Lista de Exercícios 3

Vinicius Graciano Santos
vgs@dcc.ufmg.br

6 de junho de 2010

1 Exercícios Teóricos

1. O espaço de parâmetros (m, n) pode assumir valores entre $[-\infty, \infty]$, o que implica que não podemos amostrá-lo por completo. Além disso, não é possível representar retas verticas, $x = k$. Dessa maneira, uma forma mais interessante de representação é o espaço de parâmetros (ρ, θ), onde uma reta é representada pela sua distância ρ e o ângulo θ de sua normal, ambas em relação à origem. Assim, um ponto (x, y) na imagem é representado no novo espaço de parâmetros pela seguinte equação:

$$\rho = x \cos\theta + y \sin\theta$$

   Além de podermos representar as retas verticais, ambos ρ e θ são limitados, com $\rho \in [0, \sqrt{N^2 + M^2}]$ e $\theta \in [0, \pi]$, sendo N e M as dimensões da imagem. Note também que nesse modelo os pontos da imagem serão representados por senóides no espaço de parâmetros.

2. Podemos utilizar a técnica de detecção de retas baseada na transformada de Hough para esse propósito. O algoritmo consiste em incrementar os contadores das retas no espaço de parâmetros representadas por cada ponto do conjunto de dados. Por exemplo, dado um ponto (x, y) desse conjunto, considerando-se o espaço (n, m), devemos incrementar todos os respectivos contadores da reta $n = x(-m) + y$. Assumindo que a quantidade de *outliers* não seja significativa, é possível encontrar uma aproximação do modelo linear que representa esses dados a partir da localização do máximo global encontrado nos contadores do espaço de parâmetros.

3. Um alvo de calibração deve possuir um formato e padrões simples para facilitar a sua medição e o processo de correspondência dos pontos no plano da imagem, fornecendo também uma quantidade aceitável de pontos para a solução dos sistemas lineares que buscam determinar os parâmetros intrínsecos e extrínsecos. Os padrões normalmente utilizados são sequências bem definidas de vários quadriláteros em preto e branco, onde é possível utilizar as suas quinas como o conjunto de pontos necessários para a calibração, podendo ser detectadas através da interseção das retas encontradas através do método da transformada de Hough.

   Também é desejável que o próprio formato do objeto defina a base de referência do mundo, por motivos de simplicidade. Assim não é necessário nenhuma transformação adicional das medidas reais dos padrões para um outro referencial. Se o padrão do alvo de calibração estiver todo contido em um único plano, será necessário mais de uma imagem do objeto em diferentes ângulos para que o sistema possua uma solução não-trivial.

4. Sim, é possível. O problema de encontrar a profundidade da imagem é conhecido como *Shape-from-X*, onde X pode ser substituído por um dos vários métodos existentes. Além da visão estéreo, existem métodos baseados em texturas, *shading*, movimento, *focus/defocus*, entre outros. Os dois primeiros métodos necessitam de apenas uma imagem e são descritos abaixo:

   - Shape-from-shading: recupera a superfície de acordo com a variação gradual da iluminação em um objeto. O sistema precisa conhecer a iluminação da cena, como a direção da luz e o modelo de reflexão, normalmente assumido como sendo Lambertiano; o que pode acarretar em erros em imagens reais, que contém objetos especulares.

- Shape-from-texture: O método se baseia na distorção de *texels* individuais. A sua variação pela imagem nos permite estimar a forma da superfície observada. A reconstrução leva em conta as distorções geradas pela projeção perspectiva, que projeta objetos distantes com tamanhos menores dos que os mais próximos e diminui o tamanho de objetos não paralelos ao plano de projeção (*foreshortening*). A superfície é reconstruída a partir do cálculo das suas normais, gerando um campo de vetores. Considerando-se que o mapa seja denso e que as superfícies tenham variações suaves, é possível aproximar a forma dos objetos.

2 Exercícios Práticos

1. O sistema estéreo utilizado foi composto de duas câmeras disponíveis no *Verlab*. O alvo de calibração utilizado foi um padrão de tabuleiro simples contido em um plano. Abaixo segue uma foto do alvo e a visualização dos parâmetros extrínsecos.

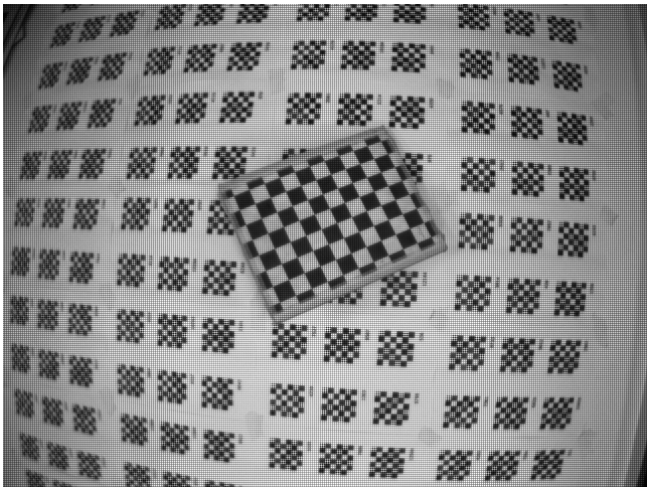


Figura 1: Alvo de Calibração

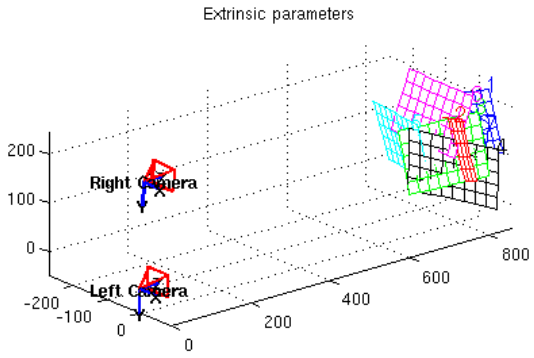


Figura 2: Visualização dos Parâmetros Extrínsecos

2. O algoritmo **CORR_MATCHING** foi implementado sem nenhuma melhoria de eficiência. Os resultados mostraram que o algoritmo é lento e portanto o uso da geometria epipolar para diminuir a sua complexidade temporal é fundamental para aplicações que necessitam de respostas mais rápidas.

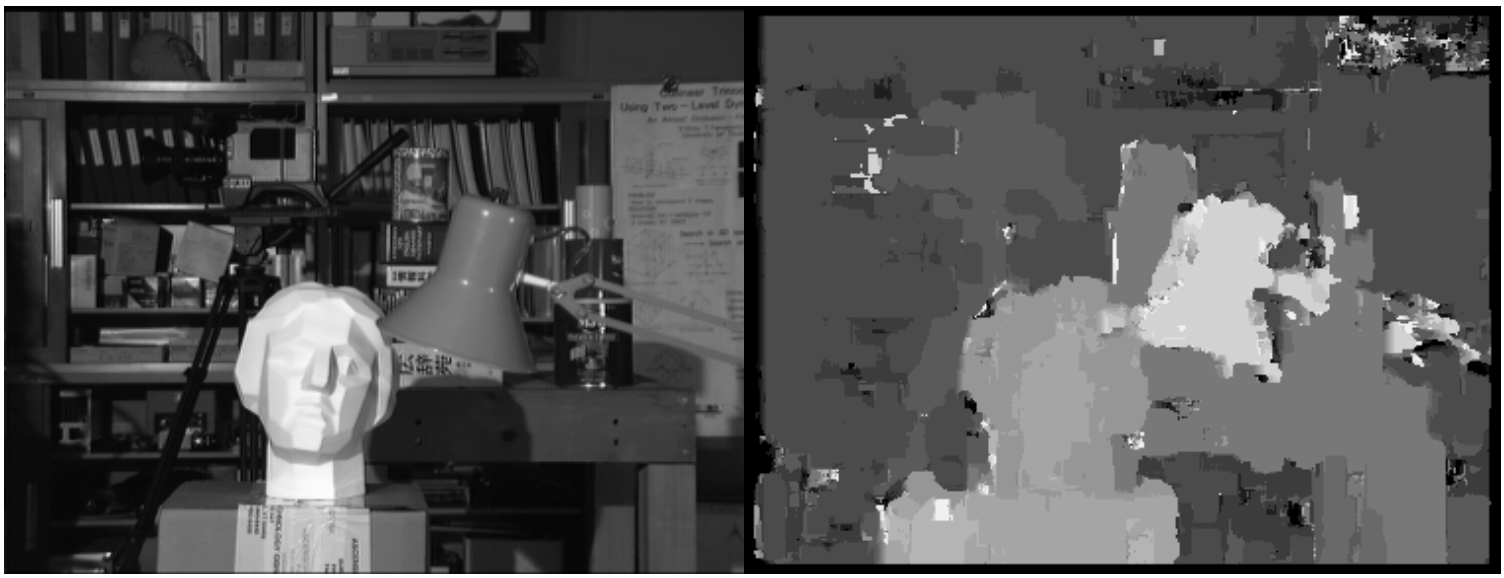


Figura 3: Uma das imagens originais e o mapa de disparidade

```
1 #include <limits>
2 #include <cstdlib>
3 #include <iostream>
```

```
4
5  #include "cv.h"
6  #include "highgui.h"
7
8  using namespace cv;
9  using namespace std;
10
11 double correlate(Mat &left, Mat &right, Point center_l, Point center_r, int W){
12         double sum = 0;
13         for (int i = -W; i <= W; i++)
14                 for (int j = -W; j <= W; j++)
15                         sum += pow(left.at<uchar>(center_l.x + i, center_l.y + j) - right.at<uchar>(center_r.x + i,
                                center_r.y + j),2);
16         return sum;
17 }
18
19 void corr_matching(Mat &left, Mat &right, int W = 5, int maxDisp = 16)
20 {
21         double min, disprow, dispcol, aux;
22         Mat disparityX(left.rows, left.cols, CV_8U);
23         Mat disparityY(left.rows, left.cols, CV_8U);
24
25         assert(W % 2 != 0);
26
27         //Calcula a disparidade horizontal e vertical.
28         for (int i = W; i + W < left.rows; i++){
29                 for (int j = W; j + W < left.cols; j++){
30                         disprow = 0;
31                         dispcol = 0;
32                         min = numeric_limits<double>::max();
33                         for (int k = i - maxDisp; k <= i + maxDisp; k++){
34                                 for (int l = j - maxDisp; l <= j + maxDisp; l++){
35                                         if ((k - W >= 0 && k + W < left.rows) && (l - W >= 0 && l + W < left.cols)){
36                                                 aux = correlate(left, right, Point(i, j), Point(k, l), W);
37
38                                                 if (aux < min){
39                                                         min = aux;
40                                                         disprow = k - i;
41                                                         dispcol = l - j;
42                                                 }
43                                         }
44                                 }
45                         }
46                         disparityX.at<uchar>(i, j) = (uchar) fabs(dispcol);
47                         disparityY.at<uchar>(i, j) = (uchar) fabs(disprow);
48                 }
49         }
50
51         //Mapeia a disparidade no intervalo [0, 255] para melhor visualizacao humana. :-)
52         disparityX.convertTo(disparityX, CV_8U, 255/maxDisp);
53         disparityY.convertTo(disparityY, CV_8U, 255/maxDisp);
54         imwrite("dispX.bmp", disparityX);
55         imwrite("dispY.bmp", disparityY);
56 }
57
58 int main(int argc, char **argv)
59 {
60         Mat left, right;
61
62         //Unix args.
63         if (argc < 3){
64                 cout << "Usage: corr_match <left_image> <right image>" << endl;
65                 return EXIT_FAILURE;
66         }
67
68         left  = imread(argv[1], 0);
69         right = imread(argv[2], 0);
70
71         if (left.data == NULL || right.data == NULL || left.size() != right.size()){
72                 cout << "Could not read one of the images or they don't have the same size!" << endl;
73                 return EXIT_FAILURE;
74         }
75
76         corr_matching(left, right);
77         cout << "Done!" << endl;
78         return EXIT_SUCCESS;
79 }
```

3. O algoritmo desenvolvido recebe como entrada o mapa de disparidade codificado em uma imagem de intensidade e a disparidade máxima utilizada em *CORR_MATCH* para converter os valores de [0,255] para [0, max_disp]. Infelizmente, nenhum algoritmo para formar as faces a partir da nuvem de pontos resultante foi implementado, logo o arquivo de saída apenas contém informações sobre os vértices da triangulação.

```
1  #include <fstream>
2  #include <iostream>
3  #include <vector>
4  #include "cv.h"
5  #include "highgui.h"
6
7  void triang(const char *filename, Mat &dispX, Mat &dispY)
8  {
9          ofstream file;
10         vector <Mat_ <double> > points;
11
12         Mat_ <double> A(3,3);
13         Mat_ <double> R(3,3), T(3,1), left_int(3,3), right_int(3,3);
14
15         //Parametros intrinsecos e extrinsecos (Definidos manualmente para nao ser necessario imprimir uma rotina de
               leitura de arquivos).
16         T << 3.00203, 58.80952, 2.95714;
17         R << 0.9999, -0.0019, -0.0152, 0.003, 0.9975, 0.0702, 0.0150, -0.0703, 0.9974;
18         left_int << 527.84097, 0, 318.55640, 0, 527.52188, 272.60326, 0, 0, 1;
19         right_int << 526.31113, 0, 360.37305, 0, 526.01903, 273.01179, 0, 0, 1;
20
21         left_int = left_int.inv();
22         right_int = right_int.inv();
23
24         for (int i = 0; i < dispX.rows; i++){
```

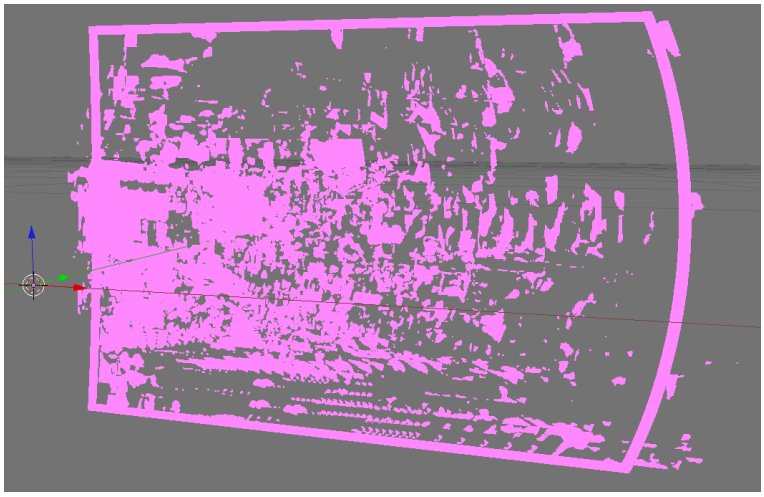
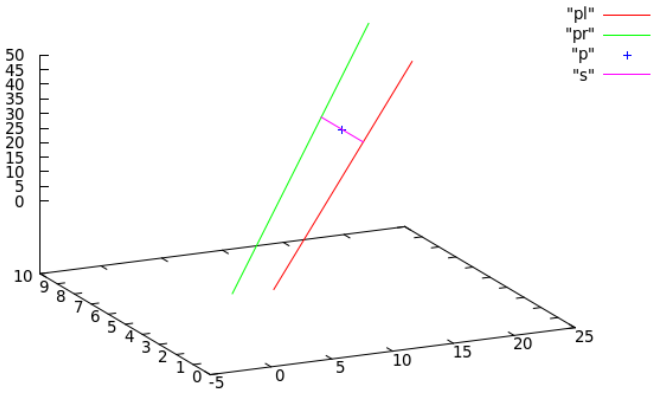


Figura 4: Imagem original da câmera esquerda e a nuvem de pontos resultante da triangulação



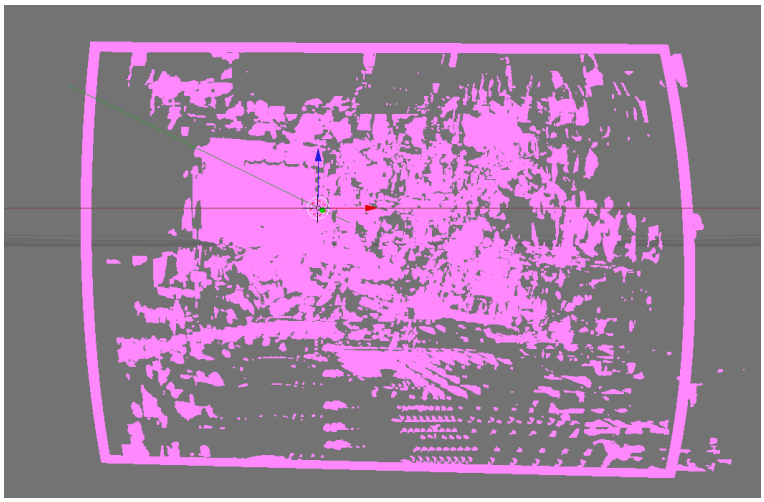


Figura 5: Verificação gráfica do algoritmo de triangulação e o resultado sobre outra perspectiva

```
25                      for (int j = 0; j < dispX.cols; j++){
26                              Mat_ <double> Pl(3,1), Pr(3,1), P(3,1);
27
28                              //Calcula Pr e Pl nos seus respectivos frames de referencia.
29                              Pl << j,i,1;
30                              Pr << j - dispX.at<double>(i,j), i - dispY.at<double>(i,j), 1;
31                              Pl = left_int*Pl;
32                              Pr = right_int*Pr;
33
34                              //Monta a matriz A do sistema linear a*Pl - b*R'*Pr + c*(Pl x R'*Pr) = T
35                              A.col(0) = Pl + 0; //+0 eh necessario para que a OpenCV copie o novo conteudo.
36                              A.col(1) = -(R.t()*Pr);
37                              A.col(2) = Pl.cross(R.t()*Pr);
38
39                              //Resolve o sistema (sem verificar viabilidade) e calcula o ponto medio do segmento perpendicular
                                    aos raios a*Pl e b*Pr.
40                              P = A.inv()*T;
41                              P = P(0,0)*Pl + 0.5*((T + P(1,0)*R.t()*Pr) - P(0,0)*Pl);
42
43                              //Insere o ponto em um vector para pos-processamento.
44                              points.push_back(P);
45                      }
46              }
47
48              //Seria possivel tratar os pontos resultantes ou definir faces para o modelo geometrico nesta parte,
49              mas apenas os vertices foram considerados.
50
51              file.open(filename);
52              for (int i = 0; i < points.size(); i++)
53                      file << "v " << points[i](0, 0) << " " << points[i](1, 0) << " " << points[i](2, 0) << endl;
54              file.close();
55              points.clear();
56      }
57
58      int main(int argc, char **argv)
59      {
60              unsigned int max;
61              Mat dispX, dispY;
62
63              if (argc < 4){
64                      cout << "Usage: " << argv[0] << "<disparityX> <disparityY> <max_disparity> <output>" << endl;
65                      cout << "       " << argv[0] << "dispX.bmp dispY.bmp 128 output.obj" << endl;
66                      return -1;
67              }
68
69              dispX = imread(argv[1], 0);
70              dispY = imread(argv[2], 0);
71
72              dispX.convertTo(dispX, CV_64FC1, atoi(argv[3])/255.);
73              dispY.convertTo(dispY, CV_64FC1, atoi(argv[3])/255.);
```

```
74          triang(argv[4], dispX, dispY);
75  }
```

3 Exercícios de Pesquisa

O benefício de se utilizar mais de duas câmeras para fazer a reconstrução geométrica de uma cena é a maior quantidade de dados iniciais, que permitem inferir a profundidade da cena com maior precisão. Além disso, é possível reconstruir totalmente um objeto dada uma quantidade aceitável de fotos mostrando os seus detalhes no espaço. Os problemas dessa abordagem são novamente a correspondência de pontos, que agora devem ser casados em múltiplas imagens, a relação espacial entre eles, é necessário encontrar um consenso sobre a profundidade real do ponto observado estimado a partir das várias câmeras, além da oclusão.

Uma das abordagens mais simples ao problema considera o ocupamento de *voxels* (volume elements) , onde a cena é represetada por um conjunto desses elementos e a tarefa a ser computada é a marcação desses *voxels* como cheios ou vazios. Normalmente um método de interseção de silhuetas a partir das várias câmeras é empregado para definir essa ocupação. Também podemos utilizar uma única câmera e posicionar o objeto em uma mesa giratória, registrando várias imagens do objeto em diferentes ângulos. É sabido que essa interseção, mesmo na ausência de ruído, não gera a verdadeira forma do objeto, mas sim uma aproximação denominada *visual hull*.

Uma outra abordagem é considerar o problema em termos de minimização de energia. Basicamente, uma função potencial é definida para representar o problema e a solução é encontrada em seu ponto mínimo. Vários algoritmos se basearam nesse conceito e foram implementados de forma eficiente utilizando-se cortes em grafos.

Referências

[1] E. Trucco, A. Verri, "Introductory Techniques for 3-D Computer Vision", Prentice Hall, 1998.

[2] R. Zhang, P.S. Tsai, J. E. Cryer, M. Shah, "Shape from Shading: A Survey". IEEE Transactions on Pattern Analysis and Machine Intelligence. pp. 690-706, v. 21. 1999.

[3] M. Langer. "Lecture 3.3: Shape from X". Disponível em: <www.cim.mcgill.ca/˜langer/646/2008/shape-from-X.pdf>. Acessado em: 24/05/2010.

[4] V. Kolmogorov, R. Zabih, S. Gortler. "Generalized Multi-camera Scene Reconstruction Using Graph Cuts". Proceedings of the International Workshop on Energy Minimization Methods in Computer Vision and Pattern Recognition, pp. 501-516. 2003.