

增强现实

增强现实是目前最热门的趋势之一。增强现实的概念可以定义为现实的提高，真实世界的视图通过叠加计算机生成的元素(例如，图像、视频或3D模型等)得到增强。为了覆盖和整合数字信息，增强现实可以使用不同类型的技术，主要是基于位置和基于识别的方法。

这一章介绍与增强现实相关的主要概念，并编写一些有趣的应用程序，以了解这项技术的潜力。本章从第一个增强现实应用程序开始，学习用OpenCV编写增强现实应用程序的知识。

主要章节如下

增强现实的介绍

Markerless-based增强现实

Marker-based增强现实

Snapchat-based增强现实

QR码检测

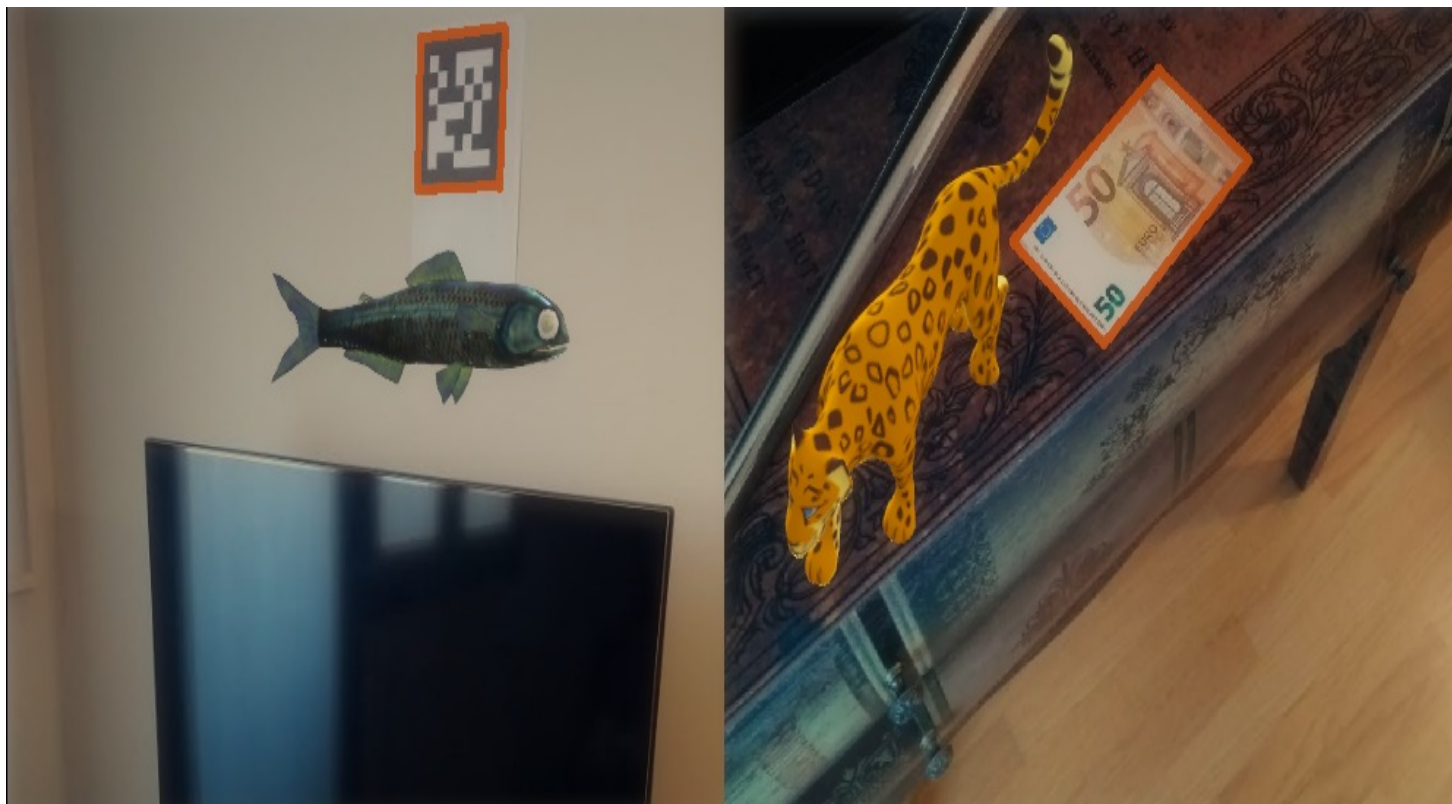
增强现实介绍

基于位置和基于识别的增强现实是增强现实的两种主要类型。这两种类型都需获取用户查找的位置。这些信息是增强现实过程中的关键，且依赖于正确计算相机姿态估计。为了完成这一任务，将这两种类型简要描述如下

- 基于位置的增强现实依赖于读取传感器的数据来检测用户的位置和方向，这些传感器在智能手机设备中非常常见，例如GPS、数字罗盘和加速计，可以获取用户寻找的位置。这些信息用于把计算机生成的元素叠加在屏幕上。
- 另一方面，基于识别的增强现实技术使用图像处理技术取得用户想要寻找的位置。图像中获取摄像机姿态，需要找到环境中已知点与相应的摄像机投影之间的对应关系。为了找到这些对应，文献中有两种主要的方法
 - 基于标记的姿态估计:这种方法依赖于使用平面标记从四个角计算摄像机的姿态(基于正方形标记的方法已经得到了广泛的应用，特别是在增强现实领域)。使用方形标记的一个主要缺点是与相机姿态的计算有关，这依赖于准确确定标记的四个角。在闭塞的情况下，这个任务可能非常困难。然而，一些基于标记检测的方法也可以很好地处理遮挡。这就是ArUco的情况。

- ◦ 基于无标记的姿态估计:当无法使用标记生成姿态估计来准备场景时, 可以使用图像中自然存在的对象进行姿态估计。计算出n个二维点及其对应的三维坐标后, 通过求解透视n点(Perspective-n-Point, PnP)问题来估计相机的姿态。由于这些方法依赖于点匹配技术, 很少能避免输入数据异常值。这就是为什么在姿态估计过程中使用鲁棒技术解决异常值问题, 例如RANSAC随机抽样一致。

在下面截图中, 基于标记和基于无标记的增强现实将与图像处理技术相结合



在截图中, 在左侧可以看到一个基于标记的方法示例, 其中标记用于从四个角计算相机姿态。在右侧可以看到一个基于无标记方法的示例, 其中使用€50记号来计算相机姿态。这两种方法将在以下部分中进行解释。

Marker less-based增强现实

通过对图像进行位姿估计, 可以得到环境中已知点与其投影之间的对应关系。在这一节中, 介绍如何从图像中提取特征来获得摄像机姿态。根据这些特征和它们的匹配, 最终得到相机姿态估计, 再用来覆盖和整合数字信息。

特征提取

特征可以描述为图像中的一个小块，它对图像缩放、旋转和光照尽可能保持不变。通过这种方法，同一场景不同角度的不同图像可检测到相同的特征。因此，一个好的特征应该是这样的

- 同一对象的不同图像提取相同的特征，重复性好，准确性高
- 不同结构的图像不具有此一特征，保持图像的独特之处。

OpenCV提供了许多检测图像特征的算法和技术。包括以下内容

- Harris Corner Detection
- Shi-Tomasi Corner Detection
- Scale Invariant Feature Transform (SIFT)
- Speeded-Up Robust Features (SURF)
- Features from Accelerated Segment Test (FAST)
- Binary Robust Independent Elementary Features (BRIEF)
- Oriented FAST and Rotated BRIEF (ORB)

在`feature_detection.py`中，将使用ORB对图像进行特征检测和描述。该算法来自OpenCV实验室，并在2011年出版的(ORB: a efficient alternative to SIFT or SURF)有说明。ORB基本上是快速关键点检测器和简短描述符的组合，并通过修改密钥来增强性能。第一步是检测keypoints。

ORB检测keypoints

用修改过的FAST-9(半径为9像素的圆圈，存储所检测到的关键点的方向)ORB检测keypoints(默认情况下为500个)。一旦检测到keypoints，下一步是计算描述符，以获得与每个检测到的关键字相关的信息。ORB使用修改过的BRIEF-32描述符来获取每个检测到的关键点的描述。例如，检测到的keypoints如下所示

```
[103  4  111  192  86  239  107  66  141  117  255  138  81  92  62  101  123  148  9
```

因此，第一点是创建ORB探测器

```
orb = cv2.ORB_create()
```

下一步是检测加载图像中的keypoints

```
keypoints = orb.detect(image, None)
```

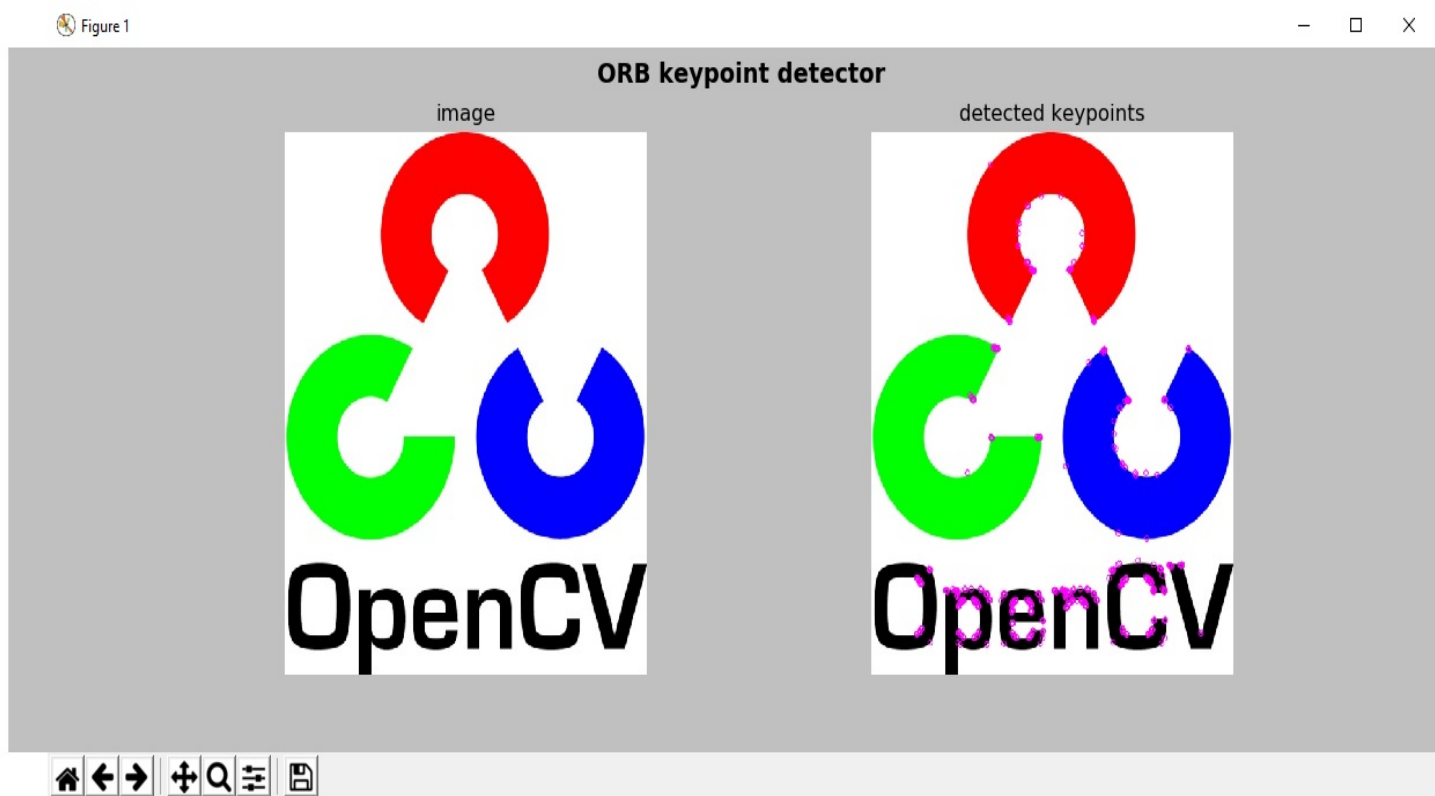
一旦检测到keypoints，下一步是计算检测到的keypoints的描述符

```
keypoints, descriptors = orb.compute(image, keypoints)
```

注意，可执行`orb.detectAndCompute(image, None)`来检测keypoints并计算所检测keypoints的描述符。最后，我们可以使用`cv2.drawKeypoints()`函数来绘制检测到的keypoints

```
image_keypoints = cv2.drawKeypoints(image, keypoints, None, color=(255, 0, 255), flag
```

在下面截图中可以看到这个脚本的输出



可以看到，右边的结果显示了检测到的ORB关键点，这些关键点已经被ORB关键点检测器检测到了。

特征匹配

下一个示例，为如何匹配检测到的特性。OpenCV提供了两个匹配器，如下所示

- **Brute-Force (BF) matcher**: 匹配器接受第一个集合中检测到的特征计算的每个描述符，并与第二个集合中的所有其他描述符进行匹配。最后，返回最近距离的匹配。
- **Fast Library for Approximate Nearest Neighbors (FLANN)** : 匹配器工作速度快于BF匹配器的大型数据集，包含了最近邻搜索的优化算法。

在`feature_match.py`脚本中，用BF matcher 查看如何匹配检测到的特性。因此，第一步是检测关键点并计算描述符

```
orb = cv2.ORB_create()
keypoints_1, descriptors_1 = orb.detectAndCompute(image_query, None)
keypoints_2, descriptors_2 = orb.detectAndCompute(image_scene, None)
```

下一步是使用`cv2.BFMatcher()` 创建BF matcher对象。

```
bf_matcher = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

第一个参数`normType`将距离测量设置为使用`cv2.NORM_L2`。如果使用ORB描述符(或其他基于二进制的描述符，如BRIEF或BRISK)，则使用的距离度量为`cv2.NORM_HAMMING`。第二个参数`crossCheck`(默认为`False`)可以设置为`True`，以便在匹配过程中只返回一致性匹配对，即两个集合中的两个特性应该彼此匹配。创建之后，下一步是使用`BFMatcher.match()` 方法匹配检测到的描述符

```
bf_matches = bf_matcher.match(descriptors_1, descriptors_2)
```

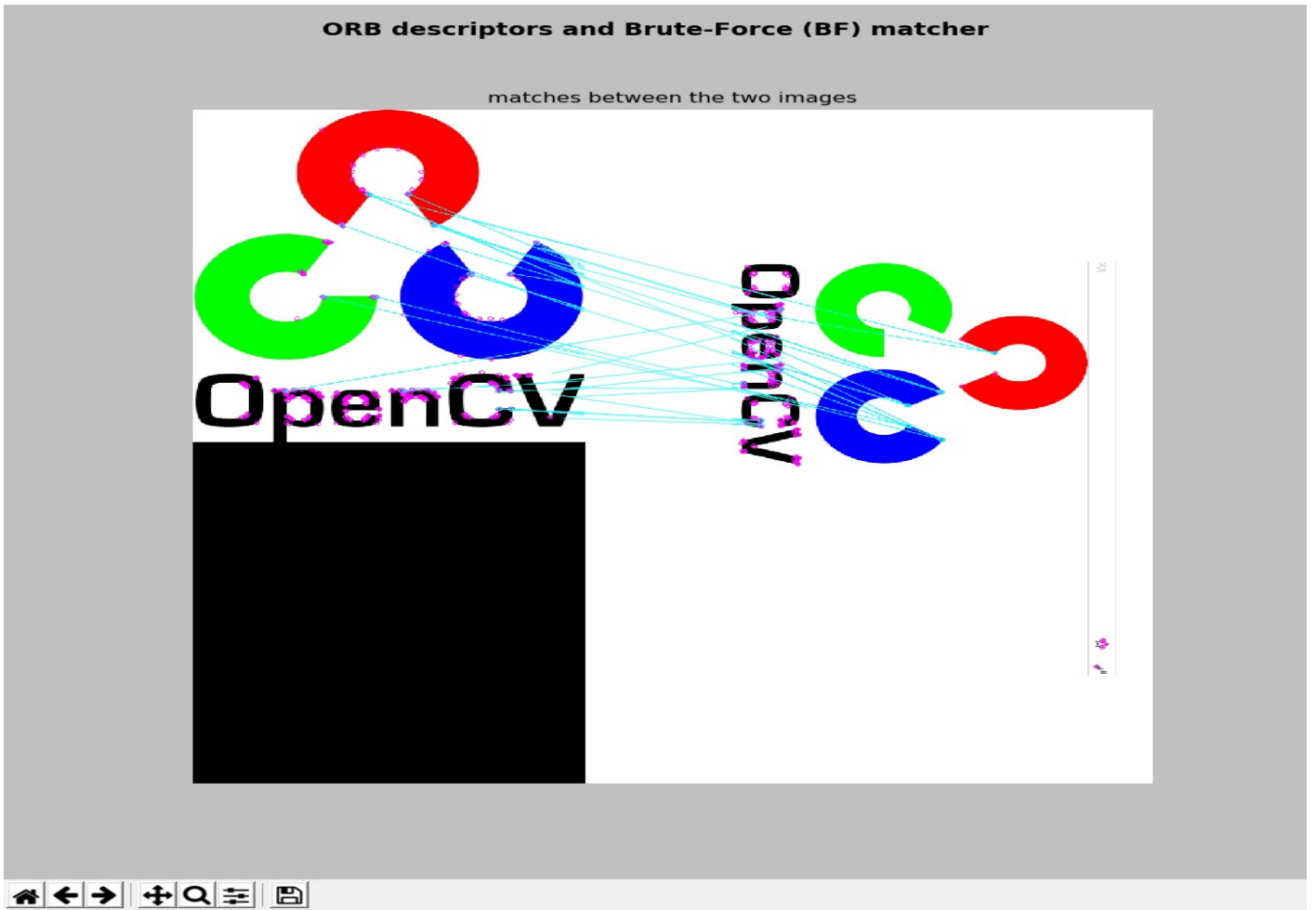
前面计算的描述符是`descriptors_1`和`descriptors_2`;通过这种方式，我们可以在两个图像中获得最佳匹配。此时，可按距离的升序对匹配进行排序

```
bf_matches = sorted(bf_matches, key=lambda x: x.distance)
```

最后，我们可用`cv2.drawMatches()`函数来绘制匹配项。在本例中，仅显示前20个匹配

```
result = cv2.drawMatches(image_query, keypoints_1, image_scene, keypoints_2, bf
```

`cv2.drawMatches()`函数的作用是:将两个图像水平连接起来，并从第一个图像绘制到第二个图像。在下面截图中可以看到`feature_match.py`脚本的输出



如截图，绘制了两个图像 (image_query、image_scene) 之间的匹配。

通过特征匹配和单应性计算寻找目标

为了完成本节，看寻找目标的最后一步。一旦特征被匹配，下一步是使用 `cv2.findHomography()` 函数查找两个图像中匹配keypoints的位置之间的透视图转换。

OpenCV提供了几种计算单应性矩阵RANSAC、最小中位数(LMEDS)和PROSAC (RHO)的方法。在本例中，用RANSAC，如下所示

```
M, mask = cv2.findHomography(pts_src, pts_dst, cv2.RANSAC, 5.0)
```

这里，`pts_src`是源图像中匹配关键点的位置，`pts_dst`是查询图像中匹配keypoints的位置

第四个参数`ransacjthreshold`设置再投影误差，将点对视为内围层。在这种情况下，如果重投影误差大于5.0，则认为对应的点对是离群值。此函数计算并返回由关键点位置定义的源和目标平面之间的透视图转换矩阵M。

最后，基于透视变换矩阵 M ，计算查询图像中对象的四个角。为了做到这一点，根据原始图像的形状计算它的四个角，并使用`cv2.perspectiveTransform()`函数转换它们以获得目标角

```
pts_corners_dst = cv2.perspectiveTransform(pts_corners_src, M)
```

其中，`pts_corners_src`包含原图像的四个角， M 为透视变换矩阵；`pts_corners_dst`输出包含了查询图像中对象的四个角。可用`cv2.polylines()`函数来绘制被检测对象的轮廓

```
img_obj = cv2.polylines(image_scene, [np.int32(pts_corners_dst)], True, (0, 255
```

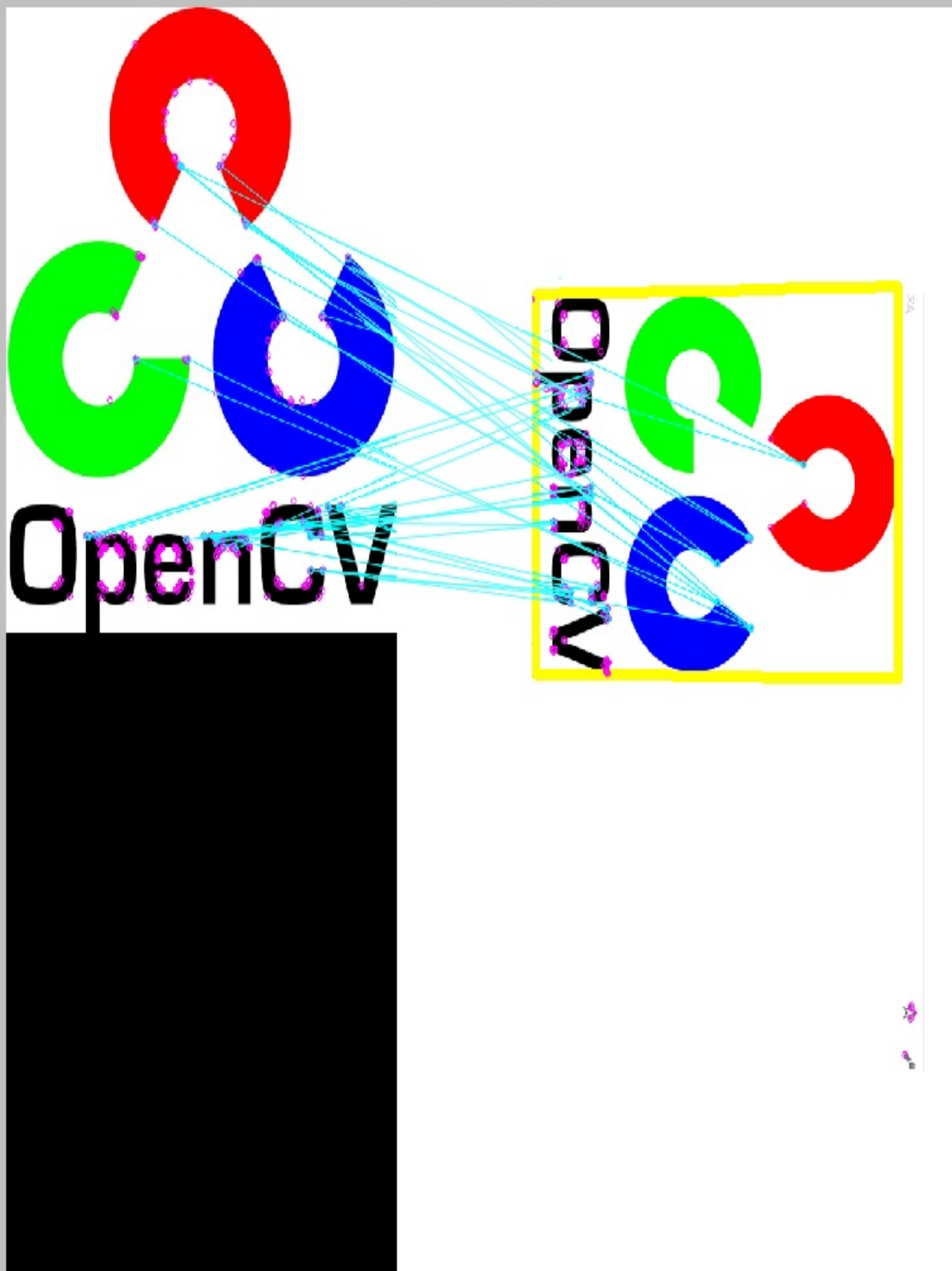
最后，还可用`cv2.drawMatches()`函数绘制匹配项，如下所示

```
img_matching = cv2.drawMatches(image_query, keypoints_1, img_obj, keypoints_2, b
```

在下面截图中可以看到`feature_matching_object_recognition.py`脚本的输出

Feature matching & homography computation

feature matching



在截图中，可看到特征匹配和单应性计算，这是物体识别的两个关键步骤。

Marker-based增强现实

在本节中，将看到基于标记的增强现实是如何工作的。可使用库、算法或包来生成和检测标记。从这个意义上说，在检测标记方面提供最先进性能的是ArUco。

ArUco自动检测标记并纠正可能的误差。此外，ArUco提出了一种解决遮挡问题的方法，将多个标记与遮挡掩码相结合，通过颜色分割来计算遮挡掩码。

如前所述，姿态估计是增强现实应用中的一个关键过程。可以根据标记进行姿态估计。使用标记的主要优点是既能有效地检测出图像中标记的四个角，又能准确地提取出标记的四个角。最后，通过之前计算出的标记的四个角可以得到相机的姿态。下一小节将看到如何创建基于标记的增强现实应用程序，从创建标记和字典开始。

创建标记和字典

使用ArUco的第一步是创建标记和字典。首先，ArUco标记是由外部和内部单元(也称为位)组成的正方形标记。外部细胞被设置为黑色，创建一个可以快速和可靠地检测到的外部边界。其余的单元(内部单元)用于编码标记。ArUco标记也可以创建不同的大小。标记的大小表示与内部基质相关的内部细胞的数量。例如，一个大小为5 x 5 (n=5)的标记由25个内部单元格组成。还可在标记边框中设置位的数量。

其次，标记字典是一组被认为在特定应用程序中使用的标记。虽然以前的库只考虑固定的字典，但ArUco提出了一种自动生成标记的方法，这些标记的数量和所需的位数都是指定的。从这个意义上说，ArUco包含了一些预定义的字典，涵盖了与标记的数量和标记大小相关的许多配置。

创建基于标记的增强现实应用程序时要考虑的第一步是打印要使用的标记。

在`aruco_create_marks.py`脚本中，我们创建了一些可以打印的标记。第一步是创建`dictionary`对象。ArUco有一些预定义的字典

```
DICT_4X4_50 = 0, DICT_4X4_100 = 1, DICT_4X4_250 = 2, DICT_4X4_1000 = 3, DICT_5X5_50 =  
DICT_5X5_1000 = 7, DICT_6X6_50 = 8, DICT_6X6_100 = 9, DICT_6X6_250 = 10, DICT_6X6_100
```

在本例中，用`cv2.aruco.Dictionary_get()`函数创建一个字典，组成250个标记。每个标记的大小为7 x 7 (n=7)

```
aruco_dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_7X7_250)
```

此时，可以使用`cv2.aruco.drawMarker()`函数来绘制标记，该函数返回准备打印的标记。`drawmarker()`的第一个参数是`dictionary`对象。第二个参数是标记`id`，其范围在0到249之间，因为我们的字典有250个标记。第三个参数是`sidePixels`，它是创建的标记图像的大小（以像素为单位）。第四个参数（默认为1，可选）是`borderBits`，它设置标记边框中的位数。

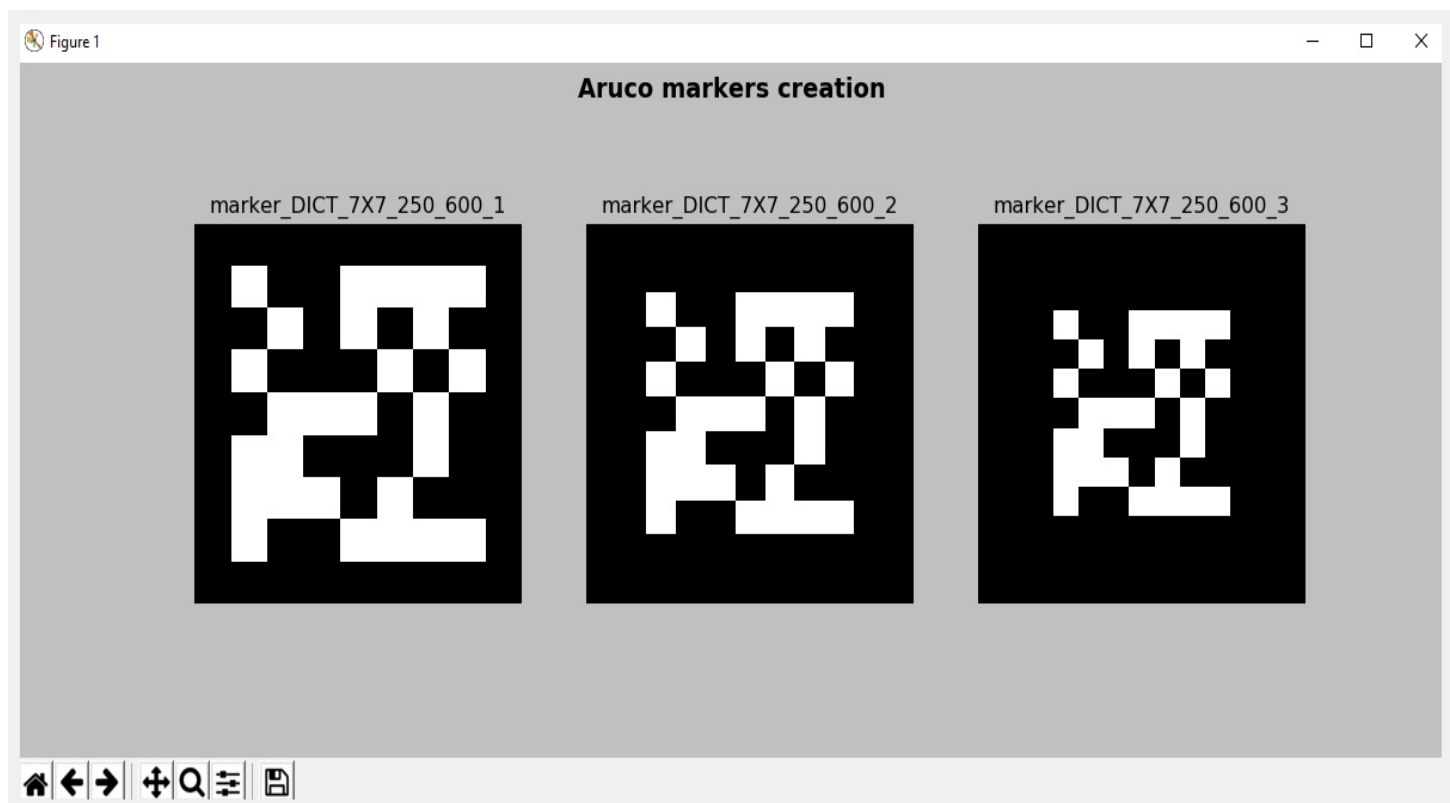
因此，在本例中，将创建三个标记来改变标记边框中的位的数量

```
aruco_marker_1 = cv2.aruco.drawMarker(dictionary=aruco_dictionary, id=2, sidePixels=60  
aruco_marker_2 = cv2.aruco.drawMarker(dictionary=aruco_dictionary, id=2, sidePixels=6  
aruco_marker_3 = cv2.aruco.drawMarker(dictionary=aruco_dictionary, id=2, sidePixe
```

这些标记映像可以保存在磁盘上(使用`cv2.imwrite()`)

```
cv2.imwrite("marker_DICT_7X7_250_600_1.png", aruco_marker_1)  
cv2.imwrite("marker_DICT_7X7_250_600_2.png", aruco_marker_2)  
cv2.imwrite("marker_DICT_7X7_250_600_3.png", aruco_marker_3)
```

在`aruco_create_marks.py`脚本中，显示了创建的标记。在下面截图中可以看到输出



在截图中，显示了创建的三个标记

检测标记

可以使用`cv2.aruco.detectmarker()`函数来检测图像中的标记

```
corners, ids, rejected_corners = cv2.aruco.detectMarkers(gray_frame, aruco_dicti
```

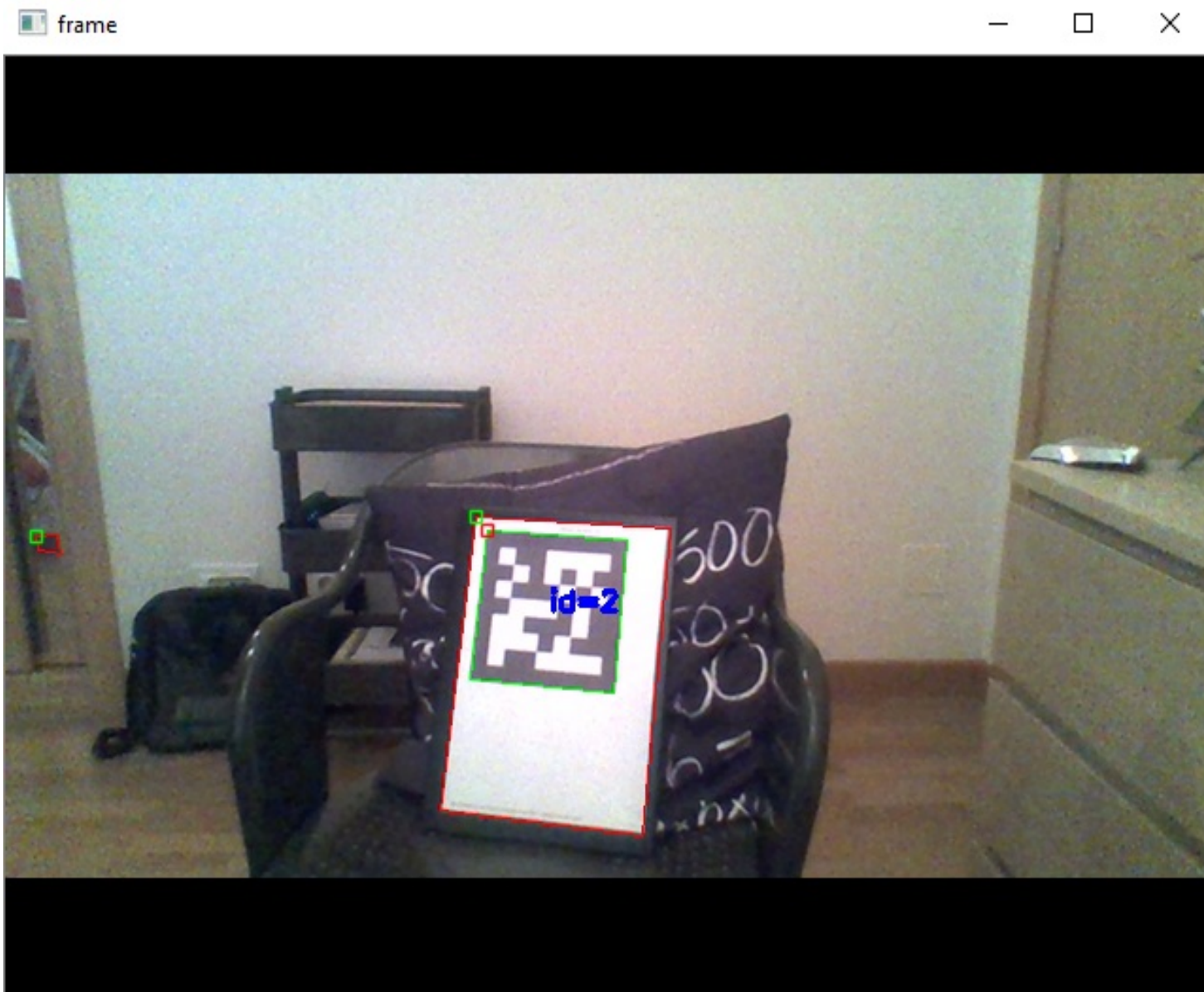
`cv2.aruco.detectmarker()`的第一个参数是要检测标记的灰度图像。第二个参数是字典对象，它应该在前面创建。第三个参数建立了所有可以在检测过程中定制的参数。此函数返回以下信息

- 返回检测到的标记的角的列表。对于每个标记，返回它的四个角(左上角、右上角、右下角和左下角)。
- 返回检测到的标记的标识符列表。
- 被拒绝的候选人名单被返回，它是由已经找到的所有方格组成的，但是它们没有适当的编码。这个被拒绝的候选列表对于调试非常有用。每一个被拒绝的候选人都是由四个角落组成的。

`aruco_detect_markers.py`脚本检测来自摄像头的标记。首先使用前面提到的`cv2.aruco.detectMarkers()`函数检测标记，然后使用`cv2.aruco.drawDetectedMarkers()`函数分别绘制检测到的标记和被拒绝的候选标记，具体如下：

```
# Draw detected markers:
frame = cv2.aruco.drawDetectedMarkers(image=frame, corners=corners, ids=ids, borderCo
# Draw rejected markers:
frame = cv2.aruco.drawDetectedMarkers(image=frame, corners=rejected_corners, borderCo
```

如果执行`aruco_detect_mark.py`脚本，检测到的标记将用绿色边框绘制，而被拒绝的候选标记将用红色边框绘制，如下面的截图所示



在截图中，可以看到检测到一个标记(id=2)，是用绿色边框绘制的。还可看到两个带有红色边框的被拒绝的候选者。

相机校正

在利用检测到的标记获得摄像机姿态之前，需要知道摄像机的标定参数。从这个意义上说，ArUco提供了执行这项任务所需的校准程序。注意，校准过程只执行一次，因为相机光学没有修改。校准过程中使用的主要功能是`cv2.aruco.calibrateCameraCharuco()`。

上述函数校准相机使用一组角落从几个视图提取的一块板。当校准过程完成后，该函数返回摄像机矩阵(一个 3×3 的浮点摄像机矩阵)和一个包含畸变系数的向量。更具体地说， 3×3 矩阵编码焦距和相机中心坐标(也称为内在参数)。畸变系数为摄像机产生的畸变的模型，函数的签名如下

```
calibrateCameraCharuco(charucoCorners, charucoIds, board, imageSize, cameraMatrix, di
```

这里，`charucoCorners`是一个包含检测到的charuco角的向量，`charucoId`是标识符列表，`board`表示board布局，而`imageSize`是输入图像的大小。输出向量`rvecs`包含每个板视图估计的旋转向量的向量组，而`tvecs`是每个模式视图估计的平移向量的向量组。如前所述，也返回摄像机矩阵、`cameraMatrix`和失真系数`distCoeffs`。

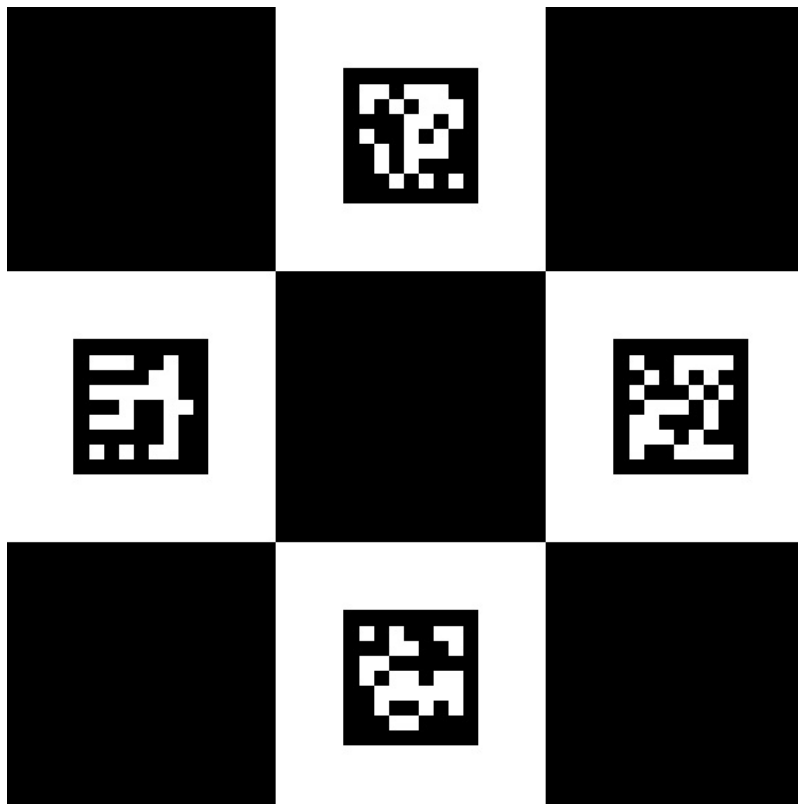
该板是使用`cv2.aruco.CharucoBoard_create()`函数创建的, 签名如下

```
CharucoBoard_create(squaresX, squaresY, squareLength, markerLength, dictionary) -> re
```

`squareX`是方块在x方向上的数值, `squaresY`是方块在y方向上的数值, `squareLength`是棋盘方块长度(通常为米), `markerLength`是标记边长(`squareLength`单位一样), 和字典集第一标记字典使用为了创建内部的标记。例如, 为创建一个板, 可用以下代码行

```
dictionary = cv2.aruco.Dictionary_get(cv2.aruco.DICT_7X7_250)
board = cv2.aruco.CharucoBoard_create(3, 3, .025, .0125, dictionary)
img = board.draw((200 * 3, 200 * 3))
```

在下面截图中可以看到创建的板



该板稍后将在校准过程中由`cv2.aruco.calibrateCameraCharuco()`函数调用


```
cal = cv2.aruco.calibrateCameraCharuco(all_corners, all_ids, board, image_size, None,
```

一旦校准过程完成，将相机矩阵和失真系数保存到磁盘。为此，用pickle，它可用于序列化和反序列化Python对象结构。

校准程序完成后，可进行相机姿态估计。

脚本`aruco_camera_calibration.py`执行校准过程。请注意，此脚本可利用先前创建的和打印的板，创板和执行校准过程

摄像机位置估计

为了估计摄像机的姿态，可以使用`cv2.aruco.estimatePoseSingleMarkers()`函数，它估计单个标记的姿态。姿态由旋转和平移向量组成。签名如下

```
cv.aruco.estimatePoseSingleMarkers( corners, markerLength, cameraMatrix, distCoeffs[,
```

其中，`cameraMatrix`和`distCoeffs`分别为摄像机矩阵和畸变系数，提供校准过程后获得的值。角参数是一个包含每个检测到的标记的四个角的向量。标记长度参数是标记端长度。注意，返回的平移向量将在相同的单元中。该函数为每个检测到的标记返回`rvecs`（旋转向量）、`tvecs`（平移向量）和`_objPoints`（所有检测到的标记角的对象点的数组）。

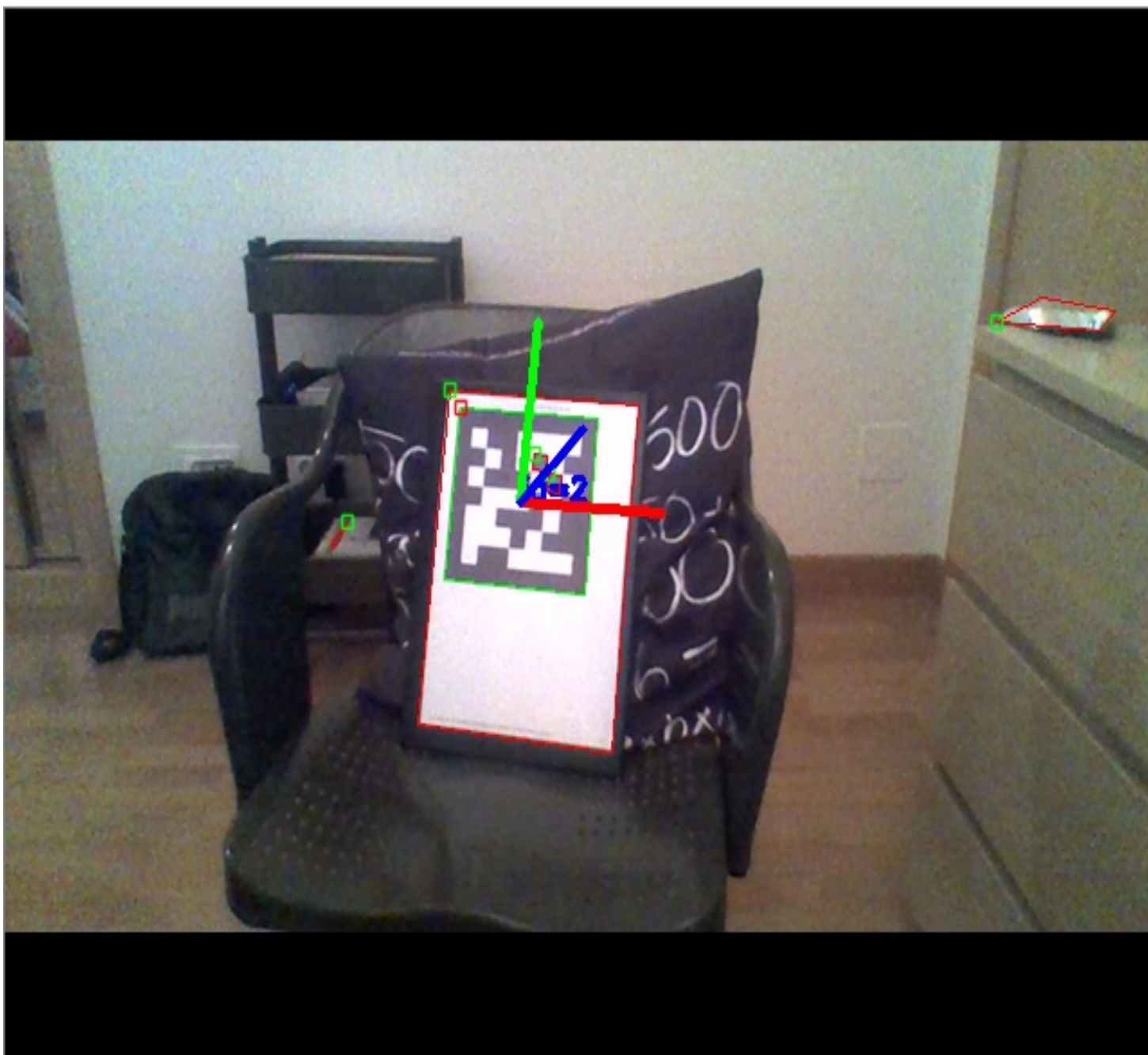
标志坐标系以标志中间为中心。因此，该标记的四个角（在自己的坐标系中）的坐标为

- $(-markerLength/2, markerLength/2, 0)$
- $(markerLength/2, markerLength/2, 0)$
- $(markerLength/2, -markerLength/2, 0)$
- $(-markerLength/2, -markerLength/2, 0)$

最后，ArUco还提供了`cv.aruco.drawaxis()`函数，该函数可用于为每个检测到的标记绘制系统轴，签名如下

```
cv.aruco.drawAxis( image, cameraMatrix, distCoeffs, rvec, tvec, length ) -> image
```

在前面的函数中已经介绍了所有的参数，除了长度参数，长度参数设置绘制轴的长度（与`tvec`在同一单元中）。在下面截图中可以看到脚本`aruco_detect_markers.py`的输出



在截图中，可以看到只检测到一个标记，并且还绘制了该标记的系统轴。

相机姿态估计和基本增强

在这一点上，可覆盖一些图像，形状，或3D模型，以看到一个完整的增强现实应用程序。在第一个示例中，将用标记的大小覆盖一个矩形。执行此功能的代码如下

```

if ids is not None:

# rvecs and tvecs are the rotation and translation vectors respectively
rvecs, tvecs, _ = cv2.aruco.estimatePoseSingleMarkers(corners, 1, cameraMatrix, d

for rvec, tvec in zip(rvecs, tvecs):
    # Define the points where you want the image to be overlaid (remember: marker coo
    desired_points = np.float32([[-1 / 2, 1 / 2, 0], [1 / 2, 1 / 2, 0], [1 / 2, -1 /

    # Project the points:
    projected_desired_points, jac = cv2.projectPoints(desired_points, rvecs, tve

    # Draw the projected points:
    draw_points(frame, projected_desired_points)

```

第一步是定义要覆盖图像或模型的点。当我们想要在检测到的标记上覆盖矩形时，这些坐标是 $-1/2, 1/2, 0$, $1/2, 1/2, 0$, $1/2, -1/2, 0$, $-1/2, -1/2, 0$ 。

记住，须在标记坐标系中定义这些坐标。下一步是使用 `cv2.projectPoints()` 函数投射这些点

```

projected_desired_points, jac = cv2.projectPoints(desired_points, rvecs, tvecs,

```

最后，用 `draw_points()` 函数来绘制这些点

```

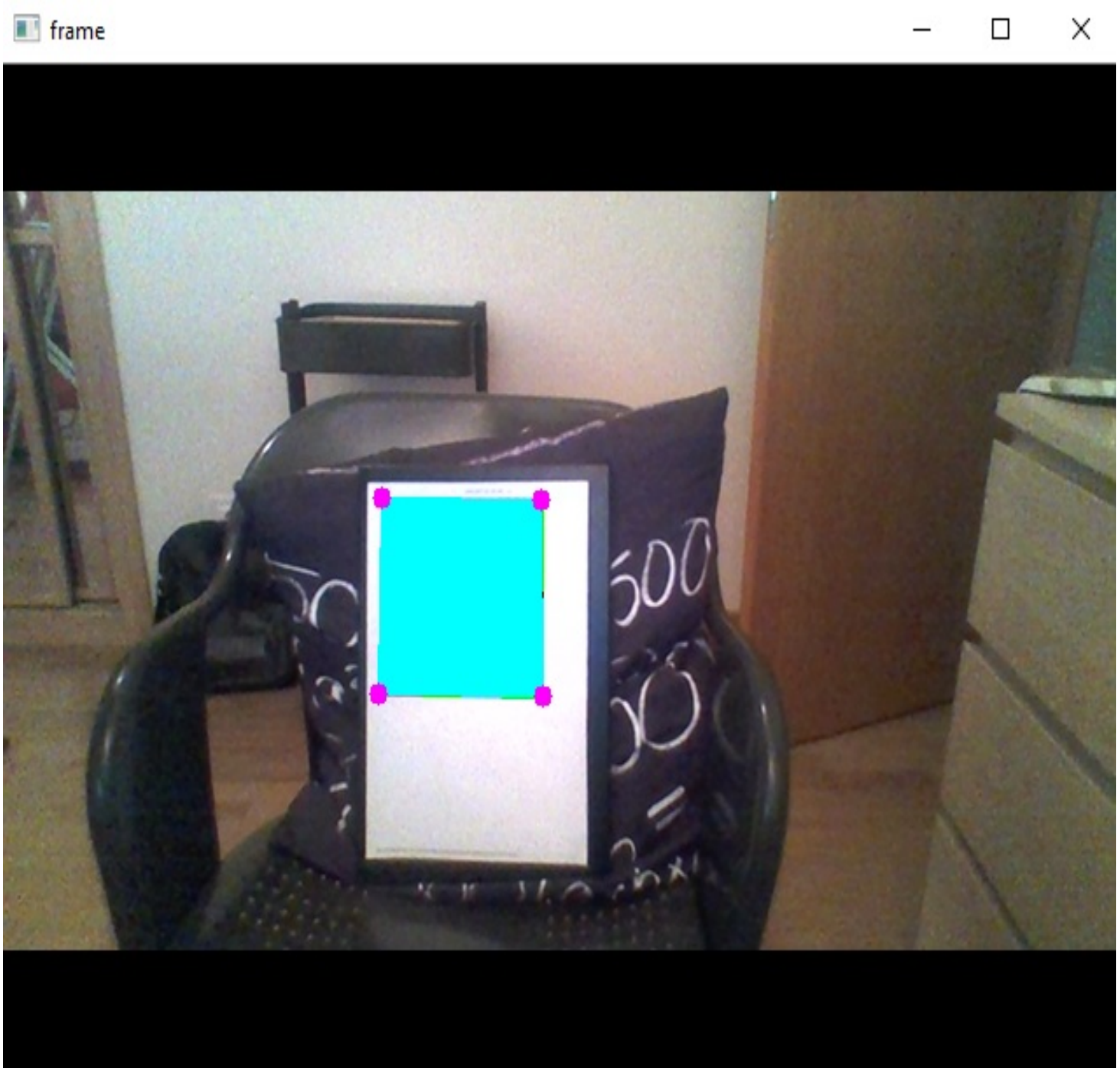
def draw_points(img, pts):
    """ Draw the points in the image"""

    pts = np.int32(pts).reshape(-1, 2)
    img = cv2.drawContours(img, [pts], -1, (255, 255, 0), -3)
    for p in pts:
        cv2.circle(img, (p[0], p[1]), 5, (255, 0, 255), -1)

    return img

```

`aruco_detect_markers_draw_square.py` script 输出见截屏



在截图中，可以看到一个青色矩形覆盖在检测到的标记上。此外，可看到矩形的四个角，它们是用洋红色画的。

相机姿态估计和更先进的增强

修改`aruco_detect_markers_draw_square.py`脚本，以叠加覆盖高级的增强。

在本例中，将覆盖树的图像，在下面截图中可以看到



为了执行这个增强，编写了`draw_augmented_overlay()`函数，如下所示


```

def draw_augmented_overlay(pts_1, overlay_image, image):
    """Overlay the image 'overlay_image' onto the image 'image'"""

    # Define the squares of the overlay_image image to be drawn:
    pts_2 = np.float32([[0, 0], [overlay_image.shape[1], 0], [overlay_image.shape[1],
    overlay_image.shape[0], 0], [overlay_image.shape[1], overlay_image.shape[0]]])

    # Draw border to see the limits of the image:
    cv2.rectangle(overlay_image, (0, 0), (overlay_image.shape[1], overlay_image.shape[0]),
    cv2.BORDER_THICK, cv2.cvtColor(np.zeros([1, 1, 3]), cv2.COLOR_GRAY2BGR), 2)

    # Create the transformation matrix:
    M = cv2.getPerspectiveTransform(pts_2, pts_1)

    # Transform the overlay_image image using the transformation matrix M:
    dst_image = cv2.warpPerspective(overlay_image, M, (image.shape[1], image.shape[0]),
    flags=cv2.INTER_LINEAR)

    # cv2.imshow("dst_image", dst_image)

    # Create the mask:
    dst_image_gray = cv2.cvtColor(dst_image, cv2.COLOR_BGR2GRAY)
    ret, mask = cv2.threshold(dst_image_gray, 0, 255, cv2.THRESH_BINARY_INV)

    # Compute bitwise conjunction using the calculated mask:
    image_masked = cv2.bitwise_and(image, image, mask=mask)
    # cv2.imshow("image_masked", image_masked)

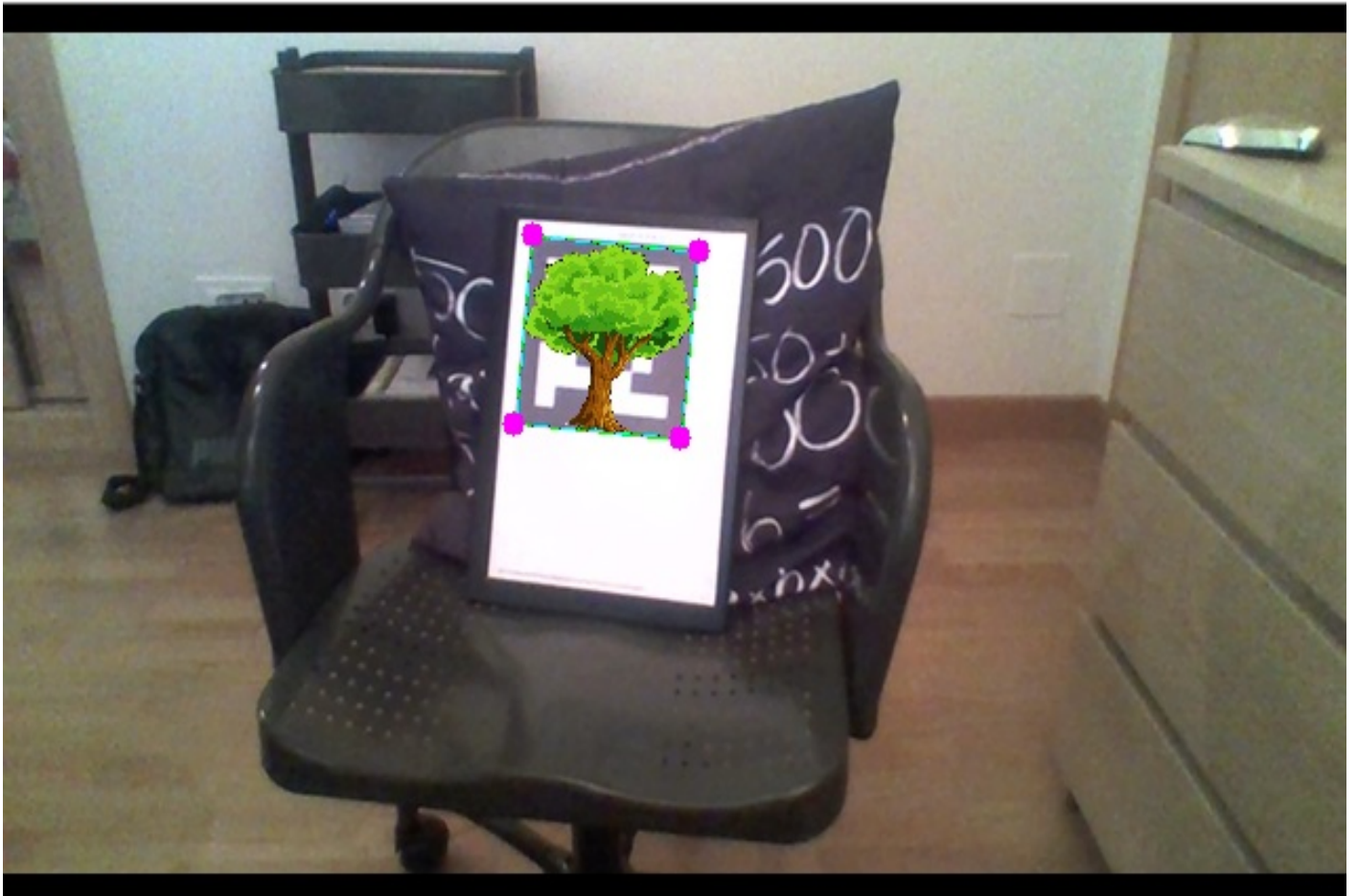
    # Add the two images to create the resulting image:

    result = cv2.add(dst_image, image_masked)
    return result

```

`draw_augmented_overlay()` 函数的作用是:首先定义叠加图像的正方形。然后计算变换矩阵,利用变换矩阵对叠加图像进行变换,得到`dst_image`图像。接下来创建`mask`并使用前面创建的`mask`来计算按位操作,以获得`image_mask`图像。最后一步是执行`dst_image`和`image_mask`之间的加法,以获得最终返回的结果图像。

在下面截图中可以看到`aruco_detect_markers_augmented_reality.py`脚本的输出



为了覆盖更复杂和先进的3D模型，可以使用OpenGL。开放图形库(OpenGL)是一个用于渲染2D和3D模型的跨平台API。

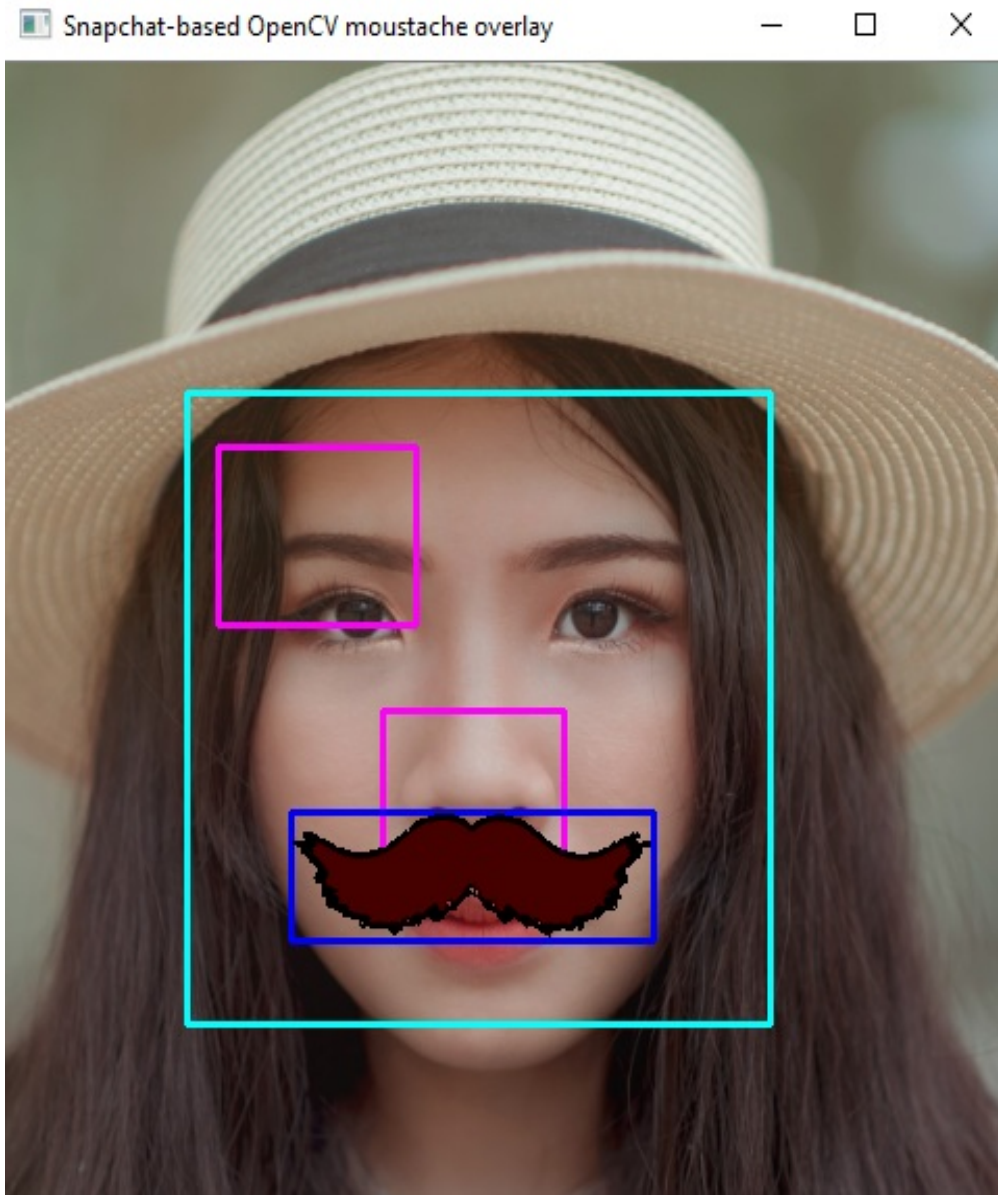
，PyOpenGL (<http://pyopengl.sourceforge.net/>)是最常见的跨平台Python OpenGL。

Snapchat-based增强现实

在本节中，将看到如何创建一些有趣的基于快照的过滤器。在本例中，将创建两个过滤器。第一个在鼻子和嘴巴之间覆盖了一大片胡子。第二张在被检测到的脸上覆盖了一副眼镜。在下面的小节中，您将看到如何实现此功能

基于snapchat的增强现实OpenCV胡子覆盖

在snapchat_augmented_reality_moustache.py脚本中，将胡子覆盖在检测到的脸上。图像不断地从网络摄像头捕捉。我们还包括了使用测试图像的可能性，而不是从网络摄像头捕获的图像。这对于调试算法很有用。在解释这个脚本的关键步骤之前，看下面截图，为测试图像算法的输出



第一步是检测图像中的所有面孔。可以看到，青色矩形表示图像中检测到的人脸的位置和大小。算法的下一步是遍历图像中所有检测到的人脸，在其区域内搜索鼻子。洋红色的矩形表示图像中检测到的鼻子。一旦检测到鼻子，下一步是调整想要覆盖胡子的区域，这是根据之前计算的鼻子的位置和大小来计算的。在本例中，蓝色矩形表示胡子将被覆盖的位置。你也可以看到图像中有两个被检测到的鼻子，只有一个胡子覆盖着。这是因为执行基本检查是为了知道检测到的鼻子是否有效。一旦检测到一个有效的鼻子，胡子就会被覆盖，如果继续对检测到的脸进行迭代，或者分析另一个帧。

因此，在这个脚本中，同时检测脸部和鼻子。为了检测这些对象，创建了两个分类器，一个用于检测人脸，另一个用于检测鼻子。要创建这些分类器，需要以下代码

```
face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
nose_cascade = cv2.CascadeClassifier("haarcascade_mcs_nose.xml")
```

一旦创建了分类器，下一步就是在图像中检测这些对象。在这种情况下，使用了 `cv2.detectMultiScale()` 函数。该函数在输入的灰度图像中检测不同大小的对象，并以矩形列表的形式返回检测到的对象。例如，为了检测人脸，可以使用以下代码

```
faces = face_cascade.detectMultiScale(gray, 1.3, 5)
```

此时，遍历检测到的人脸，尝试检测鼻子

```
# Iterate over each detected face:
for (x, y, w, h) in faces:
    # Draw a rectangle to see the detected face (debugging purposes):
    # cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 255, 0), 2)

    # Create the ROIS based on the size of the detected face: roi_gray = gray[y:y + h
    roi_color = frame[y:y + h, x:x + w]

    # Detects a nose inside the detected face:
    noses = nose_cascade.detectMultiScale(roi_gray)
```

一旦检测到鼻子，遍历所有检测到的鼻子，并计算胡子将覆盖的区域。一个基本的检查是为了过滤掉假鼻子的位置。在成功的情况下，胡子将覆盖在图像的基础上先前计算的区域

```

for (nx, ny, nw, nh) in noses:
    # Draw a rectangle to see the detected nose (debugging purposes):
    # cv2.rectangle(roi_color, (nx, ny), (nx + nw, ny + nh), (255, 0, 255)

    # Calculate the coordinates where the moustache will be placed: x1 = int(nx - nw
    x2 = int(nx + nw / 2 + nw)
    y1 = int(ny + nh / 2 + nh / 8)
    y2 = int(ny + nh + nh / 4 + nh / 6)

    if x1 < 0 or x2 < 0 or x2 > w or y2 > h:
        continue

    # Draw a rectangle to see where the moustache will be placed (debugging purposes)
    # cv2.rectangle(roi_color, (x1, y1), (x2, y2), (255, 0, 0), 2)

    # Calculate the width and height of the image with the moustache:
    img_moustache_res_width = int(x2 - x1)
    img_moustache_res_height = int(y2 - y1)
    # Resize the mask to be equal to the region were the glasses will be placed:
    mask = cv2.resize(img_moustache_mask, (img_moustache_res_width, img_moustache
    mask_inv = cv2.bitwise_not(mask)
    img = cv2.resize(img_moustache, (img_moustache_res_width, img_moustache_res_h

    # Take ROI from the BGR image:
    roi = roi_color[y1:y2, x1:x2]
    # Create ROI background and ROI foreground:
    roi_bakground = cv2.bitwise_and(roi, roi, mask=mask_inv)
    roi_foreground = cv2.bitwise_and(img, img, mask=mask)


    # Show both roi_bakground and roi_foreground (debugging purposes):
    # cv2.imshow('roi_bakground', roi_bakground)
    # cv2.imshow('roi_foreground', roi_foreground)

    # Add roi_bakground and roi_foreground to create the result:
    res = cv2.add(roi_bakground, roi_foreground)

    # Set res into the color ROI:
    roi_color[y1:y2, x1:x2] = res

    break

```

一个关键点是图像。该图像是使用图像的alpha通道来叠加的。这样，在图像中只绘制覆盖图像的前景。在下面的截图中，可以看到基于叠加图像的alpha通道创

建的胡子面具



要创建这个掩码，执行以下操作

```
img_moustache = cv2.imread('moustache.png', -1)
img_moustache_mask = img_moustache[:, :, 3]
```

在下面截图中可以看到snapchat_augmented_reality_moustache.py脚本的输出



以下截图中包含的所有胡子都可以用在你的增强现实应用中



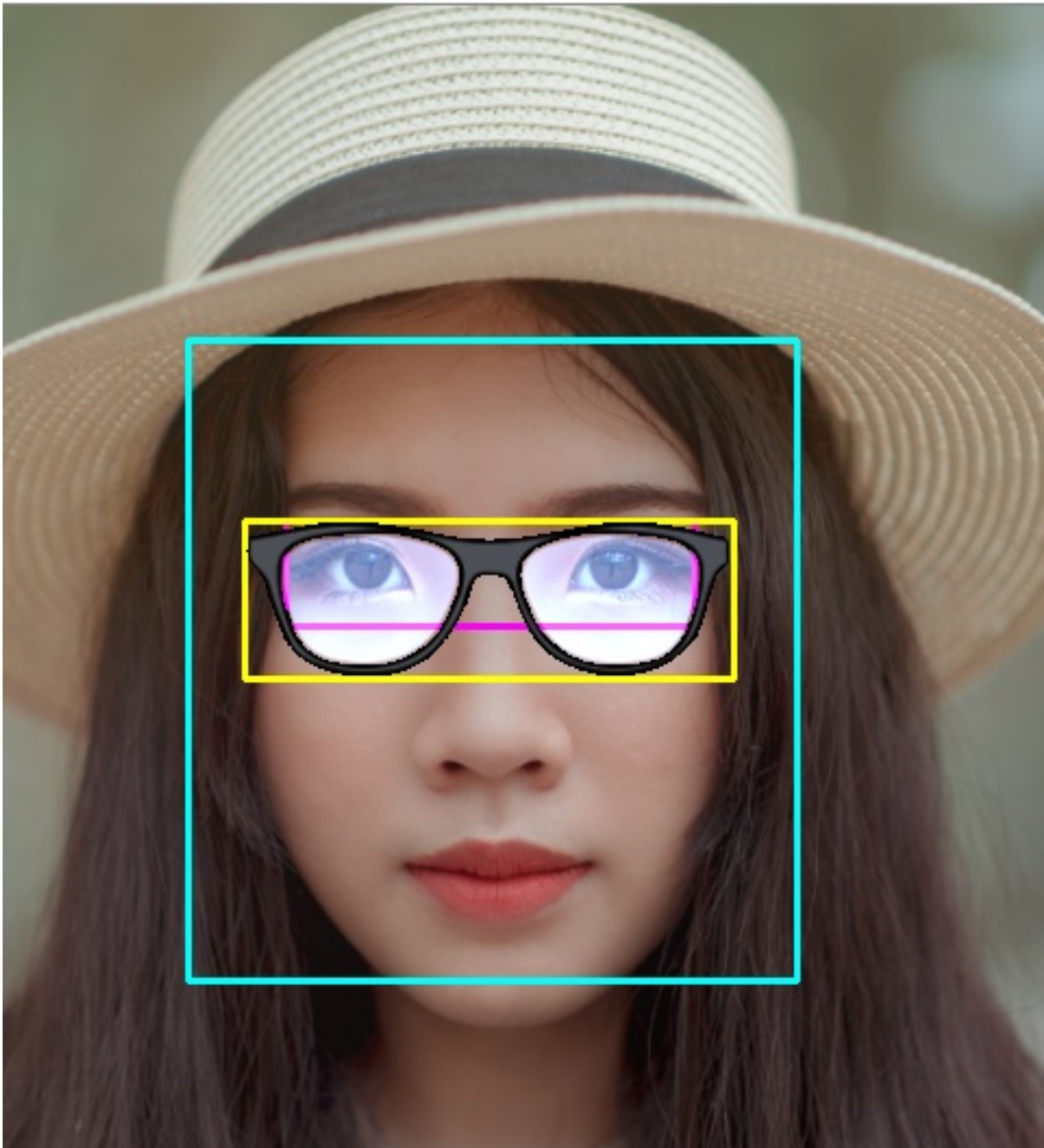
实际上，我们还创建了moustache.svg文件，其中包括这六种不同的moustaches。

基于snapchat的增强现实OpenCV眼镜覆盖

以类似的方式，我们还编写了snapchat_agumeted_reality_glasses.py脚本，将一幅眼镜覆盖在被检测到的脸部的眼睛区域。在这种情况下，为了检测图像中的眼睛，使用了一对眼睛检测器。因此，应该相应地创建分类器

```
eyepair_cascade = cv2.CascadeClassifier("haarcascade_mcs_eyepair_big.xml")
```

在下面截图中，可以看到使用测试图像时算法的输出



青色矩形表示图像中检测到的人脸的位置和大小。洋红色的矩形表示图像中检测到的一对眼睛。黄色矩形表示眼镜将被覆盖的位置，该位置是根据眼镜对区域的位置和大小计算出来的。如你所见，一些透明已添加到玻璃覆盖的形象，以使它们更现实。

这也可以在眼镜图像mask看到，在下面截图中显示



snatchat_augmented_reality_glasses.py 输出截图



所有这些眼镜都可以在下面的截图中看到



最后，创建了`glasses.svg`文件，其中包括六种不同的眼镜。可在增强现实应用中使用所有这些眼镜。

QR码检测

为了完成这一章，将学习如何在图像中检测二维码。这样，二维码也可以作为增强现实应用的标记。`detectanddecode()` 函数检测并解码包含二维码的图像中的二维码。图像可以是灰度或颜色(BGR)。

此函数返回以下内容

- 返回找到的二维码的顶点数组。如果没有找到二维码，这个数组可以是空的。
- 经过校正和二值化的二维码被返回。
- 返回与此二维码关联的数据。
-

在`qr_code_scanner.py`脚本中，用前面提到的函数检测和解码二维码。下面对要点进行注释。

首先，加载图像，如下所示

```
image = cv2.imread("qr_code_rotate_45_image.png")
```

接下来，用以下代码创建二维码检测器

```
qr_code_detector = cv2.QRCodeDetector()
```

然后，调用`cv2.detectAndDecode()`函数，如下所示

```
data, bbox, rectified_qr_code = qr_code_detector.detectAndDecode(image)
```

在解码数据之前检查是否找到二维码，并使用`show_qr_detection()`函数显示检测结果

```
if len(data) > 0:
    print("Decoded Data : {}".format(data))
    show_qr_detection(image, bbox)
```

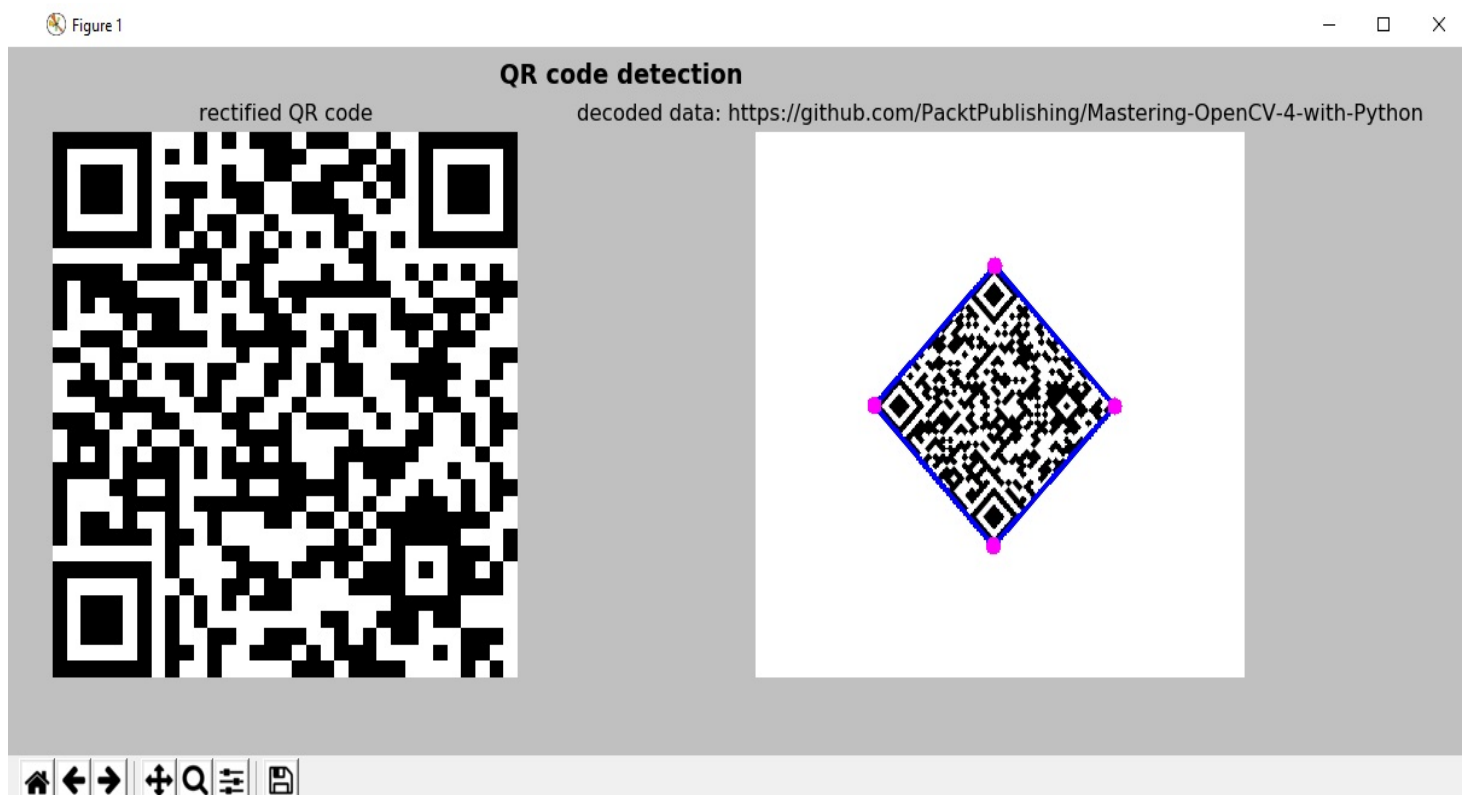
`show_qr_detection()`函数绘制检测到的二维码的线条和边角

```
``` python
def show_qr_detection(img, pts):
 """Draws both the lines and corners based on the array of vertices of the found Q
 pts = np.int32(pts).reshape(-1, 2)
 for j in range(pts.shape[0]):
 cv2.line(img, tuple(pts[j]), tuple(pts[(j + 1) % pts.shape[0]]), (255, 0, 0)

 for j in range(pts.shape[0]):
 cv2.circle(img, tuple(pts[j]), 10, (255, 0, 255), -1)
```

`qr_code_scanner.py`脚本的输出可以在下面截图中看到

div align=center>



<div align=left

在截图中，可以看到经过校正和二值化的二维码(左侧)，以及被检测到的标记(右侧)，其中有蓝色边框，以及突出显示检测结果的品红方形点。

## 总结

在这一章中，介绍了增强现实，编写了几个示例程序，以了解如何构建标记和无标记增强现实应用程序。此外，我们还了解了如何覆盖简单的模型(包括形状或图像)。

如前所述，为了覆盖更复杂的模型，可以使用PyOpenGL (Python OpenGL)。在本章中，为了简化，没有使用这个库。

如何创建一些有趣的基于快照的过滤器。值得注意的是，在第11章，人脸检测、跟踪和识别，将会涉及到更先进的人脸检测、跟踪和面部地标位置的算法。本章中编写的基于snapchatb-ased的过滤器可以很容易地进行修改，以包含一个更健壮的管道，从而获得眼镜和胡子应该覆盖的位置。可了解如何检测QR码。QR码在增强现实应用程序中可以用作标记。

在第十章，OpenCV的机器学习，将介绍机器学习，机器学习如何在计算机视觉项目中使用。

## 问题

1. 初始化ORB检测器，找到keypoints,，用ORB计算加载图像image的描述符
2. 绘制之前检测到的keypoints
3. 创建BFMatcher对象，并匹配前面计算过的descriptors\_1和descriptors\_2
4. 排序之前计算的匹配项，并绘制前20个匹配项
5. 使用ArUco在灰度图像中检测标记
6. 使用ArUco绘制检测到的标记
7. 使用ArUco绘制被拒绝的标记
8. 检测并解码图像中包含的二维码