基于 SIFT 关键点检测的图像匹配

摘要:

图像匹配是找到两幅不同图像之间的空间位置关系，是实现图像融合，目标识别的关键步骤。图像匹配主要可分为以灰度为基础的匹配和以特征为基础的匹配。图像特征匹配有三个基本步骤：特征提取，特征描述和特征匹配。特征提取就是从图像中提取关键点，特征描述是用一组数学向量对特征点进行描述，主要保证不同的向量和不同特征点之间是一种对应关系，同时相似的关键点差异尽可能小。特征匹配就是特征向量之间的距离计算。

SIFT 算法是一种基于尺度空间的算法，对于旋转和尺度均有不变性，并且对于噪声、视角变化和光照变化具有良好的**鲁棒性**，同时 SIFT 算法具有独特性好、信息量丰富、运行速度快等特点。

本文中采用 SIFT 关键点检测算法实现图像特征匹配，实验在**双目图像匹配**、**目标识别**问题上的效果。

关键字：图像匹配，SIFT 算法，鲁棒性，双目图像匹配，目标识别

目录

SIFT 算法简介

SIFT（Scale Invariant Feature Transform，尺度不变特征变换匹配算法）是由 David G. Lowe 教授在 1999 年（《Object Recognition from Local Scale-Invariant Features》）提出的高效区域检测算法，在 2004 年（《Distinctive Image Features from Scale-Invariant Keypoints》）得以完善。

SIFT 算法的特点:

1.稳定性：SIFT 特征是图像的局部特征，其对旋转、尺度缩放、亮度变化保持不变性，对视角变化、仿射变换、噪声也保持一定程度的稳定性；

2.独特性：信息量丰富，适用于在海量特征数据库中进行快速、准确的匹配；

3.多量性：即使少数的几个物体也可以产生大量的 SIFT 特征向量；

4.高速性：经优化的 SIFT 匹配算法甚至可以达到实时的要求；

5.可扩展性：可以很方便的与其他形式的特征向量进行联合。

SIFT 算法在一定程度上可解决：

1. 目标的旋转、缩放、平移

2. 图像仿射/投影变换

3. 光照影响

4. 目标遮挡

5. 杂物场景

SIFT 算法原理

SIFT 算法的实质是在不同的尺度空间上查找关键点，并计算出关键点的方向。SIFT 所查找到的关键点是一些十分突出，不会因光照，仿射变换和噪音等因素而变化的点，如角点、边缘点、暗区的亮点及亮区的暗点等。

SIFT 算法分解为以下四步：

1. 尺度空间极值检测：搜索所有尺度上的图像位置。通过高斯微分函数来识别潜在的对于尺度和旋转不变的兴趣点。

2. 关键点定位：在每个候选的位置上，通过一个拟合精细的模型来确定位置和尺度。关键点的选择依据于它们的稳定程度。

3. 方向确定：基于图像局部的梯度方向，分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向、尺度和位置进行变换，从而提供对于这些变换的不变性。

4. 关键点描述：在每个关键点周围的邻域内，在选定的尺度上测量图像局部的梯度。这些梯度被变换成一种表示，这种表示允许比较大的局部形状的变形和光照变化。

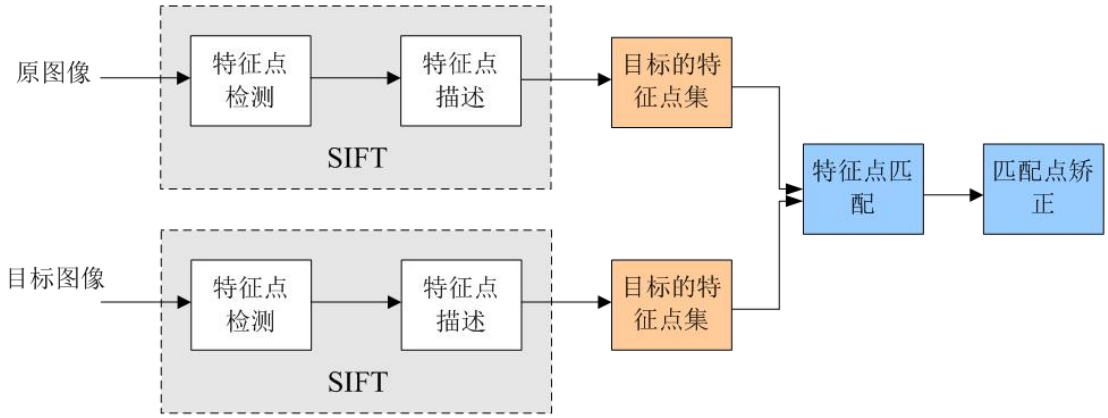


Figure 1 ： SIFT 算法流程

检测尺度空间极值

检测尺度空间极值就是搜索所有尺度上的图像位置，通过高斯微分函数来识别对于尺度和旋转不变的兴趣点。其主要步骤可分为建立高斯金字塔、生成 DOG 高斯差分金字塔和 DOG 局部极值点检测。

(1) 建立高斯金字塔

尺度空间在实现时，使用高斯金字塔表示，高斯金字塔的构建分为两部分：

1.对图像做不同尺度的高斯模糊

2.对图像做降采样（隔点采样）

为了让尺度体现其连续性，高斯金字塔在简单降采样的基础上加上了高斯滤波。将图像金字塔每层的一张图像使用不同参数做高斯模糊，使得金字塔的每层含有多张高斯模糊图像，将金字塔每层多张图像合称为一组(Octave)，金字塔每层只有一组图像，组数和金字塔层数相等，每组含有多层 Interval 图像。

高斯图像金字塔共 o 组、s 层， 则有：

$$\sigma(s) = \sigma_0 * 2^{s/S}$$

σ表示尺度空间坐标，s 表示 sub-level 层坐标，σ_0表示初始尺度，S 表示每组层数。

(2) 建立 DOG 高斯差分金字塔

为了有效提取稳定的关键点，利用不同尺度的高斯差分核与卷积生成。

DOG 函数：

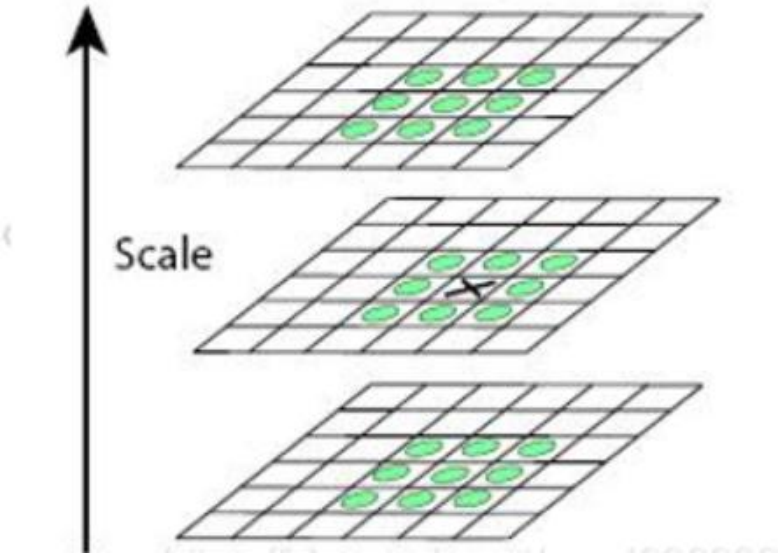
$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y)$$

$$D(x, y, \sigma) = [G(x, y, k\sigma) - G(x, y, \sigma)] * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma)$$

可以通过高斯差分图像看出图像上的像素值变化情况。（如果没有变化，也就没有特征。特征必须是变化尽可能多的点。）DOG 图像描绘的是目标的轮廓。

(3) DOG 局部极值检测

特征点是由 DOG 空间的局部极值点组成的。为了寻找 DOG 函数的极值点，每一个像素点要和它所有的相邻点比较，看其是否比它的图像域和尺度域 的相邻点大或者小。



中间的检测点和它同尺度的 8 个相邻点和上下相邻尺度对应的 9×2 个 点共 26 个点比较，以确保在尺度空间和二维图像空间都检测到极值点。

关键点定位

以上方法检测到的极值点是离散空间的极值点，以下通过拟合三维二次函数来精确确定关键点的位置和尺度，同时去除低对比度的关键点和不稳定的边缘响应点(因为 DOG 算子会产生较强的边缘响应)，以增强匹配稳定性、提高抗噪声能力。

利用已知的离散空间点插值得到的连续空间极值点，为了提高关键点的稳定性，需要对尺度空间 DOG 函数进行曲线拟合。

关键点主方向分配

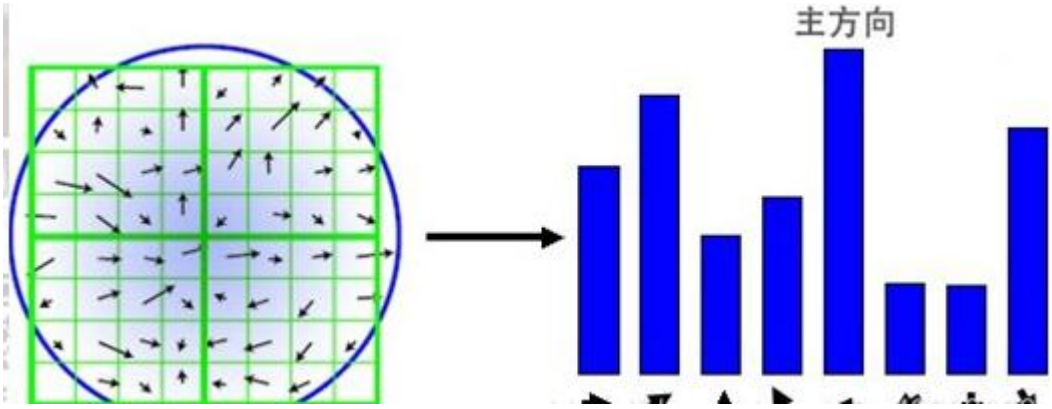
关键点主方向分配就是基于图像局部的梯度方向，分配给每个关键点位置一个或多个方向。所有后面的对图像数据的操作都相对于关键点的方向、尺度和位置进行变换，使得描述符具有旋转不变性。

对于在 DOG 金字塔中检测出的关键点，采集其所在高斯金字塔图像 3σ邻域窗口内像素的梯度和方向分布特征。梯度的模值和方向如下：

梯度幅值：
$$m(x,y)=\sqrt{\left(L(x+1,y)-L(x-1,y)\right)^2+\left(L(x,y+1)-L(x,y-1)\right)^2}$$

梯度方向：
$$\theta(x,y)=\tan^{-1}\left[\frac{L(x,y+1)-L(x,y-1)}{L(x+1,y)-L(x-1,y)}\right]$$

在完成关键点的梯度计算后，使用直方图统计邻域内像素的梯度和方向。梯度直方图将 0~360 度的方向范围分为 36 个柱(bins)，其中每柱 10 度。



方向直方图的峰值代表了该特征点处邻域梯度的方向，以直方图中最大值作为该关键点的主方向。为了增强匹配的鲁棒性，只保留峰值大于主方向峰值 80%的方向作为该关键点的辅方向。

至此，将检测出的含有位置、尺度和方向的关键点即是该图像的 SIFT 特征点。

关键点特征描述

通过以上步骤，对于每一个关键点，拥有三个信息：位置、尺度以及方向。接下来就是为每个关键点建立一个描述符，用一组向量将这个关键点描述出来，使其不随各种变化而改变，比如光照变化、视角变化等。这个描述子不但包括关键点，也包含关键点周围对其有贡献的像素点，并且描述符应该有较高的独特性，以便于提高特征点正确匹配的概率。

SIFT 描述子是关键点邻域高斯图像梯度统计结果的一种表示。通过对关键点周围图像区域分块，计算块内梯度直方图，生成具有独特性的向量，这个向量是该区域图像信息的一种抽象，具有唯一性。

SIFT 算法实践

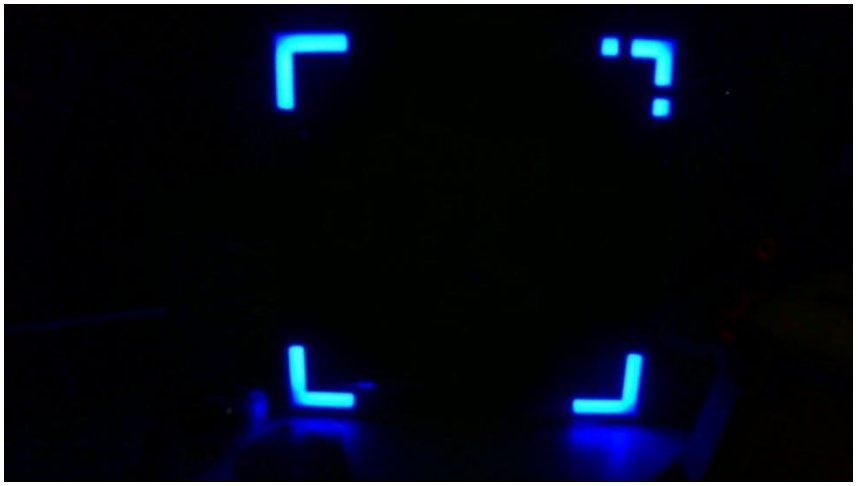
在理解 SIFT 算法原理之后，笔者用 python 进行编程实现该算法，并选取各种类型的图像进行测试，以观察算法效果：（代码见附录）

开发环境：ubuntu18.04

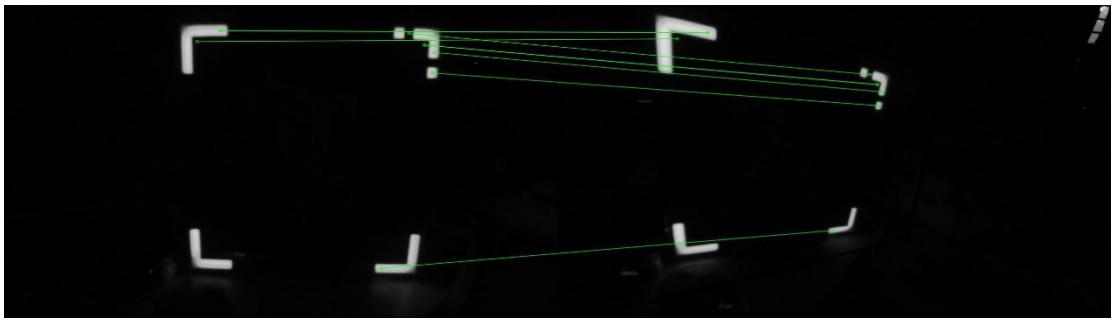
注：采用 opencv-python3.4 版本，过高的版本没有 SIFT 算法相关的函数。

（1）首先测试的一组图像是之前在实验室调低曝光后的免驱相机拍摄的角点发光收集框，模拟的是双目相机看到的效果（从不同角度）。

原图如下:



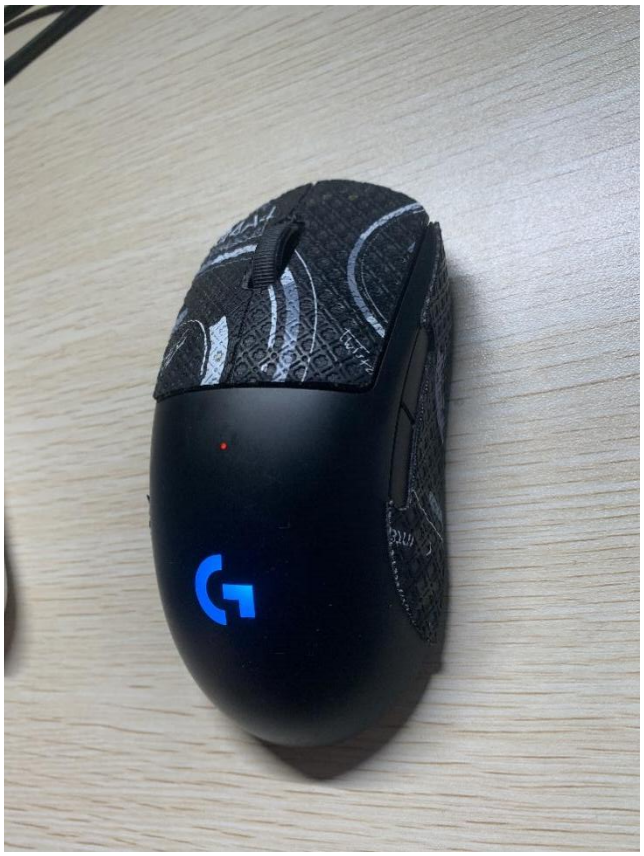
使用编写的程序进行处理得到：



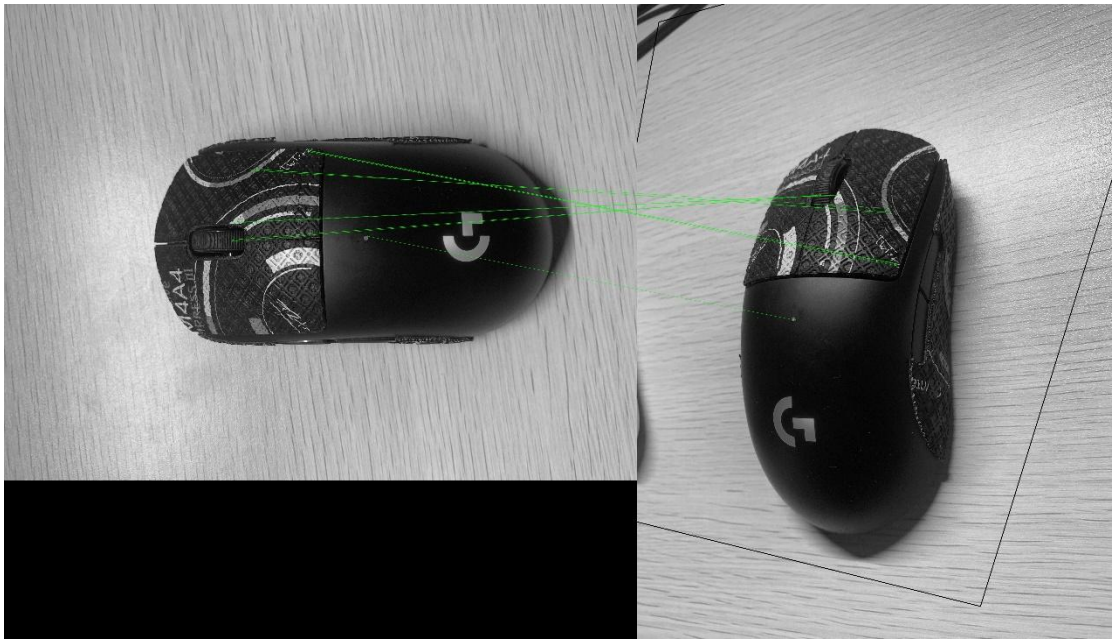
观察输出图片可以发现：通过调整合适的阈值，找到关键点匹配的效果较好。

（2）第二组图片选取一组目标物体呈 90 角且视角偏移光线不同的图像（用手机拍摄自己的鼠标）

原图：



使用编写的程序进行处理得到：



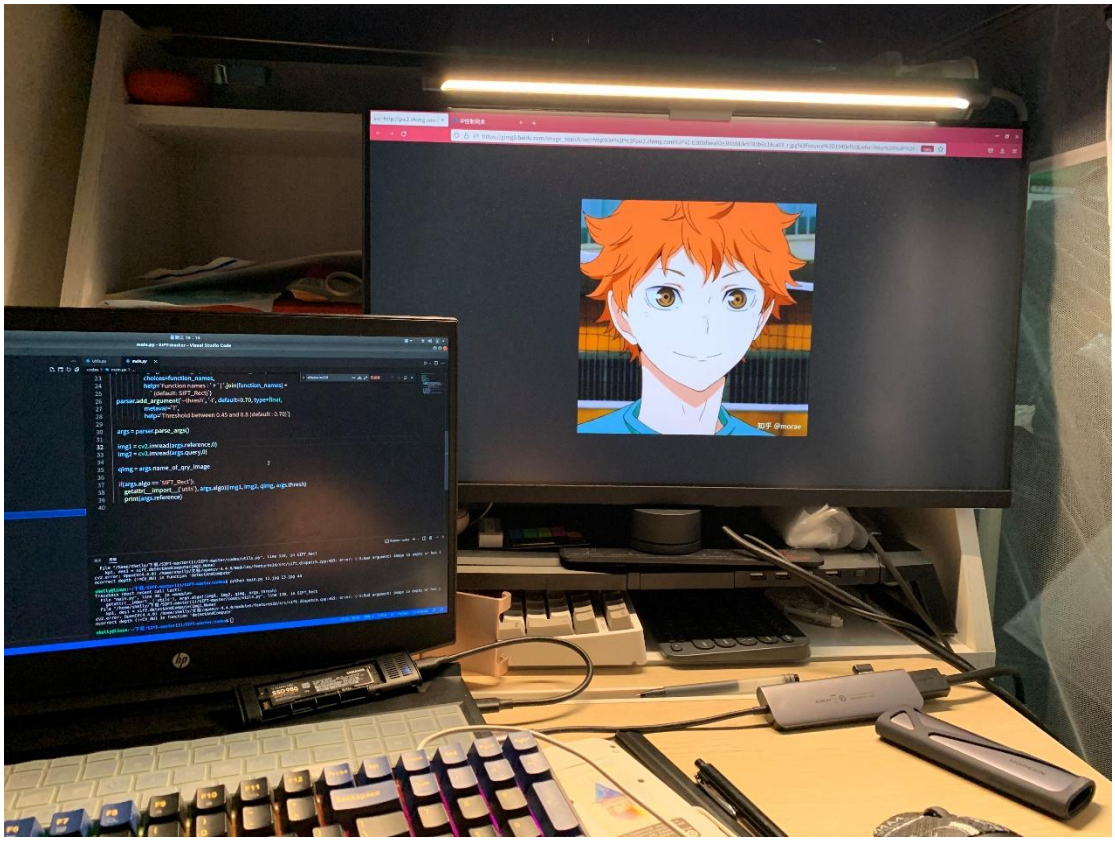
可以观察到：SIFT 算法对于光线，色差，旋转的鲁棒性较好，匹配关键点精度较高。

（3）第三组图像选择的是目标物体特写以及目标物体在复杂环境中的两张图片，也就是检测 SIFT 算法在目标检测中的应用效果：

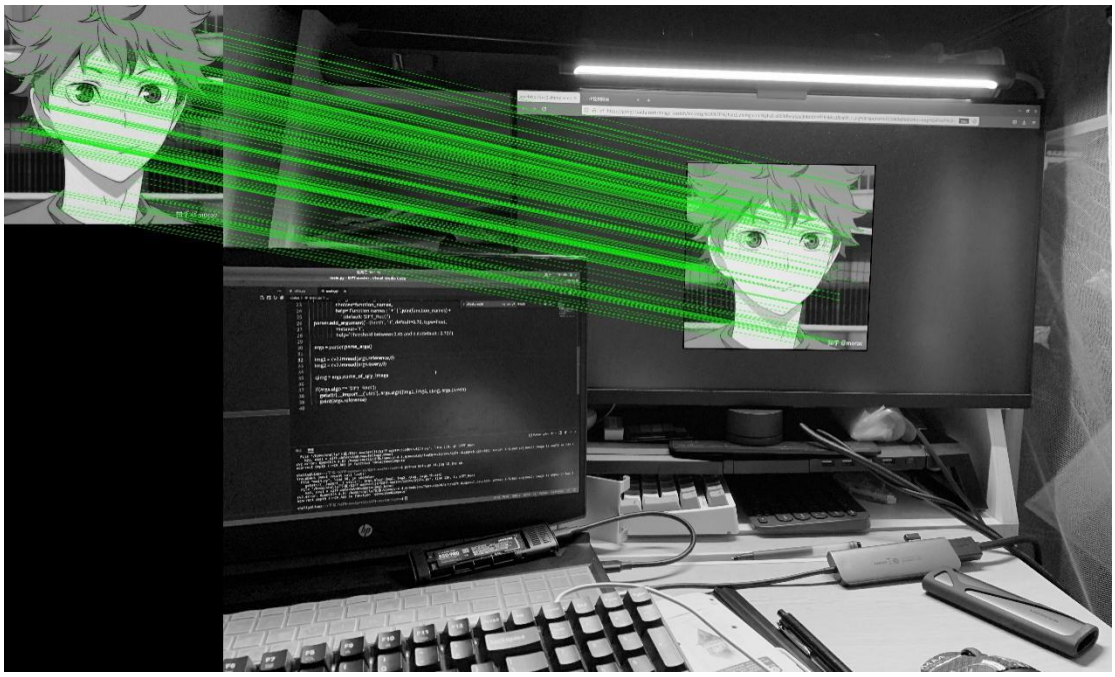
原图：（在网上找了一张我喜欢的动漫角色头像图）

把这个图片在我的屏幕上显示出来并用手机拍照：

（明显看出整个目标物体光线不同且周围的环境比较复杂）



输出图像：



观察结果可以发现：SIFT 算法应用于目标检测的效果也很棒，不易受到外界环境复杂的影响，鲁棒性好。

实践代码重点

（1）基于 BFmatcher 的 SIFT 实现过程：

BFmatcher（Brute-Force Matching）暴力匹配，应用 BFMatcher.knnMatch( )函数来进行核心的匹配，knnMatch（k-nearest neighbor classification）k 近邻分类算法。

kNN 算法则是从训练集中找到和新数据最接近的 k 条记录，然后根据他们的主要分类来决定新数据的类别。该算法涉及 3 个主要因素：训练集、距离或相似的衡量、k 的大小。kNN 算法的核心思想是如果一个样本在特征空间中的 k 个最相邻的样本中的大多数属于某一个类别，则该样本也属于这个类别，并具有这个类别上样本的特性。该方法在确定分类决策上只依据最邻近的一个或者几个样本的类别来决定待分样本所属的类别。

kNN 方法在类别决策时，只与极少量的相邻样本有关。由于 kNN 方法主要靠周围有限的邻近的样本，而不是靠判别类域的方法来确定所属类别的，因此对于类域的交叉或重叠较多的待分样本集来说，kNN 方法较其他方法更为适合。

经检验 BFmatcher 在做匹配时会耗费大量的时间。

（2）findHomography： 计算多个二维点对之间的最优单映射变换矩阵 H（3 行 x3 列） ，使用最小均方误差或者 RANSAC 方法。函数功能：找到两个平面之间的转换矩阵。

M，mask = cv2.findHomography（src_pts，dst_pts，cv2.RANSAC，5.0）

＃第三个参数用于计算单应矩阵的方法。 可以使用以下方法：

0 - 使用所有点的常规方法

CV_RANSAC - 基于 RANSAC 的鲁棒方法

CV_LMEDS - 最少中位数的鲁棒方法

＃第四个参数取值范围在 1 到 10，绝一个点对的阈值。原图像的点经过变换后点与目标图像上对应点的误差＃超过误差就认为是异常值

＃返回值中 M 为变换矩阵 mask 是掩模

（3）RANSAC 算法的输入是一组观测数据（往往含有较大的噪声或无效点），一个用于解释观测数据的参数化模型以及一些可信的参数。RANSAC 通过反复选择数据中的一组随机子集来达成目标。被选取的子集被假设为局内点。

总结:

SIFT 算法对于双目图像匹配问题，目标检测问题有着比较好的实现效果，相比起传统的二值化 findcounters 方法有着更高的鲁棒性，实践证明，算法对于光线，旋转，尺度缩放，视角变化均有较好的不变性。识别检测特征点较多，匹配精度比较高。

当然 SIFT 算法也存在缺点：相比起传统图像处理方法，对于大分辨率图像实时性不高，这个需要优化 sift 匹配算法提高效率。

附录:

main.py

```
#!/usr/bin/env python
"""Script to extract roi using several algorithms
"""
import argparse
import numpy as np
import cv2
import matplotlib.pyplot as plt
from scipy import ndimage, misc
import numpy as np
function_names = ['SIFT_Rect']
if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='ROI Detection Extraction')
    parser.add_argument('reference',metavar='QRY', help='Query Image')
    parser.add_argument('query', metavar='REF', help='Reference Image')
    parser.add_argument('name_of_qry_image', metavar='N_QRY', help='Name of Query
Image')
    parser.add_argument('--algo', '-a', default='SIFT_Rect', metavar='ALGO',
                        choices=function_names,
                        help='Function names : ' + ' | '.join(function_names) +
                            ' (default: SIFT_Rect)')
    parser.add_argument('--thresh', '-t', default=0.70, type=float,
                        metavar='T',
                        help='Threshold between 0.45 and 0.8 (default : 0.70)')
    args = parser.parse_args()
    img1 = cv2.imread(args.reference,0)
    img2 = cv2.imread(args.query,0)
    qimg = args.name_of_qry_image
```

```
    if(args.algo == 'SIFT_Rect'):
        getattr(__import__('utils'), args.algo)(img1, img2, qimg, args.thresh)
        print(args.reference)
```

utils.py

```
#!/usr/bin/env python
import cv2
import matplotlib.pyplot as plt
import numpy as np
from scipy import misc, ndimage
def order_points(pts):
    rect = np.zeros((4, 2), dtype = "float32")
    s = pts.sum(axis = 1)
    rect[0] = pts[np.argmin(s)]
    rect[2] = pts[np.argmax(s)]
    diff = np.diff(pts, axis = 1)
    rect[1] = pts[np.argmin(diff)]
    rect[3] = pts[np.argmax(diff)]

    return rect
def four_point_transform(image, pts):
    rect = order_points(pts)
    (tl, tr, br, bl) = rect
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))

    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
```

```
    maxHeight = max(int(heightA), int(heightB))


    dst = np.array([
        [0, 0],
        [maxWidth - 1, 0],
        [maxWidth - 1, maxHeight - 1],
        [0, maxHeight - 1]], dtype = "float32")


    M = cv2.getPerspectiveTransform(rect, dst)
    warped = cv2.warpPerspective(image, M, (maxWidth, maxHeight))
    return warped


def SIFT_Rect(img1, img2, qimg, thresh):
    image = img2
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1,None)
    kp2, des2 = sift.detectAndCompute(img2,None)
    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1,des2, k=2)
    good = []
    for m,n in matches:
            if m.distance < thresh*n.distance:
                good.append(m)
    if len(good)>5:
        src_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-1,1,2)
        dst_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-1,1,2)
        M, mask = cv2.findHomography(src_pts, dst_pts, cv2.RANSAC,5.0)
        matchesMask = mask.ravel().tolist()
        h,w = img1.shape
```

```
    pts = np.float32([ [0,0],[0,h-1],[w-1,h-1],[w-1,0] ]).reshape(-1,1,2)
    dst = cv2.perspectiveTransform(pts,M)
    img2 = cv2.polylines(img2,[np.int32(dst)],True,[0,255,0],3, cv2.LINE_AA)
draw_params = dict(matchColor = (0,255,0), # draw matches in green color
                   singlePointColor = None,
                   matchesMask = matchesMask, # draw only inliers
                   flags = 2)
pts = np.int32(dst)[:,0]
warped = four_point_transform(image, pts)
res = cv2.drawMatches(img1,kp1,img2,kp2,good,None,**draw_params)
cv2.imwrite('{}_match_result.jpg'.format(qimg), res)
cv2.imwrite('{}_warped.jpg'.format(qimg), warped)
return 0
```