

# Stereo Camera 图像处理与机器视觉大作业

## Stereo Camera 图像处理与机器视觉大作业

### 0. 成员分工

#### 1. 问题描述

#### 2. 实现细节

##### 2.1. 相机标定(Stereo Calibration)

###### 2.1.1. 绘制标定板

###### 2.1.2. 单目像机标定

###### 2.1.3. 单目像机标定实现

###### 2.1.4. 双目相机标定

###### 2.1.4.1. 双目标定原理

###### 2.1.4.2. 双目标定代码所用函数说明

##### 2.2. 双目像机图像矫正(Stereo Processing)

###### 2.2.1. 双目相机图片矫正原理

###### 2.2.2. 双目矫正代码所用函数说明

##### 2.3. 稀疏匹配(Sparse Stereomatching)

###### 2.3.1. 角点检测(Detector)

###### 2.3.1.1. Harris角点检测

###### 2.3.1.2. FAST角点检测

###### 2.3.2. 描述子运算(Descriptor)

###### 2.3.3. 特征点匹配(Matching)

###### 2.3.4. 极几何修正(Utilizing Epipolar Geometry)

##### 2.4. 稠密匹配(SGBM)

###### 2.4.1 预处理

###### 2.4.2 代价计算

###### 2.4.3 动态规划

###### 2.4.4 后处理

###### 2.4.5 SGBM算法的参数含义及数值选取

##### 2.5. 视差及点云深度计算

##### 2.6. 3D点云可视化

###### 2.6.1. Pangolin库介绍

###### 2.6.2. 代码细节展示

### 3. 实验过程及结果

#### 3.1. 相机标定及图像校正实验

##### 3.1.1. 绘制标定板

##### 3.1.2. 单目像机标定

##### 3.1.3. 双目像机标定

##### 3.1.4. 双目像机图像矫正

#### 3.2. 角点检测实验

#### 3.3. 稀疏点云建模实验

#### 3.4. 稠密点云建模实验

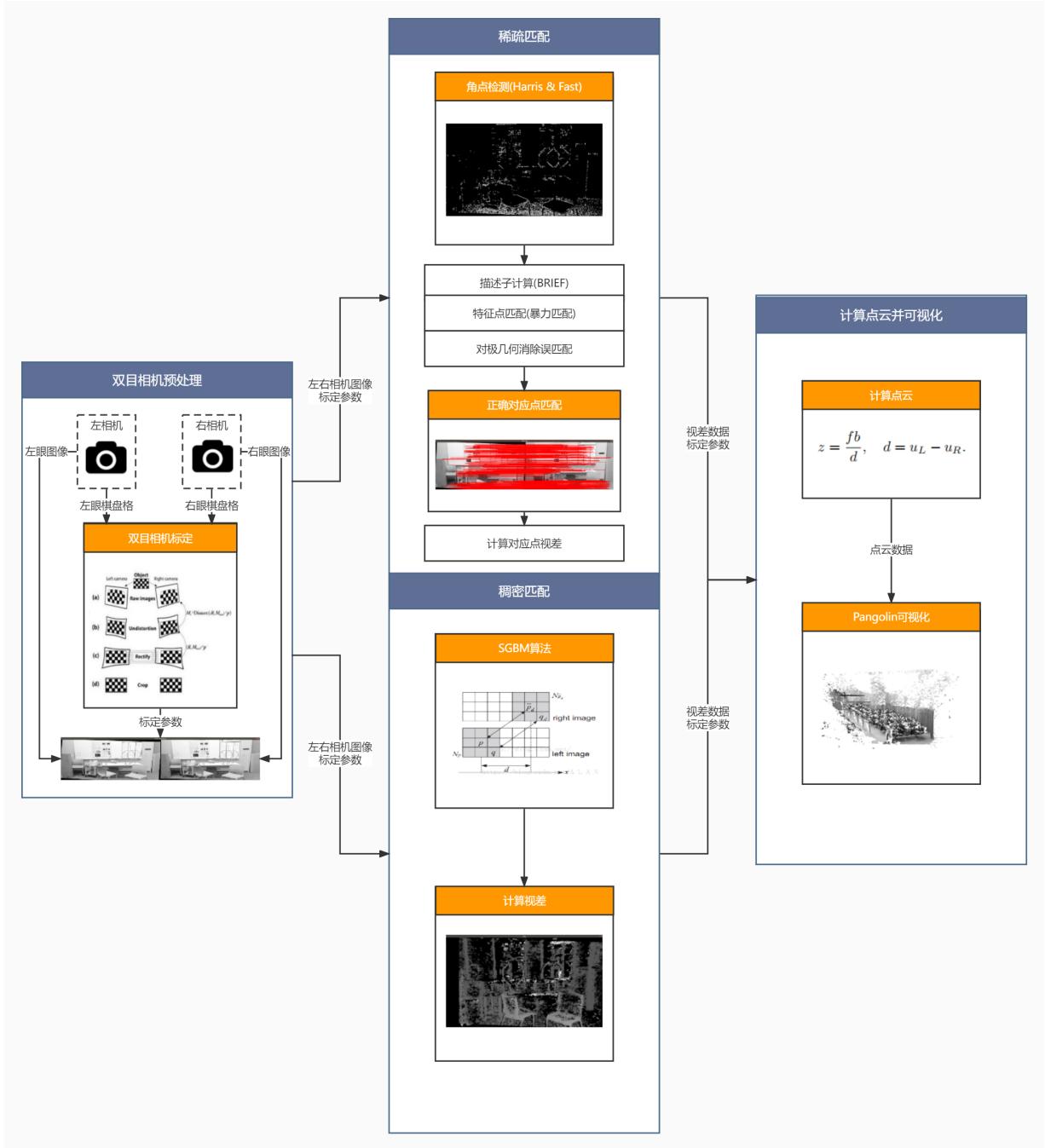
#### 4. 应用背景分析或展望

## 0. 成员分工

学号	姓名	工作内容	占比
1950083	刘智宇	Harris角点检测及FAST角点检测算法实现，报告撰写，PPT制作	33.33%
1950089	季马泽宇	特征点描述及匹配，视差深度计算，点云可视化，报告撰写，PPT制作	33.33%
1950079	陈昊鹏	相机内参标定及去畸变，双目相机标定，报告撰写，PPT制作	33.33%

## 1. 问题描述

本项目完成的是基于双目立体视觉的三维重构任务。实现了从相机标定、图像匹配到点云可视化的全部过程。整体流程如以下流程图所示。



首先是相机标定，分为单目相机标定、去畸变和双目相机对齐两个部分。首先使用张氏棋盘格标定法计算出单目相机的内参和畸变系数。再使用双目相机标定对齐左右眼图像。利用标定参数对左右相机图像进行预处理，并且切除多余黑边，得到对齐的左右眼图像。

图像匹配使用了两种不同的匹配方式。一是基于特征点的稀疏匹配：首先使用Harris或者Orb角点检测算法找出特征点，接着使用Brief计算特征描述符，利用暴力匹配的方式计算出特征点间的对应关系，最后使用对极几何剔除误匹配。最终，我们得到图像中对应的特征点。二是使用了基于SGBM算法的稠密图匹配算法，左右两幅图所有点的匹配关系。

得到匹配点关系后，结合相机内参以及对应点关系，可以计算出图像的视差，最终计算出每个点的深度。我们记录下了每个点的三维坐标信息，并使用pangolin库进行可视化，最终得到了图像的三维重建结果。

## 2. 实现细节

### 2.1. 相机标定(Stereo Calibration)

## 2.1.1. 绘制标定板

本实验对摄像机的标定采用的是张正友博士的张氏标定法，棋盘是一块由黑白方块间隔组成的标定板，我们用它来作为相机标定的标定物。

```
//函数声明，默认每行11个block，每列8个block，block大小为75个像素。也就是10*7个内点
void drawChessBoard(int blocks_per_row, int blocks_per_col, int block_size)
{
    Size board_size = Size(block_size * blocks_per_row, block_size * blocks_per_col);
    Mat chessboard = Mat(board_size, CV_8UC1);
    unsigned char color = 0;
    for (int i = 0; i < blocks_per_row; i++)
    {
        color = ~color;
        for (int j = 0; j < blocks_per_col; j++)
        {
            chessboard(Rect(i * block_size, j * block_size, block_size,
block_size)).setTo(color);
            color = ~color;
        }
    }
    Mat chess_board = Mat(board_size.height + 100, board_size.width + 100, CV_8UC1,
Scalar::all(256)); //上下左右留出50个像素空白
    chessboard.copyTo(chess_board.rowRange(50, 50 + board_size.height).colRange(50, 50 +
board_size.width));
    imshow("chess_board", chess_board);
    imwrite("chess_board.png", chess_board);
    waitKey(-1);
    destroyAllWindows();
}
```

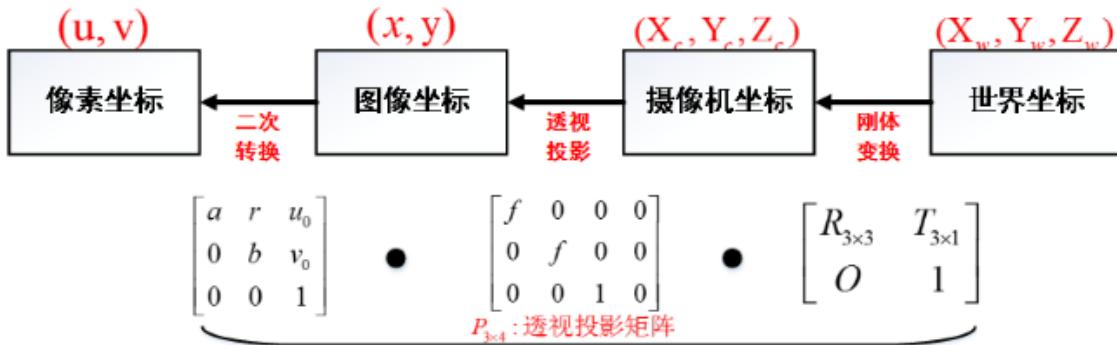
## 2.1.2. 单目像机标定

在进行相机标定之前首先要将摄像机拍摄的图片从世界坐标转换成摄像机坐标，再转换成图像坐标，最后转换成像素坐标。

这四个坐标系的定义如下：

1. 世界坐标系 (world coordinate)，以其为基准可以描述相机和待测物体的空间位置。
2. 相机坐标系 (camera coordinate)，原点位于镜头光心处，x、y轴分别与相面的两边平行，z轴为镜头光轴，与像平面垂直。
3. 图像坐标系 (photo coordinate)，是坐标原点在CCD图像平面的中心x,y轴分别平行于图像像素坐标系的(u,v)轴，坐标用(x,y)表示。
4. 像素坐标系 (pixel coordinate)，是一个二维直角坐标系，反映了相机CCD/CMOS芯片中像素的排列情况。原点位于图像的左上角，轴、轴分别于像面的两边平行。像素坐标系中坐标轴的单位是像素（整数）。

各个坐标系转换方法如下图所示：



$$Z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{1}{dx} & 0 & u_0 \\ 0 & \frac{1}{dy} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 & 0 \\ 0 & f_y & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & T \\ \vec{0} & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

内参矩阵
外参矩阵

张正友标定法思想：

1. 通过每张照片上的点的对应关系求出转换矩阵H (其中用到了最小二乘法思想减小误差)
2. 通过多张图片求出的多个H, 来进一步求内参矩阵A, 然后就进一步可求出外参矩阵, 最后再通过最小二乘法思想对畸变进行评估, 进而求出更加精确的值。
3. 相机的畸变和内参是相机本身的固有特性, 标定一次即可一直使用。但由于相机本身并非理想的小孔成像模型以及计算误差, 采用不同的图片进行标定时得到的结果都有差异。一般重投影误差很小的话, 标定结果均可用。

### 2.1.3. 单目像机标定实现

使用到的函数说明：

1. `findChessboardCorners()`: 找到标定板内角点位置, 确定输入图像是否是棋盘模式, 并确定角点的位置
2. `find4QuadCornerSubpix()`: 对提取到的角点进一步提取亚像素信息, 用于提高标定精度, 获取内角点的精确位置。
3. `drawChessboardCorners()`: 在棋盘标定图上绘制找到的内角点 (仅用于显示)
4. `calibrateCamera()`: 通过多个视角的2D/3D对应, 求解出该相机的内参数和每一个视角的外参数。
5. `projectPoints()`: 通过得到的摄像机内外参数, 对空间的三维点进行重新投影计算, 得到空间三维点在图像上新的投影点的坐标。

```

while (getline(fin, image_file_name))
{
    image_nums++;
    Mat image_raw = imread(image_file_name); //读入图片
    if (image_nums == 1) //确认下标定图片的尺寸
    {
        image_size.width = image_raw.cols; //图像的宽对应着列数
        image_size.height = image_raw.rows; //图像的高对应着行数
        cout << "image_size.width = " << image_size.width << endl;
        cout << "image_size.height = " << image_size.height << endl;
    }
    //角点检测部分
    Mat image_gray; //存储灰度图的矩阵
    cvtColor(image_raw, image_gray, CV_RGB2GRAY); //将RGB图转化为灰度图
    // 提取角点
    bool success = findChessboardCorners(image_gray, corner_size, points_per_image);
    if (!success)
    {
        cout << "can not find the corners " << endl;
        exit(1);
    }
    else
    {
        //亚像素精确化 (两种方法)
        find4QuadCornerSubpix(image_gray, points_per_image, Size(5, 5)); //亚像素角点
        points_all_images.push_back(points_per_image); //保存亚像素角点
        //在图中画出角点位置
        //角点可视化
        drawChessboardCorners(image_raw, corner_size, points_per_image, success); //将角点连线
        //调试用查看角点连线效果
        //imshow("Camera calibration", image_raw);
        //waitKey(0); //等待按键输入
    }
}

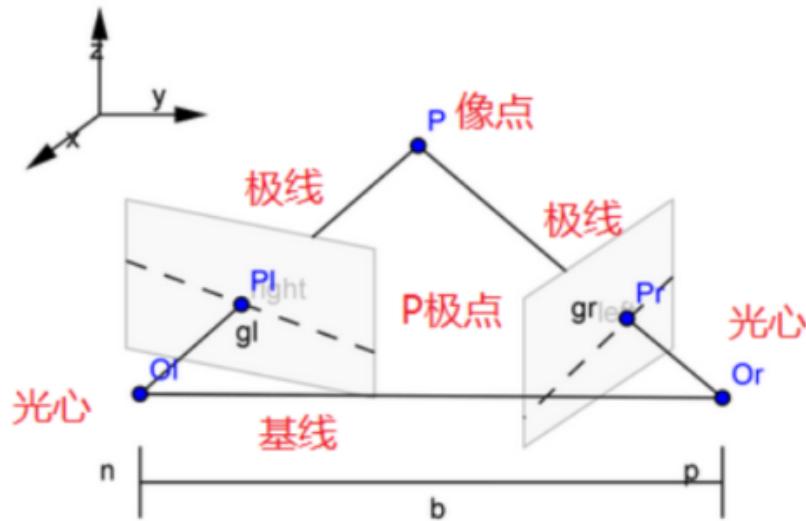
```

## 2.1.4. 双目相机标定

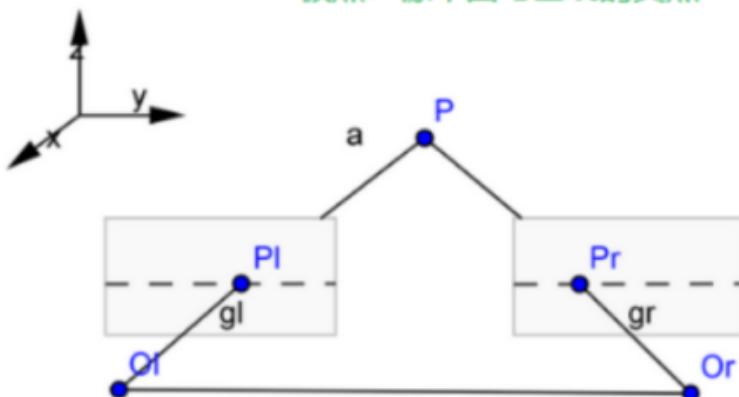
### 2.1.4.1. 双目标定原理

双目摄像头标定不仅要得出每个摄像头的内部参数，还需要通过标定来测量两个摄像头之间的相对位置（即右摄像头相对于左摄像头的三维平移  $t$  和旋转  $R$  参数），才能进行距离计算。然而，在二维空间上匹配对应点是非常耗时的，为了减少匹配搜索范围，我们可以利用极线约束使得对应点的匹配由二维搜索降为一维搜索。

双目标定前后，双目模型对比如下图所示：校正前，相机的光心不是相互平行的；校正后，极点在无穷远处，两个相机的光轴平行，像点在左右图像上的高度一致。



极点：像平面与基线的交点



### 2.1.4.2. 双目标定代码所用函数说明

RealPoint() : 计算标定板上模块的实际物理坐标

```
void RealPoint(vector<vector<Point3f>>& obj, int points_per_row, int points_per_col, int imgNumber, int squareSize)
{
    vector<Point3f> imgpoint;
    for (int rowIndex = 0; rowIndex < points_per_col; rowIndex++)
    {
        for (int colIndex = 0; colIndex < points_per_row; colIndex++)
        {
            imgpoint.push_back(Point3f(rowIndex * squareSize, colIndex * squareSize, 0));
        }
    }
    for (int imgIndex = 0; imgIndex < imgNumber; imgIndex++)
    {
        obj.push_back(imgpoint);
    }
}
```

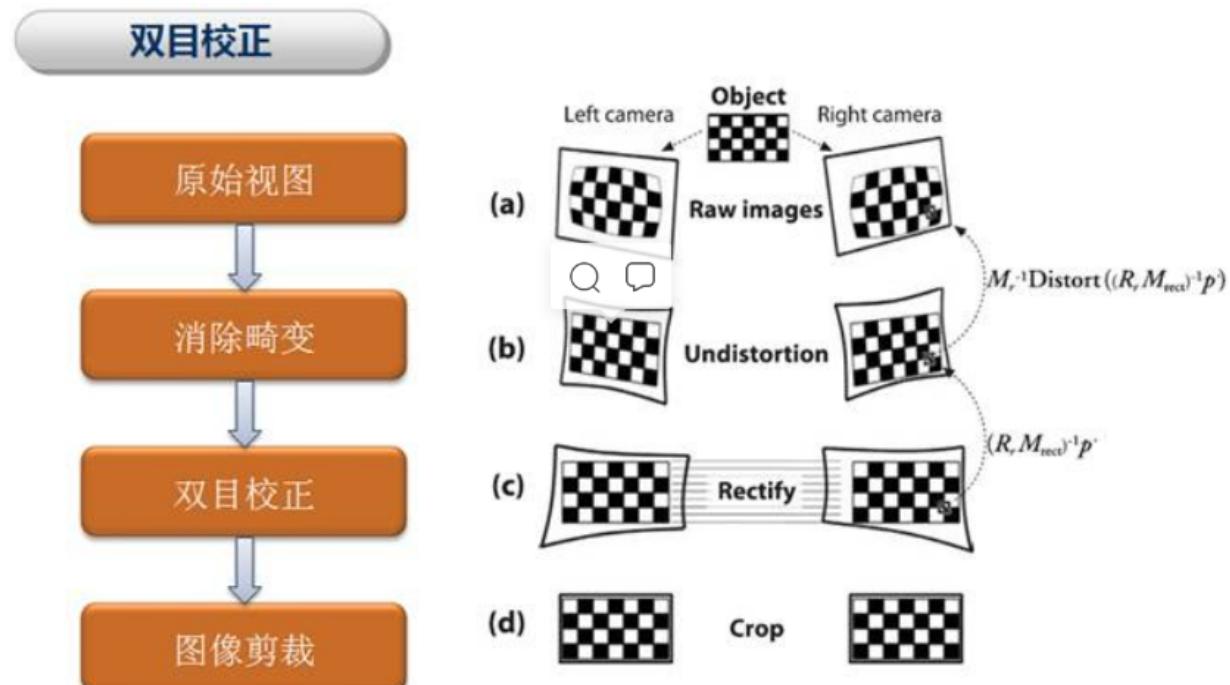
stereoCalibrate():计算了两个摄像头进行立体像对之间的转换关系

```
double rms = stereoCalibrate(objRealPoint, imagePointL, imagePointR,
    cameraMatrixL, distCoeffL,
    cameraMatrixR, distCoeffR,
    image_size, R, T, E, F, CALIB_USE_INTRINSIC_GUESS,
    TermCriteria(TermCriteria::COUNT + TermCriteria::EPS, 100, 1e-5));
```

## 2.2. 双目像机图像矫正(Stereo Processing)

### 2.2.1. 双目相机图片矫正原理

双目校正的作用就是要把消除畸变后的两幅图像严格地行对应，使得两幅图像的对极线恰好在同一水平线上（如下图所示），这样一幅图像上任意一点与其在另一幅图像上的对应点就必然具有相同的行号，只需在该行进行一维搜索即可匹配到对应点。



### 2.2.2. 双目矫正代码所用函数说明

用到的函数如下：

1. stereoRectify(): 为每个摄像头计算立体校正的映射矩阵。
2. initUndistortRectifyMap(): 计算畸变矫正和立体校正的映射变换
3. remap(): 用于重映射，即将一幅图像中某位置的像素放置到另一个图片指定位置

核心代码如下：

```
stereoRectify(cameraMatrixL, distCoeffL, cameraMatrixR, distCoeffR, image_size, R, T, Rl, Rr, P1,
Pr, Q, CALIB_ZERO_DISPARITY, -1, image_size, &validROI_L, &validROI_R);

initUndistortRectifyMap(cameraMatrixL, distCoeffL, Rl, P1, image_size, CV_32FC1, mapLx, mapLy);

initUndistortRectifyMap(cameraMatrixR, distCoeffR, Rr, Pr, image_size, CV_32FC1, mapRx, mapRy);

remap(photoleft, rectifyImageL, mapLx, mapLy, INTER_LINEAR); //重映射

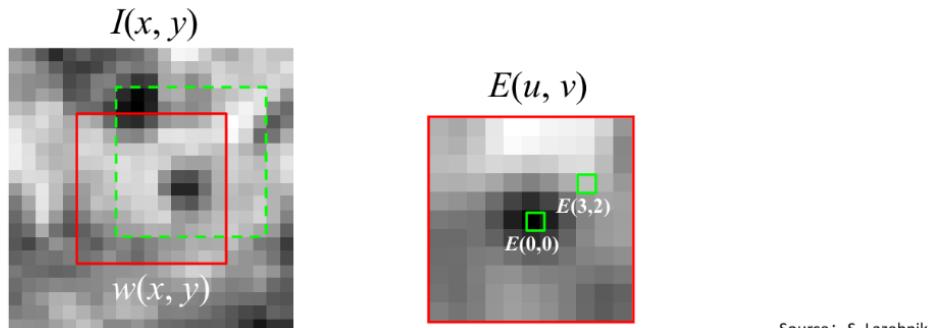
remap(photoright, rectifyImageR, mapRx, mapRy, INTER_LINEAR);
```

## 2.3. 稀疏匹配(Sparse Stereomatching)

### 2.3.1. 角点检测(Detector)

本次大作业主要手写了两个角点检测算法，Harris角点检测以及FAST角点检测。

#### 2.3.1.1. Harris角点检测



$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$$

对上式进行二阶泰勒展开，并写为矩阵形式得到下式。

$$E(u, v) \approx E(0, 0) + [u \quad v] \begin{bmatrix} E_u(0, 0) \\ E_v(0, 0) \end{bmatrix} + \frac{1}{2}[u \quad v] \begin{bmatrix} E_{uu}(0, 0) & E_{uv}(0, 0) \\ E_{uv}(0, 0) & E_{vv}(0, 0) \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

上式可化简为如下形式：

$$E(u, v) \approx [u \quad v] M \begin{bmatrix} u \\ v \end{bmatrix}$$

其中，M矩阵计算公式如下：

$$M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

建立Sobel算子，用于求取图像在x、y方向上的导数。

```
Mat SobeloperatorX = (Mat<float>(3,3)<<-1, 0, 1,
                        -2, 0, 2,
                        -1, 0, 1);
Mat SobeloperatorY = (Mat<float>(3,3)<<-1,-2,-1,
                        0, 0, 0,
                        1, 2, 1);
```

使用Sobel算子卷积得到 $I_x$ 、 $I_y$ 、 $I_x^2$ 、 $I_x I_y$ 、 $I_y^2$ 。在进行计算时为了高效性，提前用窗函数与 $I_x^2$ 、 $I_x I_y$ 、 $I_y^2$ 进行卷积。

```
Mat Ix,Iy,IxIx,IyIy,IxIy;
filter2D(srcImg,Ix,CV_32F,SobeloperatorX);
filter2D(srcImg,Iy,CV_32F,SobeloperatorY);
multiply(Ix,Ix,IxIx);
multiply(Ix,Iy,IxIy);
multiply(Iy,Iy,IyIy);
Mat weightIxIx,weightIxIy,weightIyIy;
filter2D(IxIx,weightIxIx,CV_32F,windowFunctionKernel);
filter2D(IxIy,weightIxIy,CV_32F,windowFunctionKernel);
filter2D(IyIy,weightIyIy,CV_32F,windowFunctionKernel);
```

对每一个点分别计算M矩阵并保存到二维vector<Mat>中。

```

for(int row=0;row<srcImg.rows;row++)
{
    vector<Mat>tempMatVector;
    for(int col=0;col<srcImg.cols;col++)
    {
        Mat tempMat=(Mat_<float>(2,2)<<weightIxIx.at<float>(row,col),weightIxIy.at<float>(row,col),
                    weightIxIy.at<float>(row,col),weightIyIy.at<float>(row,col));
        tempMatVector.push_back(tempMat);
    }
    IMats.push_back(tempMatVector);
}

```

获取每一个点对应的M矩阵后，需要计算每一个角点相应的Response Function。

计算公式为： $R = \det(M) - \alpha \text{trace}(M)^2 = \lambda_1\lambda_2 - \alpha(\lambda_1 + \lambda_2)^2$ 。

$\alpha$ 是一个常量，通常取0.04到0.06，本次实现采用的是中间值0.05。

```

float eigenSum=0.0f, eigenProduct=0.0f;
eigenSum = mat2x2.at<float>(0,0)+mat2x2.at<float>(1,1);
eigenProduct = mat2x2.at<float>(0,0)*mat2x2.at<float>(1,1) \
              -mat2x2.at<float>(0,1)*mat2x2.at<float>(1,0);
cornerResponse = eigenProduct - alpha*pow(eigenSum,2);

```

最后通过非极大值抑制，获取更精确的角点、剔除角点周围的非角点，同时使用阈值进行二值化。

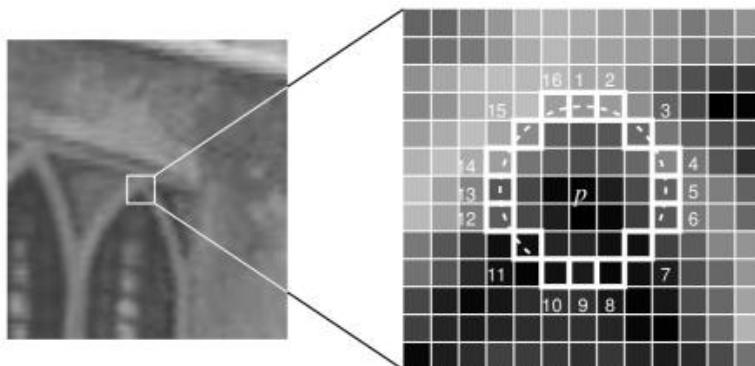
```

Mat cornerResponseNonMaxSuppress;
myNonMaxSuppressWithThreshold(cornerResponse,cornerResponseNonMaxSuppress,threshold);

```

### 2.3.1.2. FAST角点检测

在测试时，发现对于较大的图片，Harris虽然能够检测角点，但是由于该算法复杂度和没有进行并行优化的缘故，花费的时间较长。所以实现了更为快速的FAST角点检测算法。



FAST角点检测原理及步骤较为简单（配合上图进行说明）：

1. 在图像中任选一点p，假定其像素Intensity值为  $I_p$ 。
2. 以3个像素为半径画圆，得到p点周围的16个像素圆环。
3. 设定阈值threshold，如果这周围的16个像素中有连续的n个像素的像素值与  $I_p$  的差大于该阈值，那么该点就被判定为角点。实际n通常取12（16个像素周长的 3/4）。

本次实现的FAST角点检测的改进版本：首先检测p点的上下左右四个点，即编号为1, 5, 9, 12四个点，判断他们中是否有三个点和当前点像素值相差大于阈值。如果不满足，则直接跳过；如果满足，则继续使用前面的算法，全部判断剩余12个点中是否至少有9个满足条件。

首先定义了两个二维数组，分别存储圆环上点的与当前检测点的相对坐标：

```

const int KeyCircle[4][2] =
{
    {-3, 0},
    {0, -3}, {0, 3},
    {3, 0};

const int RestCircle[12][2] =
{
    {-3,-1}, {-3, 1},
    {-2,-2}, {-2, 2},
    {-1,-3}, {-1, 3},
    {1,-3}, {1, 3},
    {2,-2}, {2, 2},
    {3,-1},{3, 1};

```

```

if (keyCircleCounter < 3) // 对于关键圆上的点不能满足3个及以上就删除
    continue;

if (restCircleCounter < 9) // 对于剩余圆上的点不能满足9个及以上就删除
    continue;

// 通过上述检查的点进行记录
dstMat.at<unsigned char>(row, col) = srcImg.at<unsigned char>(row, col);

```

最后通过非极大值抑制，获取更精确的角点，并剔除角点周围的非角点。

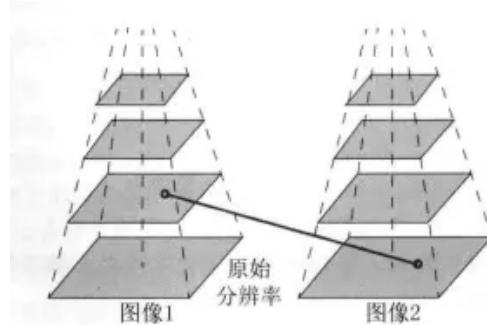
```
myNonMaxSuppress(srcMat, dstMat);
```

### 2.3.2. 描述子运算(Descriptor)

**Oriented FAST:**

ORB特征在FAST关键点的基础上还进行了改进，添加了尺度和旋转的特征，以保证特征点在缩放或旋转后仍能被检测。

尺度不变性主要由**图像金字塔**进行实现，相信用OpenCV做过图像处理的读者对此并不陌生。我们为图像建立图像金字塔，面对不同尺度下对特征点进行匹配，我们都可以在金字塔中寻找到对应的匹配，从而实现尺度不变性。



旋转不变性在ORB特征点由灰度质心法实现，其步骤如下

1. 在一块图像块B中，定义图像矩

$$m_{p,q} = \sum_{x,y \in B} x^p y^q I(x, y), p, q = \{0, 1\}$$

2. 通过矩的定义计算图像的质心  $C = \left( \frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right)$

3. 连接几何中心O与质心C，得到向量  $\overrightarrow{OC}$  定义特征方向为  $\theta = \arctan\left(\frac{m_{01}}{m_{10}}\right)$

如此一来，FAST角点就有了尺度和方向的描述，鲁棒性得到了一定的提升。

代码实现细节如下：

```

// 使用灰度质心计算特征点的方向
float m01 = 0, m10 = 0;
for (int dx = -half_patch_size; dx < half_patch_size; ++dx)
{

```

```

        for (int dy = -half_patch_size; dy < half_patch_size; ++dy)
    {
        uchar pixel = img.at<uchar>(kp.pt.y + dy, kp.pt.x + dx);
        m10 += dx * pixel;
        m01 += dy * pixel;
    }
}

float m_sqrt = sqrt(m01 * m01 + m10 * m10) + 1e-18; // 防止出现/0
float sin_theta = m01 / m_sqrt;
float cos_theta = m10 / m_sqrt;

```

### BRIEF描述子：

BRIEF描述子是一种二进制描述子，其描述向量由若干个0与1组成。其编码方式是随机选择周围两个像素p, q, 如果p的亮度高于q, 编码为1, 反之为0。我们重复这个过程n次, 就可以编码出一个n维描述向量。原始的BRIEF不具有旋转不变性, 但经过之前的FAST关键点改良后, 我们可以使用方向信息。ORB特征将BRIEF改良为Steer BRIEF, 使得描述子也具有旋转不变性。

代码实现细节如下:

```

// 计算描述子
vector<uint32_t> desc(8, 0);
for (int i = 0; i < 8; i++)
{
    uint32_t d = 0;
    for (int k = 0; k < 32; k++)
    {
        int idx_pq = i * 32 + k;
        cv::Point2f p(ORB_pattern[idx_pq * 4], ORB_pattern[idx_pq * 4 + 1]);
        cv::Point2f q(ORB_pattern[idx_pq * 4 + 2], ORB_pattern[idx_pq * 4 + 3]);

        // 将角度旋转到特征点的方向
        cv::Point2f pp = cv::Point2f(cos_theta * p.x - sin_theta * p.y, sin_theta * p.x + cos_theta * p.y) + kp.pt;
        cv::Point2f qq = cv::Point2f(cos_theta * q.x - sin_theta * q.y, sin_theta * q.x + cos_theta * q.y) + kp.pt;
        if (img.at<uchar>(pp.y, pp.x) < img.at<uchar>(qq.y, qq.x))
        {
            d |= 1 << k;
        }
    }
    desc[i] = d;
}

```

### 2.3.3. 特征点匹配(Matching)

有了特征点的构造方法, 下一步就是前后两帧特征点的匹配了。所谓特征匹配就是得到当前看到的路标与之前看到的路标的对应关系。据此对应关系我们才能根据几何学求解位姿的变化。下面简单介绍特征匹配。特征匹配最简单的思路自然是暴力匹配, 即将两张图象的特征点做二重遍历, 一一计算描述子并比较以找到对应关系。

代码实现细节如下:

```

void BfMatch(const vector<vector<uint32_t>> &desc1, const vector<vector<uint32_t>> &desc2,
vector<cv::DMatch> &matches)
{
    const int d_max = 40;

    for (size_t i1 = 0; i1 < desc1.size(); ++i1)
    {
        if (desc1[i1].empty())
            continue;
        cv::DMatch m{int}i1, 0, 256};
        for (size_t i2 = 0; i2 < desc2.size(); ++i2)

```

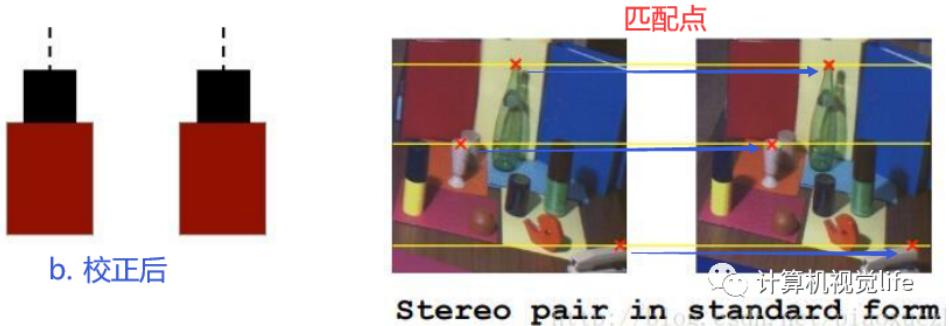
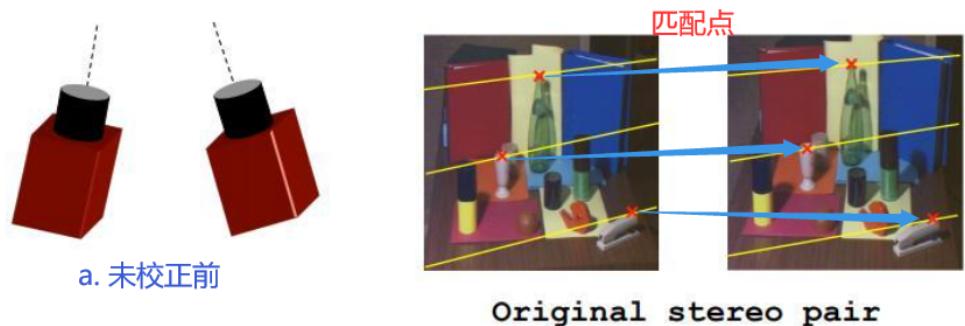
```

{
    if (desc2[i2].empty())
        continue;
    int distance = 0;
    for (int k = 0; k < 8; k++)
    {
        // 计算汉明距离
        distance += __builtin_popcount(desc1[i1][k] ^ desc2[i2][k]);
    }
    if (distance < d_max && distance < m.distance)
    {
        m.distance = distance;
        m.trainIdx = i2;
    }
}
if (m.distance < d_max)
{
    matches.push_back(m);
}
}
}
}

```

### 2.3.4. 极几何修正(Utilizing Epipolar Geometry)

由于双目相机已经经过矫正，所以左右两张图片极线已经对齐，可以保证两张图的匹配点都在同一条水平线上。因此，如果两个匹配点y坐标小于一定像素，则认为是正确的匹配结果，否则剔除。



```

// 通过极线判断匹配是否正确
vector<pair<pair<float, float>, pair<float, float>>> rectify_matches;
for (int i = 0; i < left_points.size(); i++)
{
    auto point_left = left_points[i];
    auto point_right = right_points[i];
    // 如果两个点y坐标小于一定像素，则认为是同一条极线
    if (abs(point_left.y - point_right.y) < 10)
    {
        rectify_matches.push_back(make_pair(make_pair(point_left.x, point_left.y),
        make_pair(point_right.x, point_right.y)));
    }
}

```

## 2.4. 稠密匹配(SGBM)

### 2.4.1 预处理

1. SGBM采用水平Sobel算子，把图像做处理，公式为：

$$Sobel(x, y) = 2[P(x + 1, y) - P(x - 1, y)] + P(x + 1, y - 1) - P(x - 1, y - 1) + P(x + 1, y + 1) - P(x - 1, y + 1)$$

2. 用一个函数将经过水平Sobel算子处理后的图像上每个像素点（P表示其像素值）映射成一个新的图像：PNEW表示新图像上的像素值。

映射函数：

$$P_{NEW} = \begin{cases} 0, & P < -\text{preFilterCap} \\ P + \text{preFilterCap}, & -\text{preFilterCap} \leq P \leq \text{preFilterCap} \\ 2 * \text{preFilterCap}, & P \geq \text{preFilterCap} \end{cases}$$

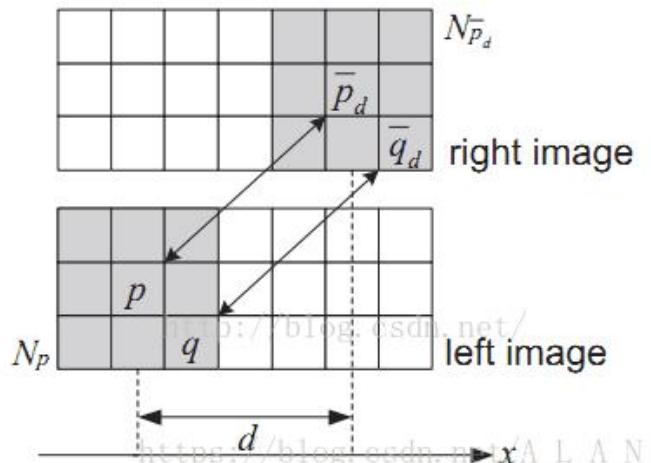
preFilterCap 为一个常数参数，opencv缺省情况下取15。预处理实际上是得到图像的梯度信息，将预处理的图像保存起来，将会用于计算代价。

### 2.4.2 代价计算

代价有两部分组成：

1. 经过预处理得到的图像的梯度信息经过基于采样的方法得到的梯度代价。

2. 原图像经过基于采样的方法得到的SAD代价。



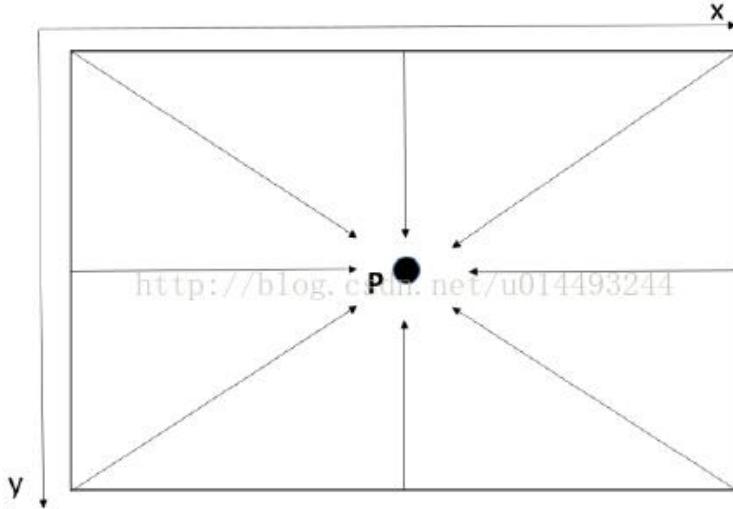
SAD(*sum of absolute differences*) :

$$C(x, y, d) = \sum_{i=-n}^n \sum_{j=-n}^n |L(x + i, y + j) - R(x + d + i, y + j)|$$

### 2.4.3 动态规划

动态规划算法本身存在拖尾效应，视差突变处易产生错误的匹配，利用态规划进行一维能量累加，会将错误的视差信息传播给后面的路径上。半全局算法利用多个方向上的信息，试图消除错误信息的干扰，能明显减弱动态规划算法产生的拖尾效应。

半全局算法试图通过影像上多个方向上一维路径的约束，来建立一个全局的马尔科夫能量方程，每个像素最终的匹配代价是所有路径信息的叠加，每个像素的视差选择都只是简单通过 WTA (Winner Takes All) 决定的。多方向能量聚集如下图所示：



在每个方向上按照动态规划的思想进行能量累积，然后将各个方向上的匹配代价相加得到总的匹配代价，如下式所示：

$$L_r(p, d) = c(p, d) + \min \left\{ \begin{array}{l} L_r(p - r, d) \\ L_r(p - r, d \pm 1) + P_1 \\ \min_{i=d_{\min}, \dots, d_{\max}} L_r(p - r, i) + P_2 \end{array} \right\} - \min_{i=d_{\min}, \dots, d_{\max}} L_r(p - r, i)$$

式中L为当前路径累积的代价函数，P1、P2为像素点与相邻点视差存在较小和较大差异情况下的平滑惩罚，P1<P2，第三项无意义，仅仅是为了消除各个方向路径长度不同造成的影响。将所有r方向的匹配代价相加得到总的匹配代价，如下：

$$s(p, d) = \sum_r L_r(p, d)$$

参数是这样设定的：

- P1 = 8cnsgbm.SADWindowSize\*sgbm.SADWindowSize;
- P2 = 32cnsgbm.SADWindowSize\*sgbm.SADWindowSize;
- cn是图像的通道数，SADWindowSize是SAD窗口大小，数值为奇数。

可以看出，当图像通道和SAD窗口确定下来，SGBM的规划参数P1和P2是常数。

## 2.4.4 后处理

opencvSGBM的后处理包含以下几个步骤：

1. 唯一性检测：视差窗口范围内最低代价是次低代价的(1 + uniquenessRatio/100)倍时，最低代价对应的视差值才是该像素点的视差，否则该像素点的视差为0。其中uniquenessRatio是一个常数参数。
2. 亚像素插值：

插值公式：

$$\text{denom2} = \frac{\max(Sp[d-1]+Sp[d+1]-2*Sp[d], 1)}{16}$$

$$d = d + \frac{(Sp[d-1]-Sp[d+1])+\text{denom 2}}{\text{denom 2}*2}$$

3. 误差阈值disp12MaxDiff默认为1，可以自己设置。

OpencvSGBM计算右视差图的方式：

通过得到的左视察图计算右视差图

## 2.4.5 SGBM算法的参数含义及数值选取

OPENCV中SGBM算法的参数含义及数值选取

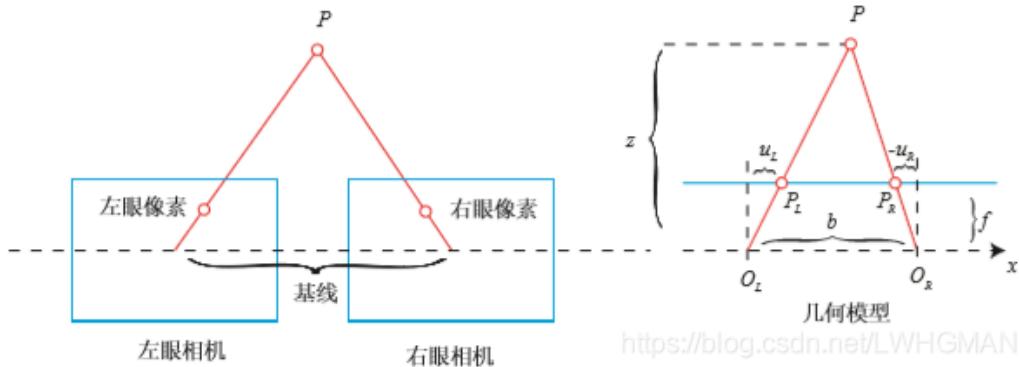
1. preFilterCap：水平sobel预处理后，映射滤波器大小。默认为15,opencv测试例程中取63。
2. SADWindowSize:计算代价步骤中SAD窗口的大小。由源码得，此窗口默认大小为5。
3. minDisparity：最小视差，默认为0。此参数决定左图中的像素点在右图匹配搜索的起点。
4. numberofDisparities：视差搜索范围，其值必须为16的整数倍
5. opencv测试例程中，P1 = 8cnsgbm.SADWindowSize\*sgbm.SADWindowSize;

6. opencv测试例程中， $P2 = 32 * \text{SADWindowSize} * \text{sgbm.SADWindowSize}$ ;
7. uniquenessRatio：唯一性检测参数。对于左图匹配像素点来说，先定义在 $\text{numberOfDisparities}$ 搜索区间内的最低代价为 $\text{mincost}$ ，次低代价为 $\text{secdmincost}$ 。opencv测试例程中， $\text{uniquenessRatio}=10$ 。
8. disp12MaxDiff：左右一致性检测最大容许误差阈值。
9. speckleWindowSize：视差连通区域像素点个数的大小。对于每一个视差点，当其连通区域的像素点个数小于 $\text{speckleWindowSize}$ 时，认为该视差值无效，是噪声。
10. speckleRange：视差连通条件，在计算一个视差点的连通区域时，当下一个像素点视差变化绝对值大于 $\text{speckleRange}$ 就认为下一个视差像素点和当前视差像素点是不连通的。

具体的参数设置如下：

```
// 初始化SBGM参数
cv::Ptr<cv::StereoSGBM> sgbm = cv::StereoSGBM::create(0, 16, 3);
sgbm->setPreFilterCap(63);
sgbm->setBlockSize(5);
sgbm->setP1(8 * 3 * 3);
sgbm->setP2(32 * 3 * 3);
sgbm->setMinDisparity(0);
sgbm->setNumDisparities(160);
sgbm->setUniquenessRatio(10);
sgbm->setSpeckleWindowSize(100);
sgbm->setSpeckleRange(32);
sgbm->setDisp12MaxDiff(1);
sgbm->setMode(cv::StereoSGBM::MODE_SGBM);
```

## 2.5. 视差及点云深度计算



图为双目相机的成像模型。 $O_L, O_R$ 为左右光圈中心，方框为成像平面， $f$ 为焦距。 $u_L$ 和 $u_R$ 为成像平面的坐标。请注意，按照图中坐标定义， $u_R$ 应该是负数，所以图中标出的距离为 $-u_R$ 。在左右双目相机中，我们可以把两个相机都看作针孔相机。它们是水平放置的，意味着两个相机的光圈中心都位于 $x$ 轴上。两者之间的距离称为双目相机的基线（记作 $b$ ），是双目相机的重要参数。

考虑一个空间点 $P$ ，它在左眼相机和右眼相机各成一像，记作 $P_L, P_R$ 。由于相机基线的存在，这两个成像位置是不同的。理想情况下，由于左右相机只在 $x$ 轴上有位移，所以 $P$ 的像也只在 $x$ 轴（对应图像的 $u$ 轴）上有差异。记它的左侧坐标为 $u_L$ ，右侧坐标为 $u_R$ ，几何关系如图右侧所示。根据 $\triangle P P_L P_R$ 和 $\triangle P O_L O_R$ 的相似关系，有

$$\frac{z-f}{z} = \frac{b-u_L+u_R}{b}$$

$$z = \frac{fb}{d}, \quad d = u_L - u_R$$

其中 $d$ 定义为左右图的横坐标之差，称为视差。根据视差，我们可以估计一个像素与相机之间的距离。视差与距离成反比：视差越大，距离越近。同时，由于视差最小为一个像素，于是双目的深度存在一个理论上的最大值，由 $fb$ 确定。我们看到，基线越长，双目能测到的最大距离就越远；反之，小型双目器件则只能测量很近的距离。相似地，我们人眼在看非常远的物体时（如很远的飞机），通常不能准确判断它的距离。

```
// 稠密建图, disp中存储了视差信息
vector<vector<4d>, Eigen::aligned_allocator<vector<4d>>> pointcloud; // 点云数组
// 便利像素点, 利用disparity和相机内参计算点深度信息, 将点存入数组中
for (int v = 0; v < left.rows; v++)
    for (int u = 0; u < left.cols; u++)
```

```

{
    if (disp.at<float>(v, u) <= 0.0 || disp.at<float>(v, u) >= 96.0)
        continue;
    vector4d point(0, 0, 0, left.at<uchar>(v, u) / 255.0); // 前三维为xyz,第四维为颜色
    // 根据双目模型计算 point 的位置
    double x = (u - cx) / fx;
    double y = (v - cy) / fy;
    double depth = fx * b / (disp.at<float>(v, u));
    point[0] = x * depth;
    point[1] = y * depth;
    point[2] = depth;
    pointcloud.push_back(point);
}

// 稀疏建图, 视差通过匹配点的坐标差计算出来
vector<vector4d, Eigen::aligned_allocator<vector4d>> pointcloud;
// 便利像素点, 利用disparity和相机内参计算点深度信息, 将点存入数组中
for (auto match : good_matches)
{
    auto point_left = match.first;
    auto point_right = match.second;
    // 计算disparity
    double disparity = point_left.first - point_right.first;
    // 计算点深度信息
    double depth = (b * fx) / disparity;
    // 获取灰度信息
    double gray_left = left.at<uchar>((int)point_left.second, (int)point_left.first) / 255.0;
    // 将点存入数组中
    vector4d point = vector4d(0, 0, 0, gray_left); // 前三维为xyz,第四维为颜色
    double x = (point_left.first - cx) / fx;
    double y = (point_left.second - cy) / fy;
    point[0] = x * depth;
    point[1] = y * depth;
    point[2] = depth;
    pointcloud.push_back(point);
}

```

## 2.6. 3D点云可视化

### 2.6.1. Pangolin库介绍

Pangolin 是一组轻量级和可移植的实用程序库，用于制作基于 3D、数字或视频的程序和算法的原型。它在计算机视觉领域被广泛使用，作为删除特定于平台的样板并使数据可视化变得容易的一种手段。

Pangolin 的总体精神是通过简单的界面和工厂，而不是窗口和视频，最大限度地减少样板文件并最大限度地提高可移植性和灵活性。它还提供了一套用于交互式调试的实用程序，例如 3D 操作、绘图仪、调整变量，以及用于 python 脚本和实时调整的下拉式类似 Quake 的控制台。

### 2.6.2. 代码细节展示

```

void showPointCloud(const std::vector<Eigen::Vector4d,
Eigen::aligned_allocator<Eigen::Vector4d>> &pointcloud, int point_size)
{

    if (pointcloud.empty())
    {
        std::cerr << "Point cloud is empty!" << std::endl;
        return;
    }
    // 创建窗口
    pangolin::CreateWindowAndBind("Point Cloud Viewer", 1024, 768);
    glEnable(GL_DEPTH_TEST); // 3d必开, 指现实镜头视角的像素
    glEnable(GL_BLEND); //颜色混合, 透过蓝玻璃看黄玻璃变绿
}

```

```

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); //设置上一行的混合方式

pangolin::OpenGLRenderState s_cam(
    pangolin::ProjectionMatrix(1024, 768, 500, 500, 512, 389, 0.1, 1000),
    pangolin::ModelViewLookAt(0, -0.1, -1.8, 0, 0, 0, 0.0, -1.0, 0.0));

pangolin::View &d_cam = pangolin::CreateDisplay()
    .SetBounds(0.0, 1.0, pangolin::Attach::Pix(175), 1.0, -1024.0f /
768.0f)
    .SetHandler(new pangolin::Handler3D(s_cam)); //创建相机视图句柄

while (pangolin::ShouldQuit() == false)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); //清空颜色和深度缓存。这样每次都会刷新显示，不至于前后帧的颜色信息相互干扰。
    d_cam.Activate(s_cam); //激活显示并设置状态矩阵
    glColor3f(1.0f, 1.0f, 1.0f, 1.0f); //前三个分别代表红、绿、蓝所占的分量，范围从0.0f~1.0f，最后一个参数是透明度Alpha值，范围也是0.0f~1.0f
    glPointSize(point_size); //设置点的大小
    glBegin(GL_POINTS); //开始绘制点云
    for (auto &p : pointcloud)
    {
        glColor3f(p[3], p[3], p[3]);
        glVertex3d(p[0], p[1], p[2]);
    }
    glEnd(); //结束绘制点云
    pangolin::FinishFrame();
    usleep(5000); //一帧5ms
}
return;
}

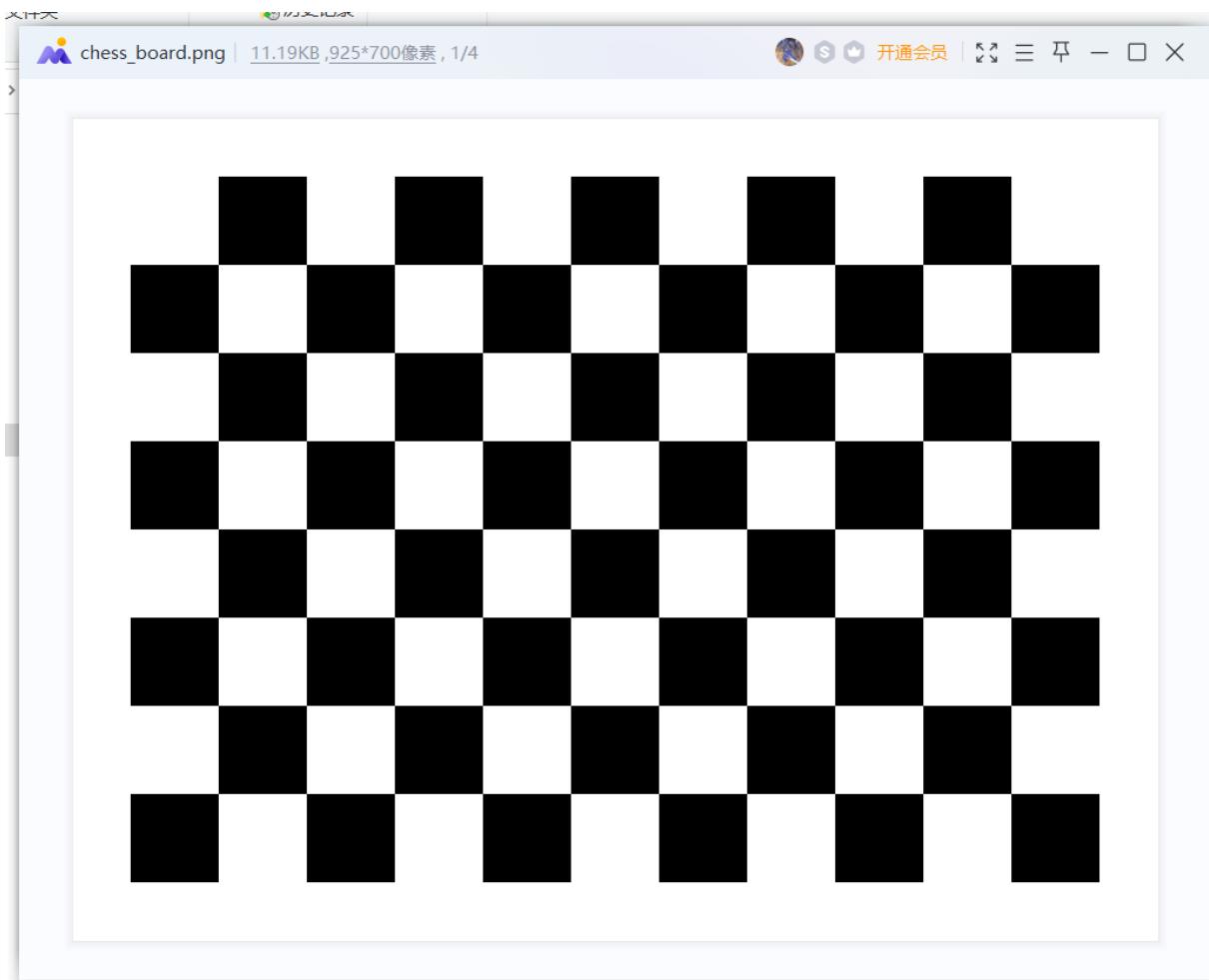
```

### 3. 实验过程及结果

#### 3.1. 相机标定及图像校正实验

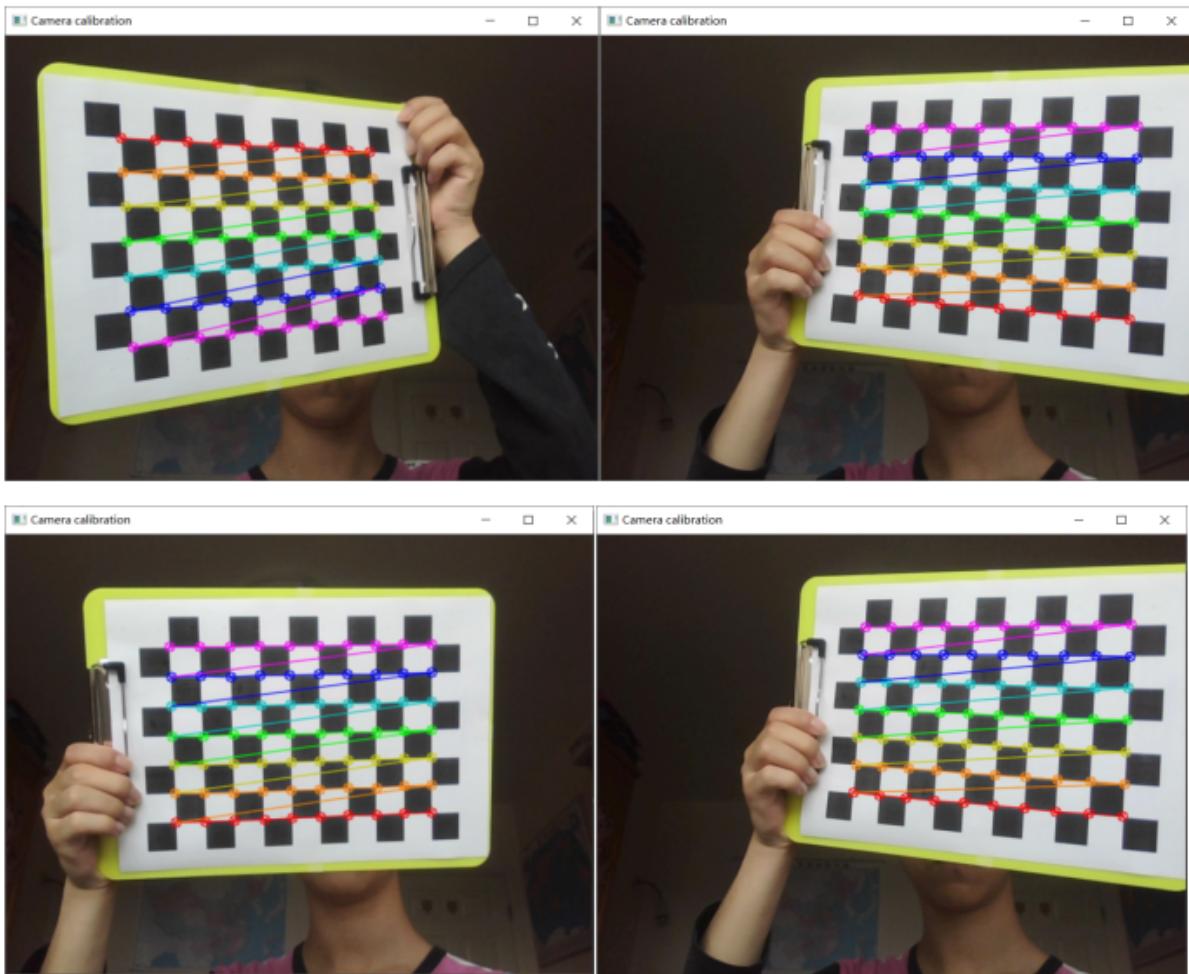
##### 3.1.1. 绘制标定板

输出的棋盘格标定板图片如下所示：



### 3.1.2. 单目像机标定

单目摄像机标定结果图:



输出内参矩阵和畸变系数：

```
C:\Users\CHP\source\repos\camera_calibration\x64\Debug\camera_calibration.exe
开始提取角点.....  

image_size.width = 640  

image_size.height = 480  

image_sum_nums = 16  

每幅图像的标定误差：  

第1幅图像的平均误差为： 0.0469963像素  

第2幅图像的平均误差为： 0.0766429像素  

第3幅图像的平均误差为： 0.0556367像素  

第4幅图像的平均误差为： 0.0692552像素  

第5幅图像的平均误差为： 0.0569692像素  

第6幅图像的平均误差为： 0.0721352像素  

第7幅图像的平均误差为： 0.0694213像素  

第8幅图像的平均误差为： 0.0598278像素  

第9幅图像的平均误差为： 0.0810246像素  

第10幅图像的平均误差为： 0.0782893像素  

第11幅图像的平均误差为： 0.0693382像素  

第12幅图像的平均误差为： 0.0690632像素  

第13幅图像的平均误差为： 0.059898像素  

第14幅图像的平均误差为： 0.025346像素  

第15幅图像的平均误差为： 0.0286113像素  

第16幅图像的平均误差为： 0.0258597像素  

总体平均误差为： 0.0590197像素  

相机内参数矩阵：  

[640.6496513562538, 0, 319.4051331834634;  

 0, 641.3254543538873, 238.0976204122528;  

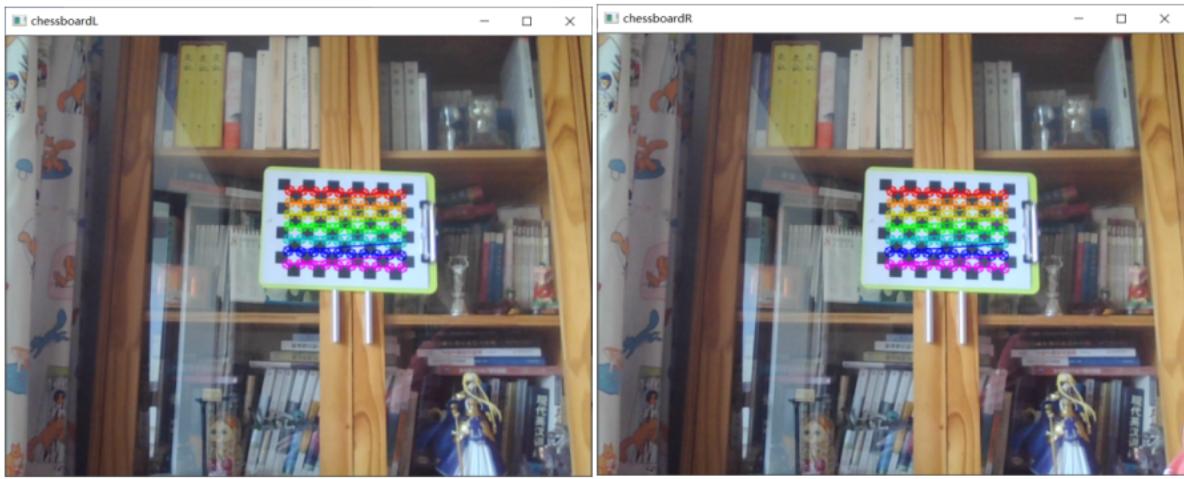
 0, 0, 1]  

畸变系数：  

[-0.03651462735638861, 0.3470364756706837, 0.007867400590965998, -0.002879643072168816, -0.9009857643404001]
```

### 3.1.3. 双目像机标定

单目摄像机标定结果图：



左右两相机的所有内参、外参及误差:

```
C:\Users\CHP\source\repos\camera_calibration\x64\Debug\camera_calibration.exe

标定成功
立体标定误差RMS为 0.127374
左相机内参矩阵: [491.3022935009177, 0, 330.3387192652675;
 0, 500.4573667832998, 298.3946678572051;
 0, 0, 1]
左相机畸变系数: [0.5878469248783409, -6.039475350938682, 0.03428167246718149, 0.00272346
 4563163268, 24.61756318335322]
右相机内参矩阵: [491.3022935009177, 0, 330.3387192652675;
 0, 500.4573667832998, 298.3946678572051;
 0, 0, 1]
右相机畸变系数: [0.5878469248783409, -6.039475350938682, 0.03428167246718149, 0.00272346
 4563163268, 24.61756318335322]
R=[0.999831920549231, 0.003591315752583765, -0.01797868465715458;
 -0.003903399055382658, 0.9998417950403066, -0.01735362689445831;
 0.01791351798644252, 0.0174208880870888, 0.9996877605190565]
T=[67.81971828436726;
 -10.94302286226466;
 -65.81699683617596]
R1=[0.7007965405721109, -0.1244968811482927, -0.7024135073491321;
 0.1173911037222408, 0.9913559907787387, -0.05858863638948391;
 0.7036359410156898, -0.04139838319853731, 0.7093536751011525]
Rr[0.71286011441161, -0.1150232517465497, -0.6918093009194918;
 0.1219849916942427, 0.9917576543569542, -0.03919715328627267;
 0.6906157535724345, -0.05644826465040231, 0.721015585362436]
P1[500.4573667832998, 0, 817.1695823669434, 0;
 0, 500.4573667832998, 263.9345535635948, 0;
 0, 0, 1, 0]
Pr[500.4573667832998, 0, 817.1695823669434, 47612.2551148681;
 0, 500.4573667832998, 263.9345535635948, 0;
 0, 0, 1, 0]
Q[1, 0, 0, -817.1695823669434;
 0, 1, 0, -263.9345535635948;
 0, 0, 0, 500.4573667832998;
 0, 0, -0.01051110403352896, 0]
Painted ImageL
Painted ImageR
wait key
```

### 3.1.4. 双目像机图像矫正



矫正后的左右图像画在画布上进行左右比对



注：本实验由于是用一台笔记本电脑自带摄像头拍摄的伪双目图像，所以拍摄误差很大，故标定和矫正结果也会有相应的误差。

### 3.2. 角点检测实验

在角点检测实验中，我们分别在测试了Harris及FAST角点检测算法使用不同阈值在下多张图片的表现，结果展示如下：

Harris 角点检测

Harris 角点检测结果

threshold	theater	chair
threshold=500		
角点数量	1555	895
threshold=1500		
角点数量	927	610
threshold=3000		
角点数量	652	459

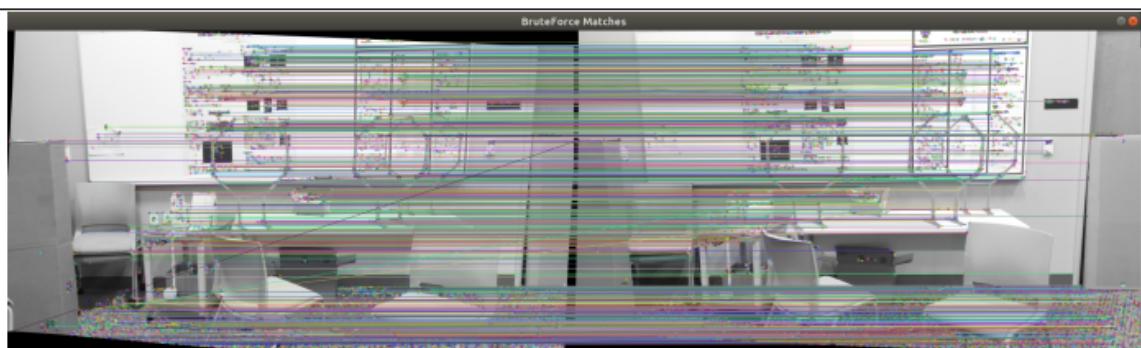
FAST 角点检测

FAST 角点检测结果		
	theater	chair
threshold=10		
角点数量	8341	12722
threshold=30		
角点数量	2095	2448
threshold=50		
角点数量	535	494

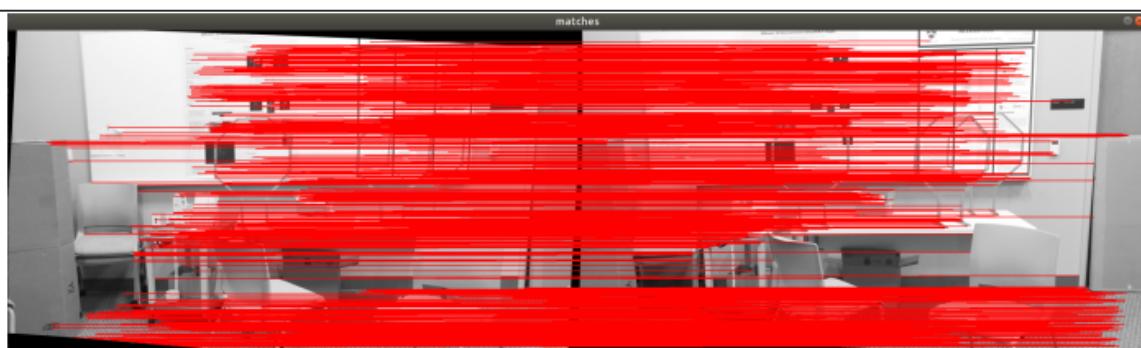
### 3.3. 稀疏点云建模实验

在稀疏点云建模的实验中，

关键点匹配结果（包含误匹配）



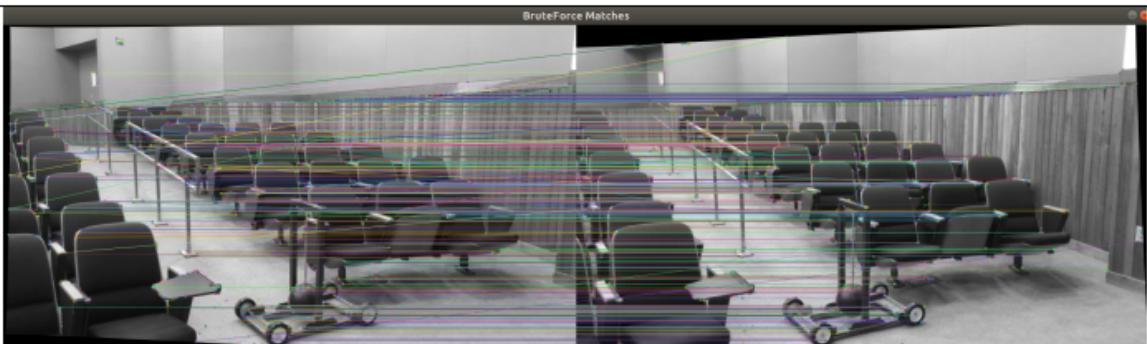
关键点匹配结果（剔除误匹配）



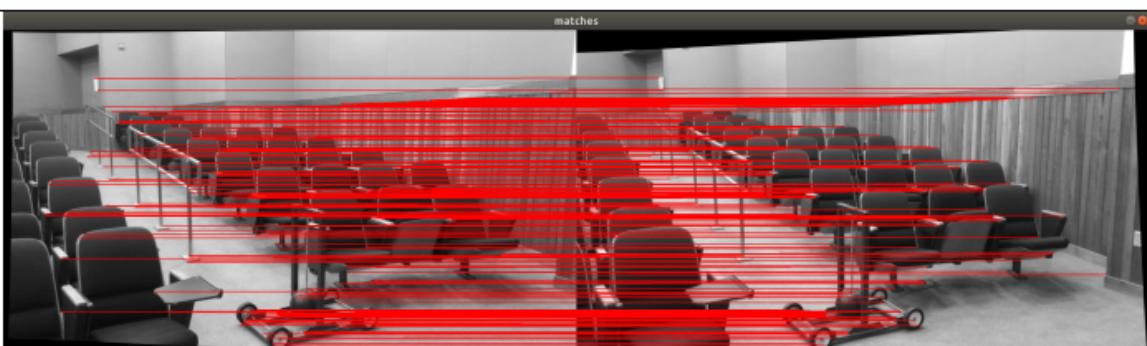
生成的稀疏点云图



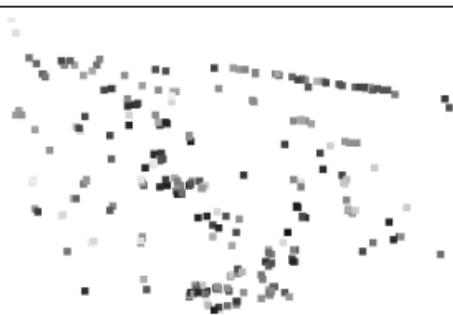
关键点匹配结果（包含误匹配）



关键点匹配结果（剔除误匹配）



生成的稀疏点云图



### 3.4. 稠密点云建模实验

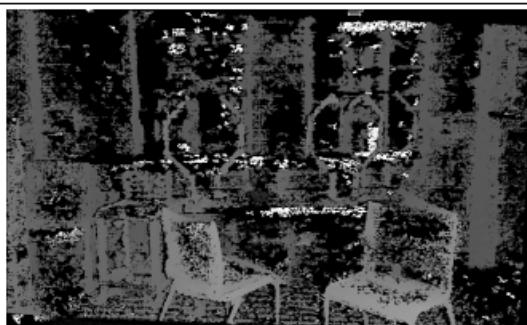
在稠密点云建模实验中，我们首先使用自己实现的角点检测，并进行关键点匹配，通过极几何对双目相机所拍摄的图片进行修正。随后计算匹配的关键点的视差，并由此最终生成稠密点云图。

桌椅图片的实验结果：

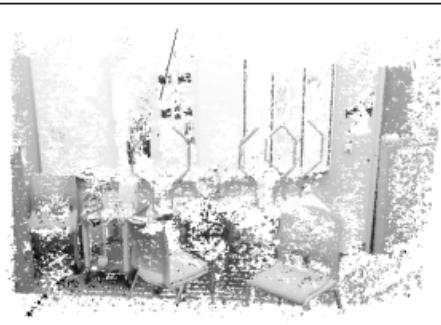
极几何修正后双目相机拍摄图片



双目相机视差图

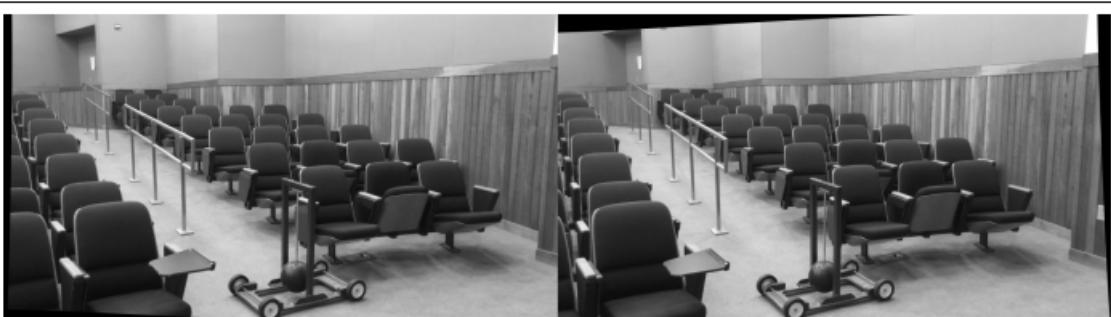


点云图

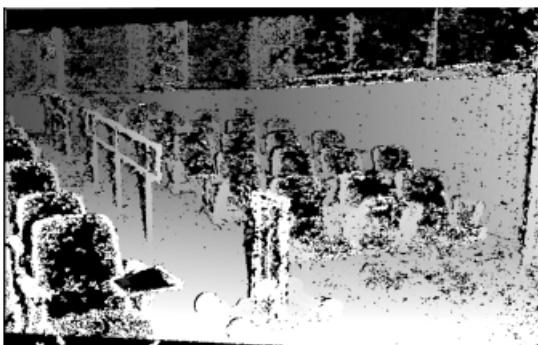


剧院座椅图片的实验结果：

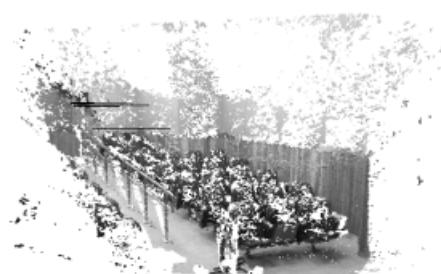
极几何修正后双目相机拍摄图片



双目相机视差图



点云图



可以发现，双目相机的视差图和稠密点云图与实际情况基本一致。

## 4. 应用背景分析或展望

随着自动化程度越来越高，机器视觉扮演着越来越重要的角色，传统的2D定位无法解决产品的空间坐标信息，而3D双目立体视觉可提供较高精度的定位。完成2D定位不能实现的功能，大大提高了生成效率，降低劳动力。随着工业机器人越来越广泛的应用，3D双目立体视觉将具有更广阔的应用前景。

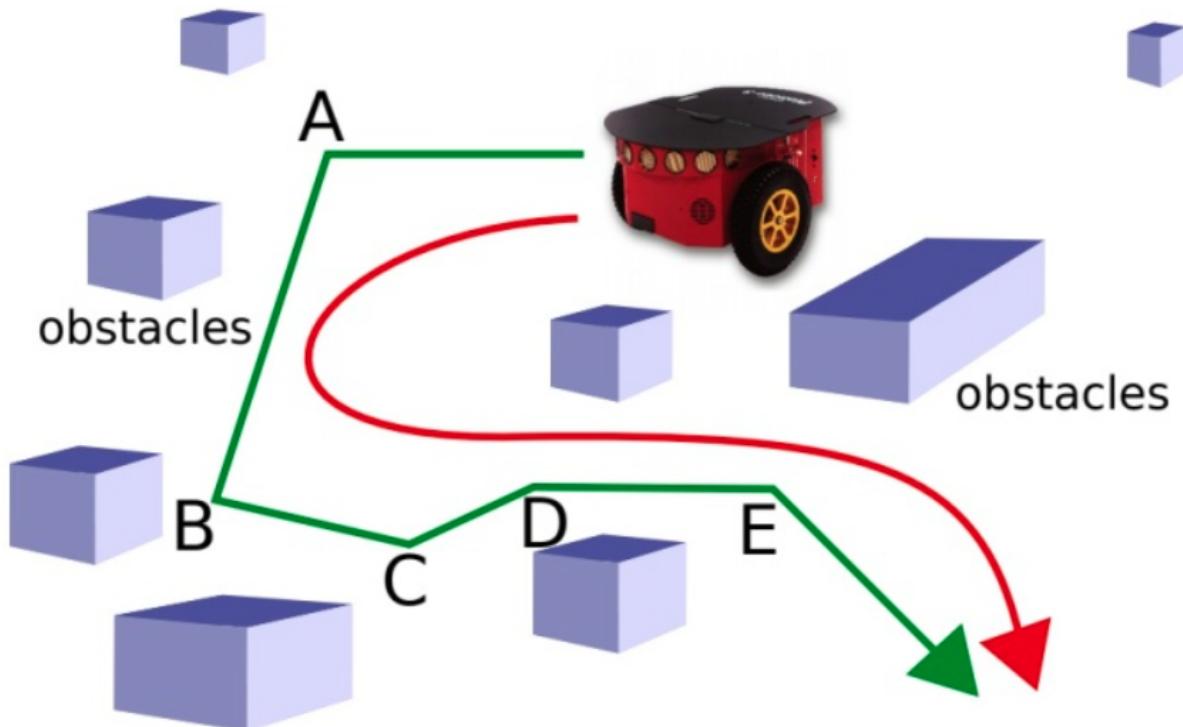
双目立体视觉不但具有精度高、扩展能力强大等优点，还具有连续工作时间长、不易损坏、保密性好、没有培训成本、结果易于保存和复制等优点。随着光学、电子学以及计算机技术的发展，将不断进步，逐渐实用化，不仅将成为工业检测、生物医学、虚拟现实等领域。而且，双目立体视觉技术广泛应用生产、生活中，为工业发展做出重要的贡献。

制造业的发展，带来了对机器视觉需求的提升，也决定了机器视觉将由过去单纯的采集、分析、传递数据，判断动作，逐渐朝着开放性的方向发展，这一趋势也预示着机器视觉将与自动化更进一步的融合。

双目立体视觉目前主要应用领域：

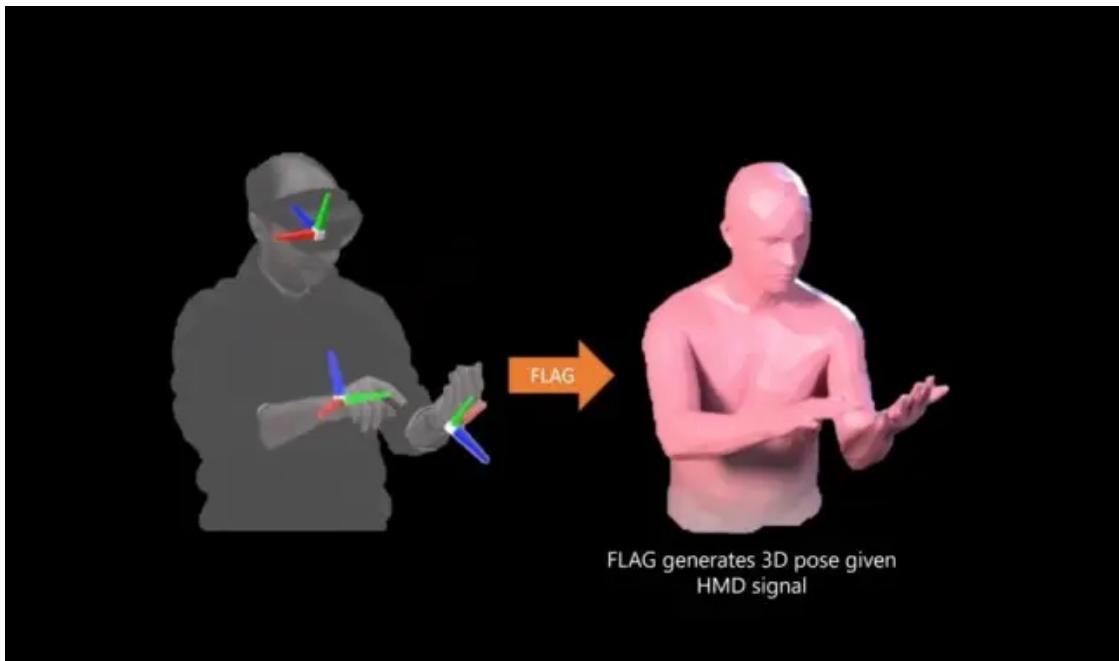
### 机器人导航

利用实时双目立体视觉和机器人整体姿态信息集成，开发仿真机器人动态行走导航系统。该系统实现分两个步骤：首先，利用平面分割算法分离所拍摄图像对中的地面与障碍物，再结合机器人身体姿态的信息，将图像从摄像机的二维平面坐标系转换到描述躯体姿态的世界坐标系，建立机器人周围区域的地图；其次根据实时建立的地图进行障碍物检测，从而确定机器人的行走方向。



### 三维测量和虚拟现实

虚拟现实的特点是人能身临其境地与三维计算机世界进行人机交互，为实现这一技术，需要众多相关技术的支持，其中三维测量技术是虚拟现实环境得以模拟实现的重要环节，是人机交互的关键所在。三维测量的范围大小、准确性、实时性及复杂性等都将极大的影响虚拟现实系统的性能。



### 车外环境识别

检测车辆外形、速度，还有远处建筑物的位置等由于汽车GPS定位辅助功能。目前我们已经实现的是多台相机在道路口的检测。譬如自动倒车控制软件，就是利用双目立体相机检测车尾部的空间和距离，用于辅助倒车入位。



此外，立体视觉还在航空测绘、军事运用、医学成像和工业检测等领域中的运用将越来越广泛。