0;

}

TotalRegions -= 20;

}

else

{

printf("Region Number: %d \t Number of Pixels: %d\n", (TotalRegions/20)-1, RegionSize);

}

}

}

}

output\_filename = "output.ppm";

WriteGreyScaleImage(OutputImage, output\_filename, ROWS, COLS);

WriteColorImage(OutputImage, ROWS, COLS);

return (0);

}

void WriteGreyScaleImage(unsigned char \*image, char \*filename, int ROWS, int COLS)

{

FILE \*fpt;

fpt = fopen(filename, "w");

fprintf(fpt, "P5 %d %d 255\n", COLS, ROWS);

fwrite(image, COLS\*ROWS, sizeof(unsigned char), fpt);

fclose(fpt);

return;

}

void SurfaceNormalCalc(cart\_coord\_t \*Coords, cart\_coord\_t \*SurfaceNormal, int ROWS, int COLS)

{

int r;

int c;

double x1;

double x2;

double y1;

double y2;

double z1;

double z2;

for (r = 0; r < ROWS - 3; r++)

{

for (c = 0; c < COLS - 3; c++)

{

x1 = Coords[r\*COLS+(c+3)].X - Coords[r\*COLS+c].X;

x2 = Coords[(r+3)\*COLS+c].X - Coords[r\*COLS+c].X;

y1 = Coords[r\*COLS+(c+3)].Y - Coords[r\*COLS+c].Y;

y2 = Coords[(r+3)\*COLS+c].Y - Coords[r\*COLS+c].Y;

z1 = Coords[r\*COLS+(c+3)].Z - Coords[r\*COLS+c].Z;

z2 = Coords[(r+3)\*COLS+c].Z - Coords[r\*COLS+c].Z;

SurfaceNormal[r\*COLS+c].X = (y1 \* z2) - (z1 \* y2);

SurfaceNormal[r\*COLS+c].Y = ((z1 \* x2) - (z2 \* x1)) \* -1;

SurfaceNormal[r\*COLS+c].Z = (x1 \* y2) - (y1 \* x2);

}

}

return;

}

void threshold(unsigned char \*image, unsigned char \*output\_image, int thresh\_val, int ROWS, int COLS)

{

int i;

for (i = 0; i < ROWS\*COLS; i++)

{

if (image[i] > thresh\_val)

{

output\_image[i] = 255;

}

else

{

// Don't overwrite values cause we'll need them later

output\_image[i] = image[i];

}

}

return;

}

/\*

\*\* Given an image, a starting point, and a label, this routine

\*\* paint-fills (8-connected) the area with the given new label

\*\* according to the given criteria (pixels close to the average

\*\* intensity of the growing region are allowed to join).

\*/

void RegionGrow(unsigned char \*image, /\* image data \*/

unsigned char \*labels, /\* segmentation labels \*/

int r, int c, /\* pixel to paint from \*/

int ROWS, int COLS,

int paint\_over\_label, /\* image label to paint over \*/

int new\_label, /\* image label for painting \*/

cart\_coord\_t \*SurfaceNormal, /\* Input: Surface Noraml Image \*/

int \*indices, /\* output: indices of pixels painted \*/

int \*count /\* output: count of pixels painted \*/

)

{

int r2, c2;

int queue[MAX\_QUEUE], qh, qt;

int curr\_pos;

cart\_coord\_t average, total; // Average of each direction and total in each direction

double dot\_product;

double tempX;

double tempY;

double tempZ;

double mag1; // Magnitude of average vector

double mag2; // Magnitude of curr position vector

double angle;

\*count = 0;

if (labels[r \* COLS + c] != paint\_over\_label)

return;

labels[r \* COLS + c] = new\_label;

average.X = SurfaceNormal[r\*COLS+c].X;

average.Y = SurfaceNormal[r\*COLS+c].Y;

average.Z = SurfaceNormal[r\*COLS+c].Z;

total.X = SurfaceNormal[r\*COLS+c].X;

total.Y = SurfaceNormal[r\*COLS+c].Y;

total.Z = SurfaceNormal[r\*COLS+c].Z;

if (indices != NULL)

indices[0] = r\*COLS + c;

queue[0] = r\*COLS + c;

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++)

{

if (r2 == 0 && c2 == 0)

continue;

if ((queue[qt] / COLS + r2) < 0 || (queue[qt] / COLS + r2) >= ROWS - 3 ||

(queue[qt] % COLS + c2) < 0 || (queue[qt] % COLS + c2) >= COLS - 3)

continue;

if (labels[(queue[qt] / COLS + r2)\*COLS + queue[qt] % COLS + c2] != paint\_over\_label)

continue;

curr\_pos = (queue[qt] / COLS + r2) \* COLS + (queue[qt] % COLS + c2);

/\* test criteria to join region \*/

tempX = (average.X \* SurfaceNormal[curr\_pos].X);

tempY = (average.Y \* SurfaceNormal[curr\_pos].Y);

tempZ = (average.Z \* SurfaceNormal[curr\_pos].Z);

dot\_product = tempX + tempY + tempZ;

mag1 = sqrt(SQR(average.X) + SQR(average.Y) + SQR(average.Z));

mag2 = sqrt(SQR(SurfaceNormal[curr\_pos].X) + SQR(SurfaceNormal[curr\_pos].Y) + SQR(SurfaceNormal[curr\_pos].Z));

angle = acos(dot\_product / (mag1 \* mag2));

if (angle > ANGLE\_THRESH)

continue;

labels[(queue[qt] / COLS + r2)\*COLS + queue[qt] % COLS + c2] = new\_label;

if (indices != NULL)

indices[\*count] = (queue[qt] / COLS + r2)\*COLS + queue[qt] % COLS + c2;

total.X += SurfaceNormal[curr\_pos].X;

total.Y += SurfaceNormal[curr\_pos].Y;

total.Z += SurfaceNormal[curr\_pos].Z;

(\*count)++;

average.X = total.X / (\*count);

average.Y = total.Y / (\*count);

average.Z = total.Z / (\*count);

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, Count: %d\n", average.X, average.Y, average.Z, (\*count));

}

*Convert2XYZ.c*

#include <stdio.h>

#include <math.h>

#include "RangeSegmentation.h"

/\*

\*\* This routine converts the data in an Odetics range image into 3D

\*\* cartesian coordinate data. The range image is 8-bit, and comes

\*\* already separated from the intensity image.

\*/

void convert2XYZ(unsigned char \*RangeImage, cart\_coord\_t \*Coords, int ROWS, int COLS)

{

int r,c;

double cp[7];

double xangle,yangle,dist;

double ScanDirectionFlag,SlantCorrection;

ScanDirectionFlag = 1;

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 \*/

/\* start with semi-spherical coordinates from laser-range-finder: \*/

/\* (r,c,RangeImage[r\*COLS+c]) \*/

/\* convert those to axis-independant spherical coordinates: \*/

/\* (xangle,yangle,dist) \*/

/\* then convert the spherical coordinates to cartesian: \*/

/\* (P => X[] Y[] Z[]) \*/

for (r=0; r<ROWS; r++)

{

for (c=0; c<COLS; c++)

{

SlantCorrection=cp[3]\*cp[1]\*((double)c-cp[2]);

xangle=cp[0]\*cp[1]\*((double)c-cp[2]);

yangle=(cp[3]\*cp[4]\*(cp[5]-(double)r))+SlantCorrection\*ScanDirectionFlag; /\* + slant correction \*/

dist=(double)RangeImage[r\*COLS+c]+cp[6];

Coords[r\*COLS+c].Z=sqrt((dist\*dist)/(1.0+(tan(xangle)\*tan(xangle))+(tan(yangle)\*tan(yangle))));

Coords[r\*COLS+c].X=tan(xangle)\*Coords[r\*COLS+c].Z;

Coords[r\*COLS+c].Y=tan(yangle)\*Coords[r\*COLS+c].Z;

}

}

return;

}

*WriteColorImage.c*

#include <stdio.h>

#include <math.h>

#include "RangeSegmentation.h"

void WriteColorImage(unsigned char \*grayscale, int ROWS, int COLS)

{

FILE \*fpt;

int r, c;

unsigned char color[3];

fpt = fopen("Colored.ppm", "wb");

fprintf(fpt, "P6 %d %d 255\n", COLS, ROWS);

for (r = 0; r < ROWS; r++)

{

for (c = 0; c < COLS; c++)

{

color[0] = (grayscale[r\*COLS+c] \* 4) % 256;

color[1] = (grayscale[r\*COLS+c] \* 2) % 256;

color[2] = (grayscale[r\*COLS+c] \* 3) % 256;

fwrite(color, 1, 3, fpt);

}

}

fclose(fpt);

return;

}

The WriteColorImage.c was written based on code found at: <http://rosettacode.org/wiki/Bitmap/Write_a_PPM_file#C>