Jump Game

核心思想

获取棋子到下一个方块的中心点的距离

计算触摸屏幕的时间

点击屏幕

重要方法

计算棋子到下一个方块中心点的距离

1. 使用 adb shell screencap -p 命令获取手机当前屏幕画面
2. 再通过图像上的信息找出棋子的坐标和下一个方块中心点的坐标
3. 然后通过两点间距离公式计算出距离

计算触摸屏幕的时间

T=A * S

其中 S 为上步算出的像素距离，T 为按压时间（ms），A 为一个系数这个系数会随着屏幕分辨率的变化而变化，在 1920*1080 的屏幕下这个系数为 1.35，在 2560*1440 的屏幕下这个系数为 1.475

点击屏幕

adb shell input swipe x y x y time(ms)

这条命令能够点击手机屏幕 x,y 位置 time(ms)

图像处理部分源码解析

图像处理部分代码都在 find_piece_and_board(im) 方法中

通过输入的图像 im 计算出棋子的坐标点以及下一个方块中心的坐标点

在 find_piece_and_board 的方法中一进来就是下面的两个嵌套在一起的 for 循环：

```
for i in range(int(h / 3), int(h * 2 / 3), 50):
  last_pixel = im_pixel[0, i]
  for j in range(1, w):
    pixel = im_pixel[j, i]
    # 不是纯色的线，则记录 scan_start_y 的值，准备跳出循环
    if pixel[0] != last_pixel[0] or pixel[1] != last_pixel[1] or pixel[2] != last_pixel[2]:
      scan_start_y = i - 50
      break
  if scan_start_y:
    break
```

这段代码的作用就是从屏幕 2/3 的位置向下寻找不是纯色的线。并将找到位置的纵坐标-50 作为，寻找棋子和方块的起始坐标。这样可以简化以后搜索的工作量，因为在这个横坐标以上是没有东西的。

接下来是查找棋子坐标的代码

```
# 查找棋子坐标
 # piece_x_sum 横坐标总量 piece_x_c 点的个数 piece_y_max 纵坐标最大值
 # 从 scan_start_y 开始往下扫描，棋子应位于屏幕上半部分，这里暂定不超过 2/3
 for i in range(scan_start_y, int(h * 2 / 3)):
  for j in range(scan_x_border, w - scan_x_border): # 横坐标方面也减少了一部分扫描开销
   pixel = im_pixel[j, i]
   # 根据棋子的最低行的颜色判断，找最后一行那些点的平均值，这个颜色这样应该 OK，暂时不提出来
   if (50 < pixel[0] < 60) and (53 < pixel[1] < 63) and (95 < pixel[2] < 110):
    piece_x_sum += j
    piece_x_c += 1
    piece_y_max = max(i, piece_y_max)

 if not all((piece_x_sum, piece_x_c)):
  return 0, 0, 0, 0
 # 平均横坐标
 piece_x = int(piece_x_sum / piece_x_c)
 # 纵坐标最大值-底座一半的高度
 piece_y = piece_y_max - piece_base_height_1_2 # 上移棋子底盘高度的一半
```

查找棋子的重要依据就是棋子的颜色较为单一并且和方块的颜色有较大差距。如果一个像素点的 RGB 像素值在 B(50, 60), G(53, 63), R(95, 110)范围内那么就认为这个像素点是属于棋子的。根据以上信息就能计算出棋子的平均横坐标，以及最大的纵坐标值。

所以不难计算出棋子坐标（棋子平均横坐标， 棋子最大纵坐标 - 底座一半的高度）其中底座一半的高度因手机分辨率而异。需要提前配置好。

最后是查找下一个方块中心点的坐标的代码

```
# 寻找最高的棋盘
 # 棋盘不会和棋子在同一侧
 # 限制棋盘扫描的横坐标，避免音符 bug
 if piece_x < w / 2:
  board_x_start = piece_x
  board_x_end = w
 else:
  board_x_start = 0
  board_x_end = piece_x
 for i in range(int(h / 3), int(h * 2 / 3)):
  last_pixel = im_pixel[0, i]
  if board_x or board_y:
```

```
    break
  board_x_sum = 0
  board_x_c = 0
  for j in range(int(board_x_start), int(board_x_end)):
    pixel = im_pixel[j, i]
    # 下一个棋盘紧贴着棋子
    # 修掉脑袋比下一个小格子还高的情况的 bug
    if abs(j - piece_x) < piece_body_width:
      continue
    # 修掉圆顶的时候一条线导致的小 bug，这个颜色判断应该 OK，暂时不提出来
    if abs(pixel[0] - last_pixel[0]) + abs(pixel[1] - last_pixel[1]) + abs(pixel[2] - last_pixel[2]) > 10:
      board_x_sum += j
      board_x_c += 1
  if board_x_sum:
    # 最高棋盘的平均横坐标
    board_x = board_x_sum / board_x_c
  last_pixel = im_pixel[board_x, i]
```

代码开头通过棋子所在的屏幕位置限制搜索的宽度，如果棋子在屏幕左边那么就在屏幕右边搜索方块，反之亦然。因为方块和棋子不会在屏幕同一侧。

然后就是自上而下得搜索方块的上顶点。

方块上顶点坐标（ 平均横坐标，当前行的纵坐标）

然后再往下纵坐标+247 的位置开始向上找颜色与上顶点一样的点，为下顶点。

当然此方法有一点局限性对于纯色的平面效果很好但是对于非纯色的平面。可能会判断出错。

如果上一跳命中中间，则下个目标中心会出现 r245 g245 b245 的点，利用这个属性弥补上一段代码可能存在的判断错误

若上一跳由于某种原因没有跳到正中间，而下一跳恰好有无法正确识别花纹，则有可能游戏失败，由于花纹面积通常比较大，失败概率较低

可改进方案

首先是目前方案对于多分辨率需要多个配置文件来记录不同分辨率下的系数以及棋子底盘一半的高度。随机测试了 6 台手机其中有两台手机因没有配饰而无法正常运作

首先是系数 A，观察方程 T=A * S，A 就是一个可训练量，利用机器学习框架比如 TensorFlow，对这个一元一次方程进行拟合。

观察棋子底盘一半的高度在代码中的作用。不难发现是为了求出棋子底盘中心的纵坐标。而棋子底盘中心的位置恰恰是棋子最宽的地方。所以可以通过找出棋子最宽处的纵坐标的方式找到棋子底盘中心的纵坐标。这样就摆脱了对配置文件的依赖，能让代码在任何手机上正常运行。

其次是对于方块中心坐标位置的判断方法出错率较高，虽然有中心白点可以弥补但是在大量跳跃的过程中还是会出现错误。3 太手机不停运行了一下午，最高分只有 2009 分。

现方法出错率高的原因是使用纯颜色方法判断，但是在实际游戏中颜色丰富的方块也不少。如果想改变就不能依赖颜色方法判断，而应该通过几何图像的形状来计算方块的位置。不难发现游戏中方块只有棱形和圆形两种形状。

首先通过 canny 或其他轮廓查找算子提取出图像的轮廓，然后通过霍夫变换提取出圆形和棱形的中心坐标。