双目测距

双目标定

重点： 两个摄像头一定要固定好。镜头不要求严格平行，但是要相互静止、

双目拍照

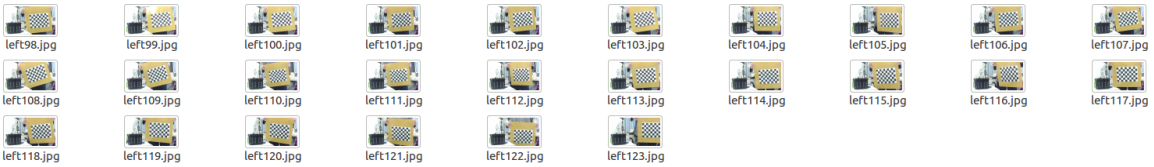
自己动手，可以省的麻烦别人

```
double t1 = getTickCount();
double t2;
int delta_t;
int a = 1;

while (1){
    cap1 >> src1;
```

```
    cap2 >> src2;

    t2 = getTickCount();
    delta_t = (int)((t2 - t1) / getTickFrequency() * 1000);
    if (delta_t > 1000){
        string aa = to_string(a);
        a++;
        imwrite("xxx/camera1/left" + aa + ".jpg", src1);
        imwrite("xxx/camera2/right" + aa + ".jpg", src2);
        t1 = t2;
    }
```

用这个程序来每隔一秒拍一张照，就不用麻烦自己的队友来当模特了，主要是双目对误差的要求很严格，拍照的时候最好是每个角度停多几秒。

最后的效果就是类似于

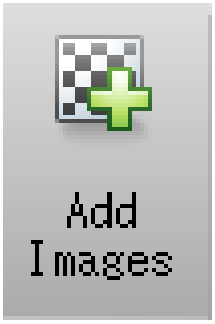


matlab标定

1. 在matlab软件里面打开 Stereo Camera Calibrator

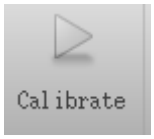
2. 然后设置，需要计算的参数，如图

3. 点击添加图片

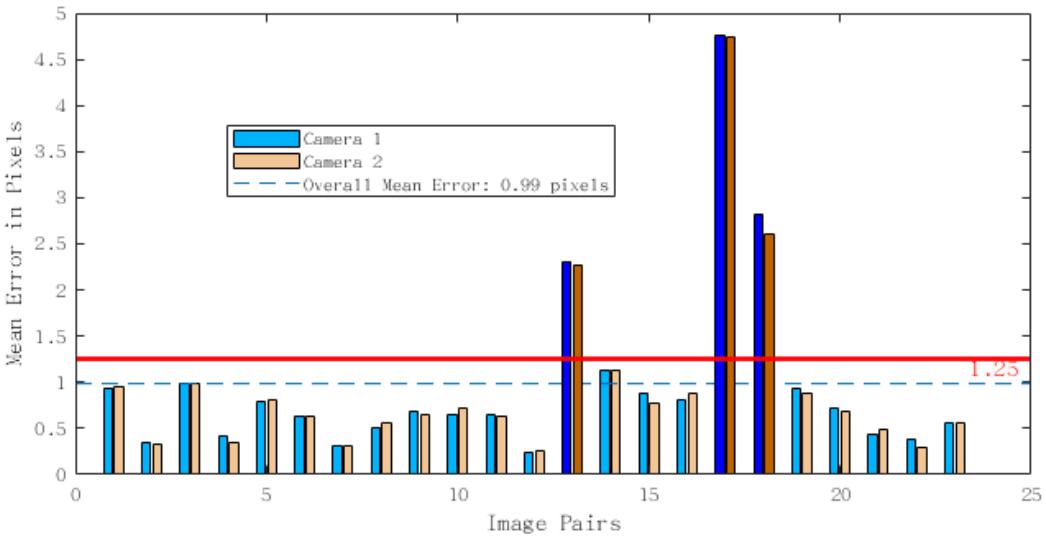


4. 填写左右相机图片的文件夹目录

5. 自动筛选完图片之后，点击标定按钮

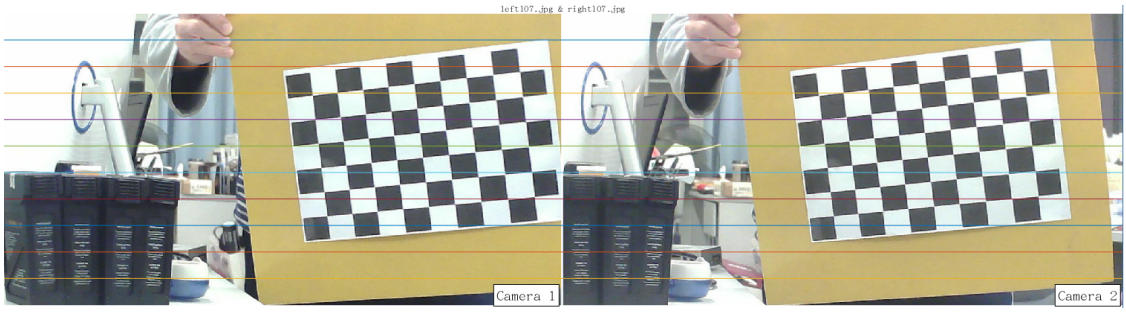


6. 手动拉动误差线，最好是将线固定在0.25-0.3左右，然后按delate键，删除掉所有的高于这个误差的图片，我这个图片是随便的给的，正常要误差拉低一点。



7. 点击 show rectify， 可以发现，两张图片变得平行了，得出结论，只要严格按照他帮你计算出来的参数去矫正图片，一定是可以矫正成功的，所以如果我们后面在矫正的时候，发现不能把两个点矫正

到同一水平线，那么只能是你的矫正的过程写错了



8. 把标定好的参数输出到工作台

保存标定参数

把标定的参数手打输入进我们的C++代码里面太麻烦了，而且如果原理了解的不扎实，还有可能输错，所以我们采用将参数保存为 .xml 文件，并且后续可以直接导入进C++当中。

1. 直接把 writeXML.m 文件放到当前的matlab工作目录下（由陈晓嘉学长编写）

2. 直接在matlab的命令行窗口输入

```
writeXML(stereoParams, '/xx/stereoparams.xml')
```

测距

网上的教程我最喜欢这两篇，对立体矫正的参数讲解的最好，看《**学习opencv3**》也行

至于测距的原理其实很简单，就是简单的利用相似三角形而已

reference

reference

需要注意的是我们不使用匹配算法，因为我们的识别目标是带有特征的，直接处理光点即可

在C++中导入参数

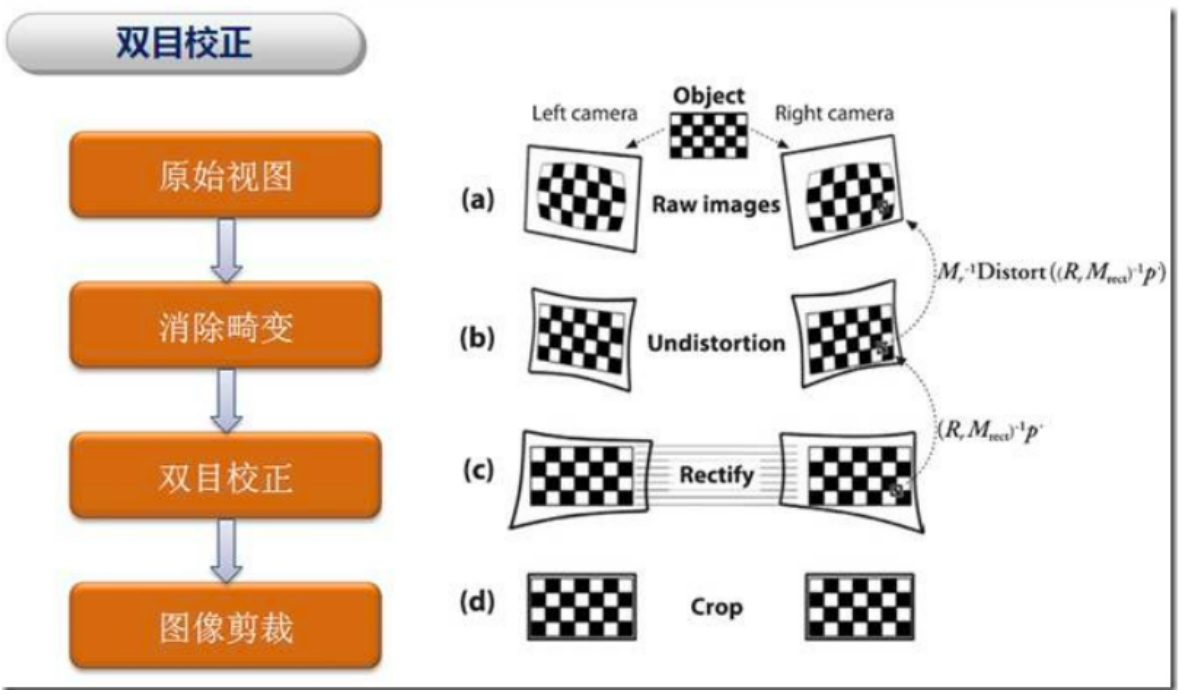
```
#define STEREOPARAMS_PATH "xxx"

FileStorage cameraYaml(STEREOPARAMS_PATH,FileStorage::READ);

cameraYaml["cameraMatrixL"] >> cameraMatrixL;
cameraYaml["cameraMatrixR"] >> cameraMatrixR;
cameraYaml["distCoeffL"]    >> distCoeffL;
cameraYaml["distCoeffR"]    >> distCoeffR;
cameraYaml["T"]             >> T;
cameraYaml["R"]             >> R;
```

取出必要参数

```
Size imageSize = Size(VIDEO_WIDTH, VIDEO_HEIGHT);

// 计算立体矫正参数
stereoRectify(cameraMatrixL, distCoeffL,
cameraMatrixR, distCoeffR,
imageSize, R, T, Rl, Rr, Pl, Pr, Q,
                  CALIB_ZERO_DISPARITY, -1,
imageSize);

b   = abs(T.at<double>(0));     // 两相机的基线距离
cx = abs(Q.at<double>(3));     // 矫正之后的光心X的坐标
cy = abs(Q.at<double>(7));     // 矫正之后的光心Y的坐标
f   = abs(Q.at<double>(11));    // 矫正之后两相机的焦距

```

矫正

这里默认相机的畸变影响不是很大，不作畸变和径向等等的矫正



矫正全图（弃用）

缺点还是很多的，比如可能在没识别到目标的时候，也在矫正图片，这是很没必要的浪费处理图片的速度.

注意，如果要使用全图矫正，记得自己去手调立体矫正时候的裁剪系数

全图映射

```
// 计算左右两图的矫正系数
initUndistortRectifyMap(cameraMatrixL, distCoeffL, Rl, Pr, outputsize, CV_32FC1, mapLx, mapLy);
initUndistortRectifyMap(cameraMatrixR, distCoeffR, Rr, Pr, outputsize, CV_32FC1, mapRx, mapRy);

// 把左右的原图转为灰度图
cvtColor(rgbImageL, grayImageL, CV_BGR2GRAY);
cvtColor(rgbImageR, grayImageR, CV_BGR2GRAY);

// 用remap函数进行映射，得到矫正后的图
remap(grayImageL, rectifyImageL, mapLx, mapLy, INTER_LINEAR);
remap(grayImageR, rectifyImageR, mapRx, mapRy, INTER_LINEAR);

// 将矫正后的图转换为 伪彩色图（没什么效果） 用于后续操作
//（不过说实话我做这一步的目的主要就是把图片变回三通道，那么就不用改动识别的代码了）
Mat rgbRectifyImageL, rgbRectifyImageR;
cvtColor(rectifyImageL, rgbRectifyImageL, CV_GRAY2BGR);
cvtColor(rectifyImageR, rgbRectifyImageR, CV_GRAY2BGR);
```

然后在rgbRectifyImageL和rgbRectifyImageR里面找到我们需要测距的 left_pt 和 right_pt

稀疏矫正

优点很多，比如单单仅矫正我们需要测距的两个点至同一水平线即可，提高了处理的速度。

需要注意的是，与全图矫正的区别，我们是识别到两个目标点了，再进入这个矫正算法，顺序是不同的，这样我们就可以，在未识别到目标的时候，不要进行矫正了，从另一方面也提高的处理的速度。

```
// 由于函数的需要，输入的点需要是向量的形式
vector<Point2f>  left_point_src;
vector<Point2f>  left_point_dst;
vector<Point2f> right_point_src;
vector<Point2f> right_point_dst;

// 由于我们仅仅需要一组点就够了。故向量的大小为1即可
if (left_point_src.size() < 1) {
    left_point_src.push_back(left_pt);
}else{
    left_point_src.pop_back();
    left_point_src.push_back(left_pt);
}
if (right_point_src.size() < 1){
    right_point_src.push_back(right_pt);
}else{
    right_point_src.pop_back();
    right_point_src.push_back(right_pt);
}

// 用undistortPoints函数，来进行单独
```

```
undistortPoints(left_point_src , left_point_dst , cameraMatrixL, distCoeffL, Rl, Pl);
undistortPoints(right_point_src, right_point_dst, cameraMatrixR, distCoeffR, Rr, Pr);

// 将矫正后的点，重新赋值给left_pt和right_pt
left_pt = left_point_dst[0];
right_pt = right_point_dst[0];
```

矫正检验

1. 对于全图矫正的校验比较直观，但是也比较麻烦

```
// 显示在同一张图上
Mat canvas;
double sf;
int w, h;
sf = 600. / MAX(imageSize.width, imageSize.height);
w = cvRound(imageSize.width * sf);
h = cvRound(imageSize.height * sf);
canvas.create(h, w * 2, CV_8UC3);   //注意通道

// 左图像画到画布上
Mat canvasPart = canvas(Rect(w * 0, 0, w, h));
                        //得到画布的一部分
resize(rgbRectifyImageL, canvasPart, canvasPart.size(), 0, 0, INTER_AREA);      //把图像缩放到跟canvasPart一样大小

//右图像画到画布上
canvasPart = canvas(Rect(w, 0, w, h));
                      //获得画布的另一部分
resize(rgbRectifyImageR, canvasPart, canvasPart.size(), 0, 0, INTER_LINEAR);

```

```
//画上对应的线条
for (int i = 0; i < canvas.rows; i += 16)
    line(canvas, Point(0, i), Point(canvas.cols, i), Scalar(0, 255, 0), 1, 8);
imshow("rectified", canvas);
```

2. 只用能笨办法，在矫正的前后把left_pt和right_pt输出一下即可

```
cout<<"left_pt_src:  "<<left_pt<<"   right_pt_src:  "<<right_pt<<endl;
/*
你的矫正
*/
cout<<"left_pt_dst:  "<<left_pt<<"   right_pt_dst:  "<<right_pt<<endl;
```

计算

```
// 最核心的计算步骤，计算视差d
xl = left_pt.x -  cx;
xr = right_pt.x -  cx;
d = xl - xr;
// 这里不这么做也可以，我只是为了误差更小而已
Y = ((cy - left_pt.y) + (cy - right_pt.y)) / 2;

// 用相似三角形的原理去求 x， y， z的值（推导过程到处都是）
x = xl * b / d;
y = Y * b / d;
z = b * f / d;
```