```
//*************************
// Created by Fabrice Mizero
// CS6444 Spring 2016, Final Project.
// Based on this paper: Adaptive Neighborhood Selection for Real-Time Surface Normal
// Estimation from Organized Point Cloud Data Using Integral Images
//
//*************************

#include "CM.h"
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <stdlib.h>
#include <gsl/gsl_math.h>
#include <gsl/gsl_eigen.h>
#define INF 1E20


int main(int argc, char **argv){


    if(argc < 2) die("Usage: ./cov_mat <width><height>[seed][beta][smothing_scale_fa
ctor]");

    const int m = atoi(argv[1]);
    const int n = atoi(argv[2]);
    const int seed = (argc > 3)?atoi(argv[3]):40;
    const int beta = (argc > 4)?atoi(argv[4]):2;
    const int scale_factor = (argc > 5)?atoi(argv[5]):3;

    srand(seed);
    point_rgbd **pc = allocate_pc(m, n);
    initialize_pc(m, n, pc);
    int i,j,k;
    float **intImgs[9];
    //Compute all integral images
    for(k=0; k<9;k++) intImgs[k] = generate_integral_image(pc,m,n,k);

    float **dci_map = allocate_float_array(m,n);
    //Compute the depth change indication map
    point_rgbd p,p1,p2;
    for(i = 0; i < m; i++){
       for(j = 0; j < n; j++){
            dci_map[i][j] = ((i+1>=m)||(j+1>=n)) ? 0 : dc_smoothing_area(pc[i][j],pc
[i+1][j],pc[i][j+1],scale_factor);
       }
     }

    //Compute distance transform map from the depth change indication map
    float **dt_map = allocate_float_array(m,n);
    dt_map = dtBin(dci_map, m, n);

   //Compute per-pixel smothing area/window
    float **smothing_windows = allocate_float_array(m,n);
    smothing_windows = smothing_windows_map(pc, m, n,dt_map, beta);

    for(i = 0; i < m; i++){
       for(j = 0; j < n; j++){
            calc_cov_mat(i,j,intImgs,smothing_windows);
       }
    }

    free(dci_map);
    free(dt_map);
    free(smothing_windows);
    free(pc);

    return 0;
}

void compute_evec(float cov_mat[3][3]){
    gsl_matrix_view m = gsl_matrix_view_array ((double *)cov_mat, 3, 3);
```

```
    gsl_vector *eval = gsl_vector_alloc (3);
    gsl_matrix *evec = gsl_matrix_alloc (3,3);
    gsl_eigen_symmv_workspace * w =  gsl_eigen_symmv_alloc (3);
    gsl_eigen_symmv (&m.matrix, eval, evec, w);
    gsl_eigen_symmv_free (w);
    gsl_eigen_symmv_sort(eval, evec, GSL_EIGEN_SORT_ABS_ASC);
    gsl_vector_view evec_i = gsl_matrix_column (evec, 0);
    gsl_vector_fprintf (stdout, &evec_i.vector, "%g");
    gsl_vector_free (eval);
    gsl_matrix_free (evec);
}

void calc_cov_mat(int m,int n, float ***intImgs, float** smoothing_windows){
    float win[9];
    float cov[3][3];
    int i,j;
    for(i=0; i < 9;i++) win[i] = window_average(intImgs[i],m,n,smoothing_windows[m][
n]);

    float X[3]    = {win[0],win[1],win[2]};
    float Y[3]    = {win[0],win[1],win[2]};
    float A[3][3] = {
        {win[3],win[6],win[7]},
        {win[6],win[4],win[8]},
        {win[7],win[8],win[5]}
    };
    float XY[3][3] = {
        {X[0]*Y[0], X[0]*Y[1], X[0]*Y[2]},
        {X[1]*Y[0], X[1]*Y[1], X[1]*Y[2]},
        {X[2]*Y[0], X[2]*Y[1], X[2]*Y[2]}
    };
    for(i=0; i<3; i++) for(j=0; j<3; j++) cov[i][j] = A[i][j] - XY[i][j];
    compute_evec(cov);
    printf("\n\n");
}

float window_average(float ** intImg,int m,int n,float r){ // S(Io,m,n,r)
    float inv = 1.0 / (4.0 * r * r);
    float region = area(intImg,m+r,n+r) - area(intImg,m-r,n+r) - area(intImg,m+r,n-r
) + area(intImg,m-r,n-r);
    return inv * region;
}

float area(float ** intImg,int x,int y){
    return ((x-1>=0)&&(y-1>=0)) ? intImg[x][y] + intImg[x-1][y-1] - (intImg[x-1][y]
+ intImg[x][y-1]): intImg[x][y];
}

float **smothing_windows_map(point_rgbd *const *pc,int m, int n,float ** dt_map, int
 beta){
    float ** windows = allocate_float_array(m,n);
    int i,j;
    float a = 1.0 / sqrt(2.0);
    double x,y;
    double eps = 0.00001;
    for(i = 0; i < m; i++){
       for(j = 0; j < n; j++){
        x = (double)  dd_smoothing_area(pc[i][j],beta);
        y = (double)  dt_map[i][j] * a;
        windows[i][j] = (gsl_fcmp(x,y,eps)) ? x : y;
       }
    }
    return windows;
}

float *dt1D(float *f, int n) {
  float *d = (float *) malloc(n * sizeof(float));
  int *v = (int *) malloc(n * sizeof(int));
  float *z = (float *) malloc((n+1) * sizeof(float));
  int k = 0;
  v[0] = 0;
  z[0] = -INF;
  z[1] = +INF;
```

```
  int q;
  for (q = 1; q <= n-1; q++) {
    float s  = ((f[q]+(q*q))-(f[v[k]]+ (v[k]*v[k])))/(2*q-2*v[k]);
    while (s <= z[k]) {
      k--;
      s  = ((f[q]+(q*q))-(f[v[k]]+(v[k]*v[k])))/(2*q-2*v[k]);
    }
    k++;
    v[k] = q;
    z[k] = s;
    z[k+1] = +INF;
  }

  k = 0;
  for (q = 0; q <= n-1; q++) {
    while (z[k+1] < q)
      k++;
    d[q] = (q-v[k])*(q-v[k]) + f[v[k]];
  }
  free(v);
  free(z);
  return d;
}

int max(int m, int n){
    return (m > n)?m:n;
}
/* dt of 2d function using squared distance */
void dt2D(float **dci, int m, int n) {
   int x,y;
  float *f = (float *) malloc(max(m,n) * sizeof(float));
  // transform along columns
  for (x = 0; x < m; x++) {
    for (y = 0; y < n; y++) {
      f[y] = dci[y][x];
    }
    float *d = dt1D(f,n);
    for (y = 0; y < n; y++) {
      dci[y][x] = d[y];
    }
    free(d);
  }

  // transform along rows
  for (y = 0; y < n; y++) {
    for (x = 0; x < m; x++) {
      f[x] = dci[y][x];
    }
    float *d = dt1D(f, m);
    for (x = 0; x < m; x++) {
      dci[y][x] = d[x];
    }
    free(d);
  }
  free(f);
}

float **dtBin(float **dci, int m, int n) {
    int x,y;
  float **out = allocate_float_array(m,n);
  for (y = 0; y < n; y++) {
    for (x = 0; x < m; x++) {
      if (dci[y][x] == 1)
    out[y][x] = 0;
      else
    out[y][x] = INF;
    }
  }
  dt2D(out,m,n);
  return out;
}

float depth(point_rgbd point){
```

```
     return (point.x*point.x) + (point.y*point.y) + (point.z*point.z);
}
float dd_smoothing_area(point_rgbd point, float beta){
    float f_dc = alpha * depth(point);
    return  beta * f_dc;
}

float dc_smoothing_area(point_rgbd p, point_rgbd p1, point_rgbd p2,float scale_facto
r){
    float thresh = scale_factor * alpha * depth(p);

    float depth_change_y = depth(p1) - depth(p);
    float depth_change_x = depth(p2) - depth(p);
    return ((depth_change_x>=thresh)||(depth_change_y>=thresh))?1.0:0.0;
}

float **generate_integral_image (point_rgbd *const *pc, int m, int n, int code){
    /*code  feature
     * 0     x
     * 1     y
     * 2     z
     * 3     xx
     * 4     yy
     * 5     zz
     * 6     xy
     * 7     xz
     * 8     yz
    */

    float ** sat = allocate_float_array(m,n);
    float a=0.0, b=0.0, c=0.0, d=0.0;
    int i,j;

    // matrix traversal loop for calculating the SAT
switch(code){
    case 0:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                     // pick up array elements within bounds and picks "zero"
                     // for values outside bounds.
                      a = (i-1>=0)             ? sat[i-1][j]    : 0;
                      b = (j-1>=0)             ? sat[i][j-1]    : 0;
                      c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                      d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].x : 0;
                      sat[i][j] = d + a + b - c;
               }
        }
    break;
    case 1:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                     // pick up array elements within bounds and picks "zero"
                     // for values outside bounds.
                      a = (i-1>=0)             ? sat[i-1][j]    : 0;
                      b = (j-1>=0)             ? sat[i][j-1]    : 0;
                      c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                      d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].y : 0;
                      sat[i][j] = d + a + b - c;
               }
        }
    break;
    case 2:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                     // pick up array elements within bounds and picks "zero"
                     // for values outside bounds.
                      a = (i-1>=0)             ? sat[i-1][j]    : 0;
                      b = (j-1>=0)             ? sat[i][j-1]    : 0;
                      c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                      d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].z : 0;
                      sat[i][j] = d + a + b - c;
               }
        }
```

```
    break;

    case 3:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                      // pick up array elements within bounds and picks "zero"
                      // for values outside bounds.
                       a = (i-1>=0)             ? sat[i-1][j]    : 0;
                       b = (j-1>=0)             ? sat[i][j-1]    : 0;
                       c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                       d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].x * pc[i-1][j-1].x: 0;
                       sat[i][j] = d + a + b - c;
               }
        }
    break;

    case 4:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                      // pick up array elements within bounds and picks "zero"
                      // for values outside bounds.
                       a = (i-1>=0)             ? sat[i-1][j]    : 0;
                       b = (j-1>=0)             ? sat[i][j-1]    : 0;
                       c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                       d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].y * pc[i-1][j-1].y : 0
;
                       sat[i][j] = d + a + b - c;
               }
        }
    break;
    case 5:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                      // pick up array elements within bounds and picks "zero"
                      // for values outside bounds.
                       a = (i-1>=0)             ? sat[i-1][j]    : 0;
                       b = (j-1>=0)             ? sat[i][j-1]    : 0;
                       c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                       d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].z * pc[i-1][j-1].z : 0
;
                       sat[i][j] = d + a + b - c;
               }
        }
    break;
    case 6:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                      // pick up array elements within bounds and picks "zero"
                      // for values outside bounds.
                       a = (i-1>=0)             ? sat[i-1][j]    : 0;
                       b = (j-1>=0)             ? sat[i][j-1]    : 0;
                       c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                       d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].x * pc[i-1][j-1].y : 0
;
                       sat[i][j] = d + a + b - c;
               }
        }
    break;
    case 7:
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                      // pick up array elements within bounds and picks "zero"
                      // for values outside bounds.
                       a = (i-1>=0)             ? sat[i-1][j]    : 0;
                       b = (j-1>=0)             ? sat[i][j-1]    : 0;
                       c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                       d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].x * pc[i-1][j-1].z : 0
;
                       sat[i][j] = d + a + b - c;
               }
        }
    break;
    case 8:
```

```
        for(i = 0; i < n; i++){
               for(j = 0; j < m; j++){
                     // pick up array elements within bounds and picks "zero"
                     // for values outside bounds.
                      a = (i-1>=0)             ? sat[i-1][j]    : 0;
                      b = (j-1>=0)             ? sat[i][j-1]    : 0;
                      c = ((i-1>=0)&&(j-1>=0)) ? sat[i-1][j-1]  : 0;
                      d = ((i-1>=0)&&(j-1>=0)) ? pc[i-1][j-1].y * pc[i-1][j-1].z : 0
;
                      sat[i][j] = d + a + b - c;
               }
        }
    break;
    default:
        printf("Integral image gen error: Invalid code\n");
    }//end switch
    return sat;
}

point_rgbd **allocate_pc(const int n, const int m) {
  int i;
  point_rgbd ** pc = (point_rgbd **)malloc((unsigned) n * sizeof(point_rgbd *));
  for (i=0; i<n; i++)  pc[i] = (point_rgbd *)malloc((unsigned) m * sizeof(point_rgbd
));
  return pc;
}

float **allocate_float_array(const int n, const int m) {
    float *data = (float *)malloc((unsigned long) n*m*sizeof(float));
    float **array= (float **)malloc((unsigned long) n*sizeof(float*));
    int i;
    for (i=0; i<n; i++)
        array[i] = &(data[m*i]);
    return array;
}

void initialize_pc(const int n, const int m, point_rgbd *const *pc) {
    int i, j;
    for(i = 0; i < n; i++) {
        for( j = 0; j < m; j ++) {
            pc[i][j].x =((float)rand()/(float)(RAND_MAX));
            pc[i][j].y =((float)rand()/(float)(RAND_MAX));
            pc[i][j].z =((float)rand()/(float)(RAND_MAX));

        }
    }
}

void die(const char *error) {
    printf("%s", error);
    exit(1);
}
```