说明

运动控制技术大作业。

微信跳一跳外挂逐像素扫描go语言版。

开发时间：6.2-6.5三天。

本来是markdown格式的，上传麻烦，就写成了pdf版本。

`git clone https://github.com/burningcl/wechat_jump_hack`

确保手机开发者模式usb连接。

下载后，运行

```
go build main.go
./main
```

会建立一个叫img的目录，存储所有图片，并且标记找到的点。

命中率

目前命中率百分百。

理论上可以一直跳。

跳少了或者多了。

更多是因为弹跳系数的设置不当。

最高分

目前最高分是 21934，运行了1832次。


历史最高分
21934
查看全部排行 ›
再玩一局
历史最高分：21934

手机像素1024 * 2224， 华为。

弹跳系数为1.934。

如何找到棋子的底部

棋子的颜色固定，更具体的说，其底部的颜色在一定范围内。

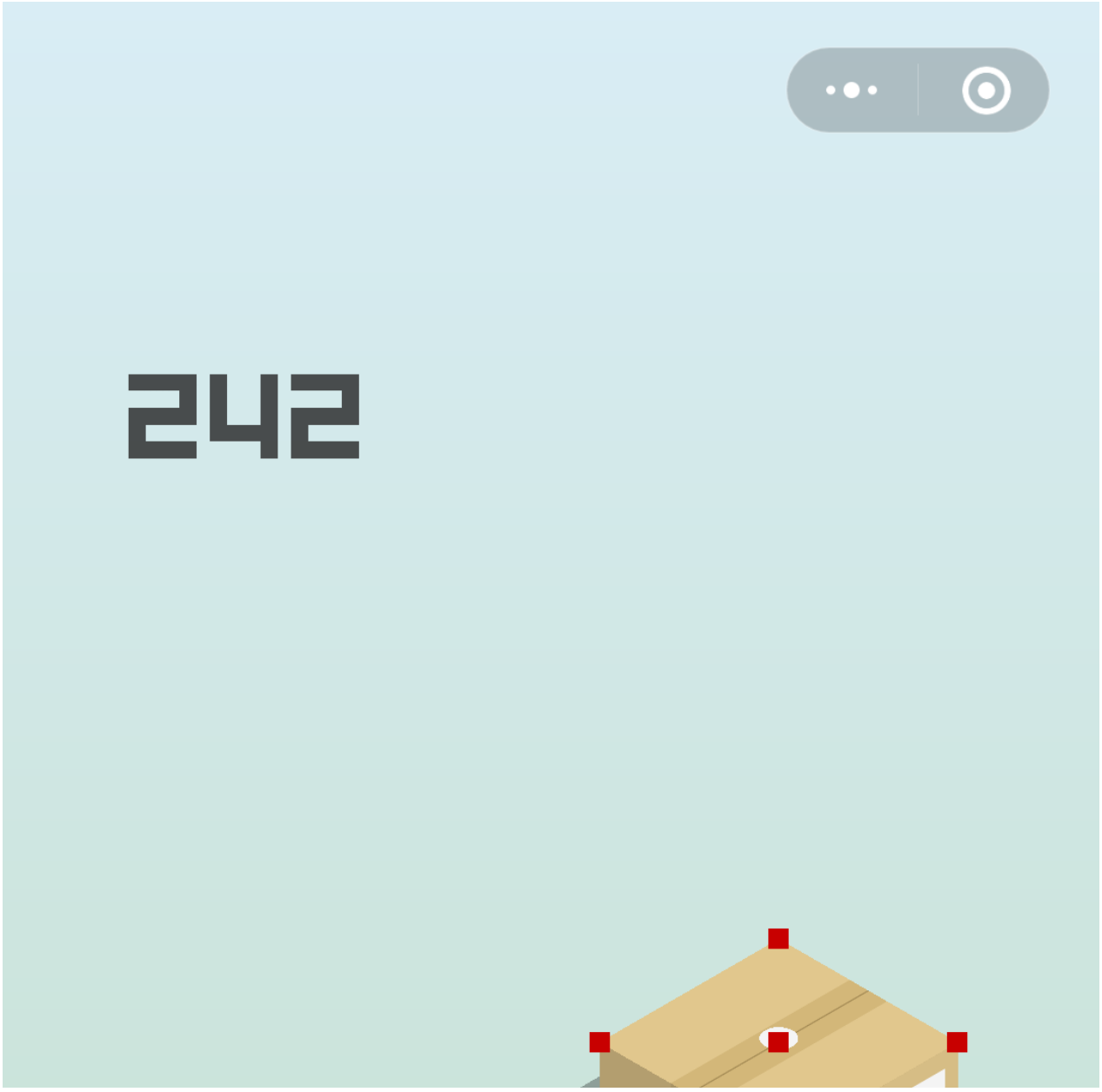
所以我们遍历整个棋盘，计算像素的rgb值，如果其差与固定的棋子颜色在`16`之内，那我们就认为这个像素点代表棋子的底部。

> ⓘ 当棋盘颜色相近时，容易混淆

棋盘颜色相近的时候

左下方的黑色标记，正常是不会识别为棋子的。

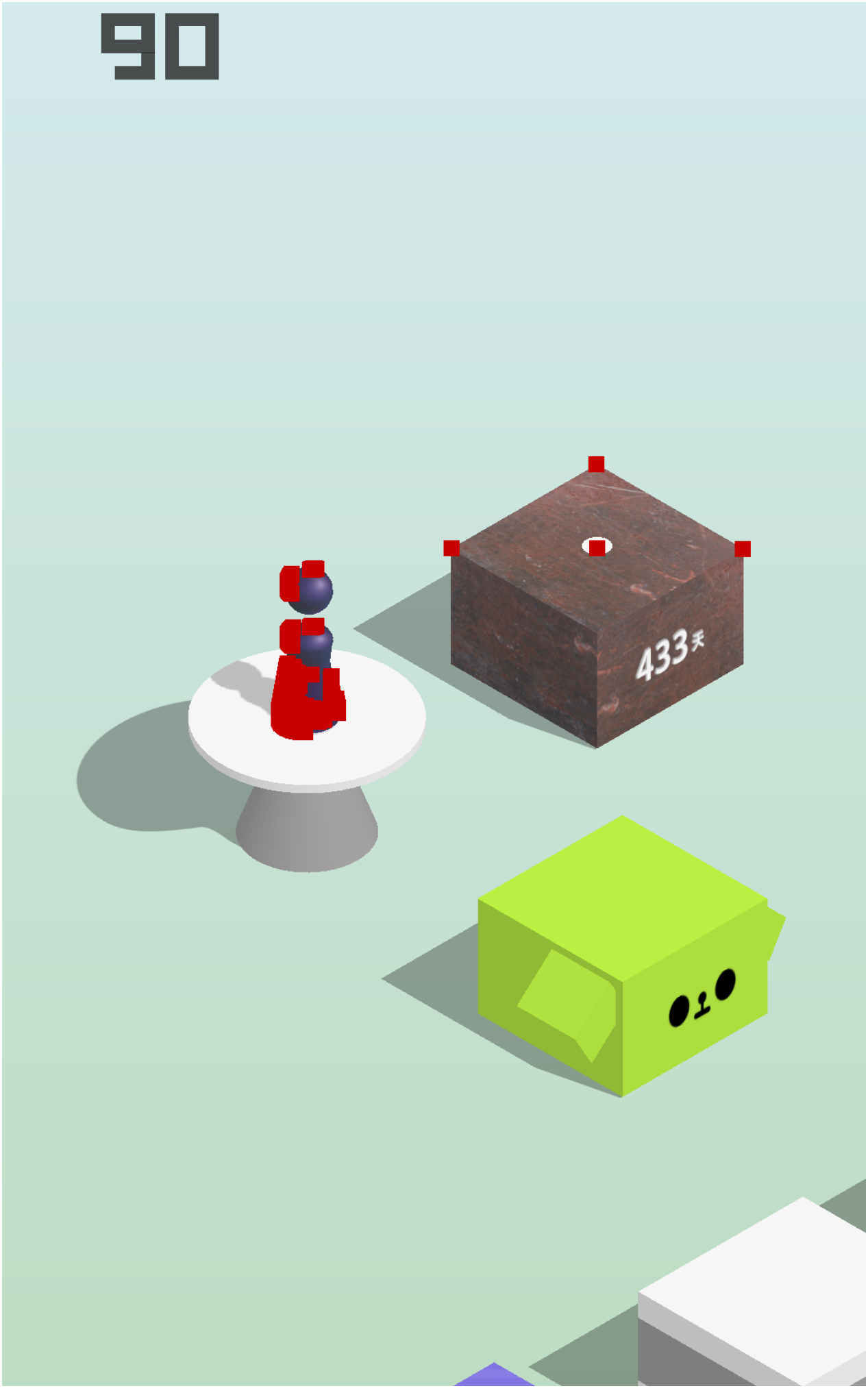
但是当它与蓝色的底色混合后，就会被识别为棋子。

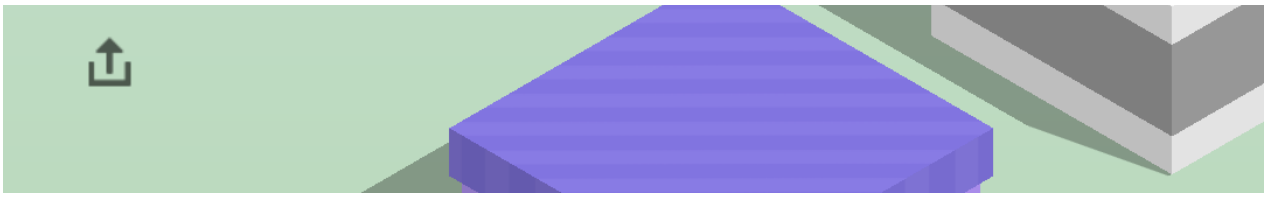


精确识别范围

首先我们标记出一张图片一般会被识别为棋子的部分。

我们可以看出，基本上是一个长宽固定的矩形。

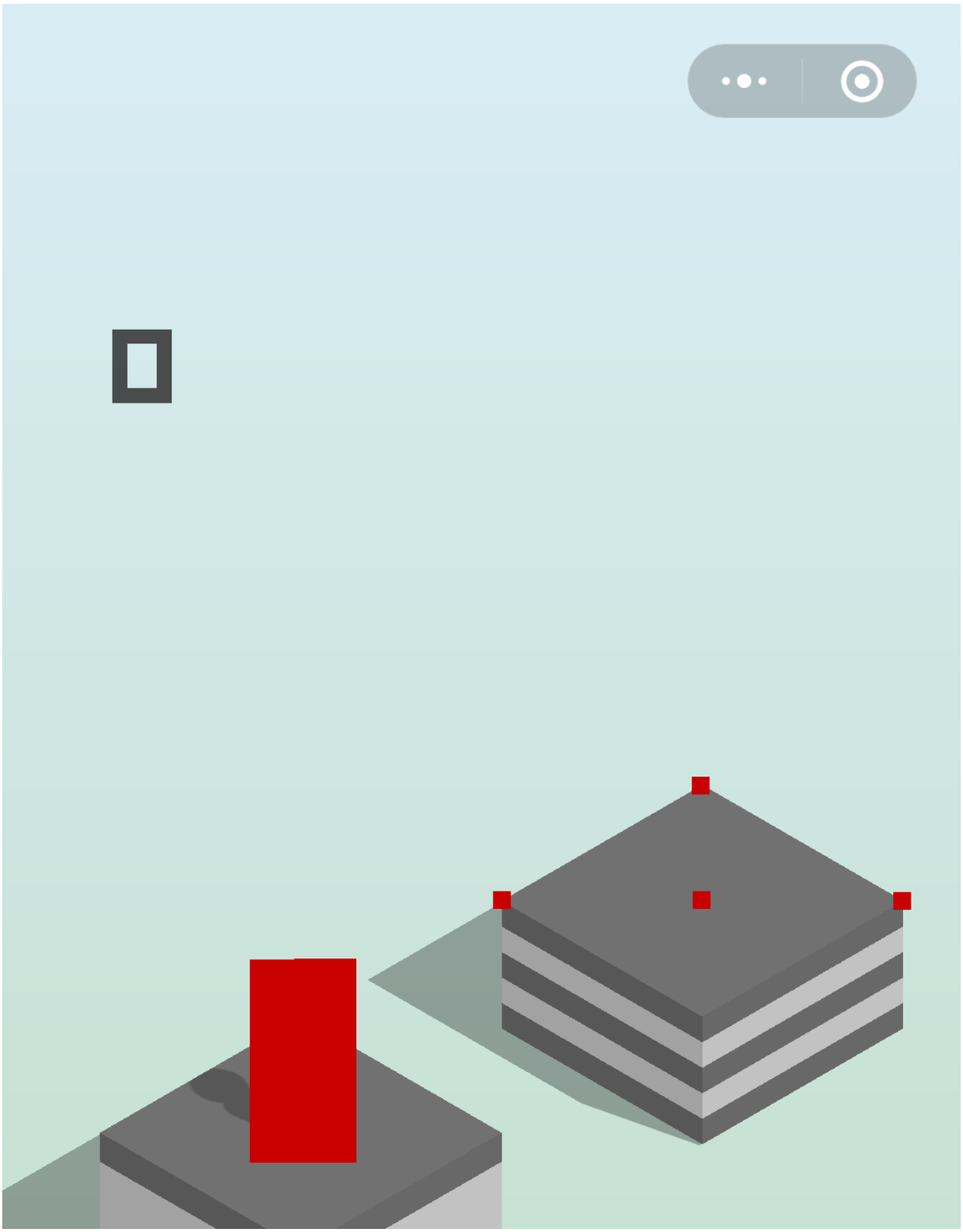

90
433天

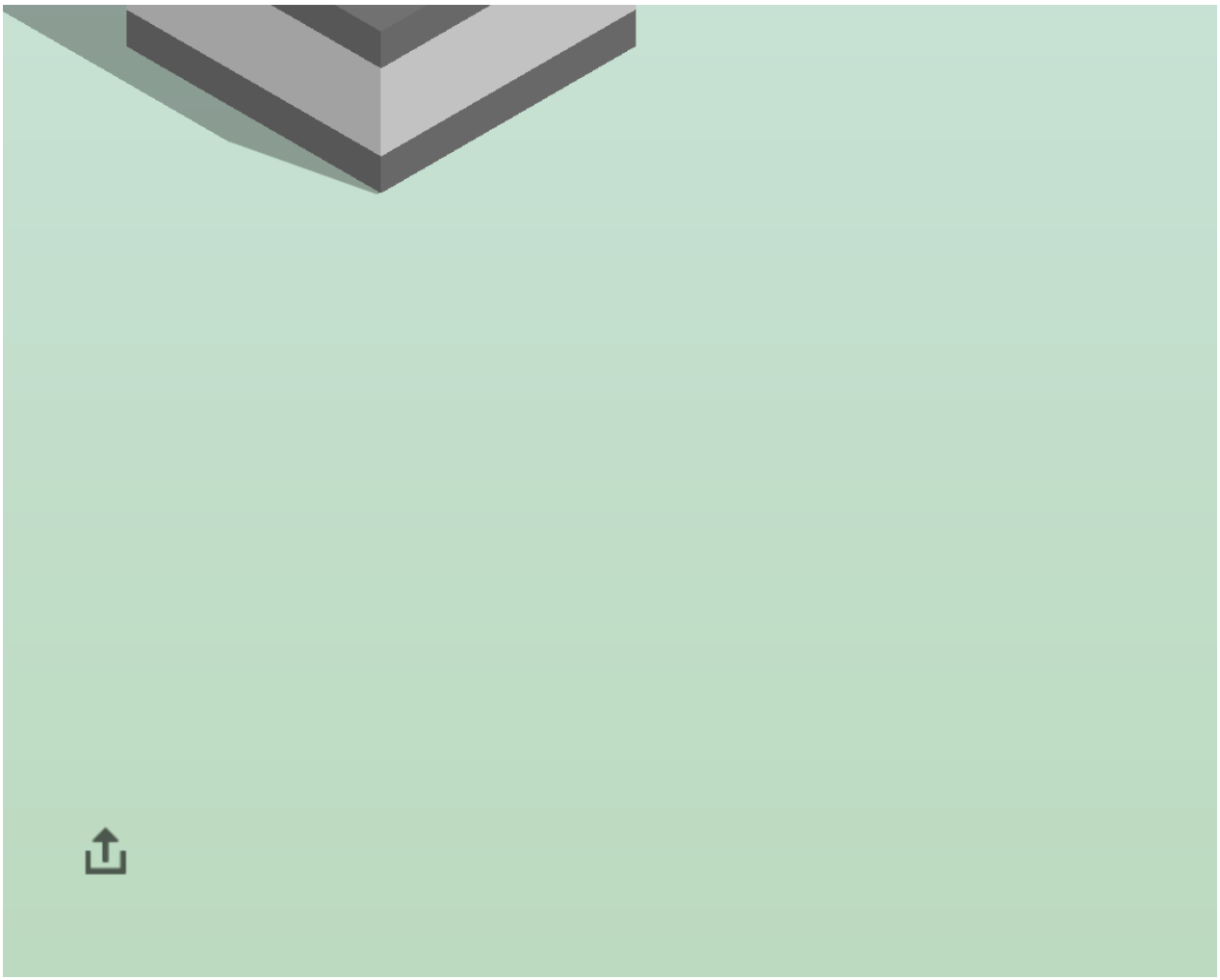


于是我们一行一行扫描，当扫描到第一个符合条件的点的时候。

接下来就只搜索这个矩形区域。

图中红色的点就是我们所说的矩形区域。





如何找到棋盘的左顶点和右顶点

和找顶点一样的做法?

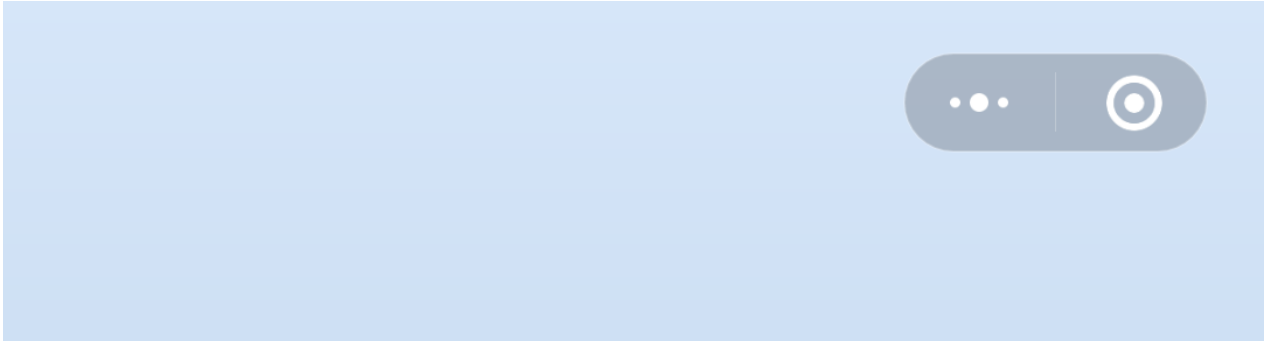
找到`顶点`后，我们`顶点`为矩形的底部中点，在一个矩形范围内搜索颜色跟顶点颜色`相似`的点，并认为这些点都代表棋盘。

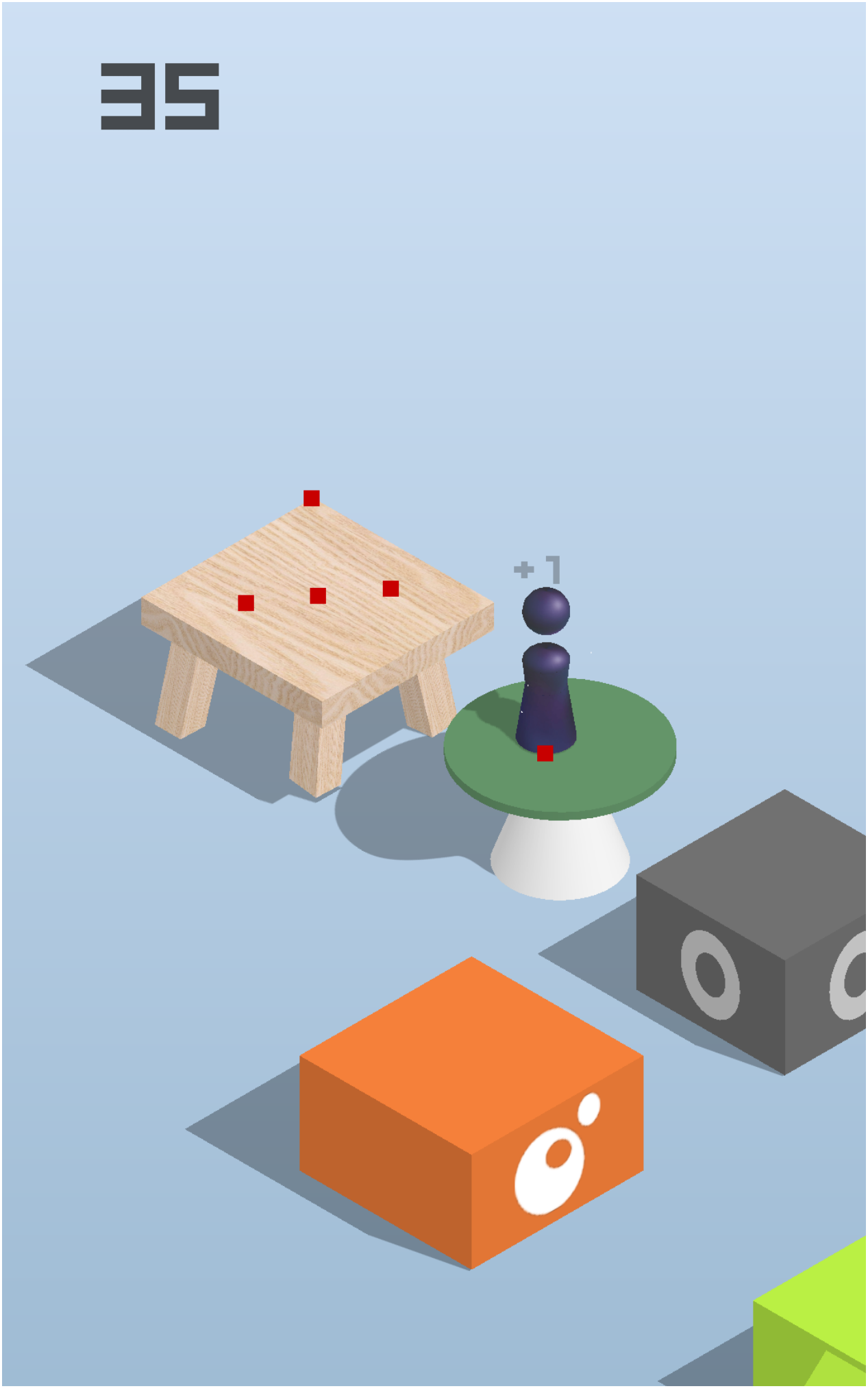
对于这些点，其坐标的x最小的我们认为是`左顶点`，x最大的我们认为是`右顶点`。

> ⓘ 搜索范围不能太大，避免将棋子等也识别进去； 搜索范围也不能太小，避免找不到左右顶点

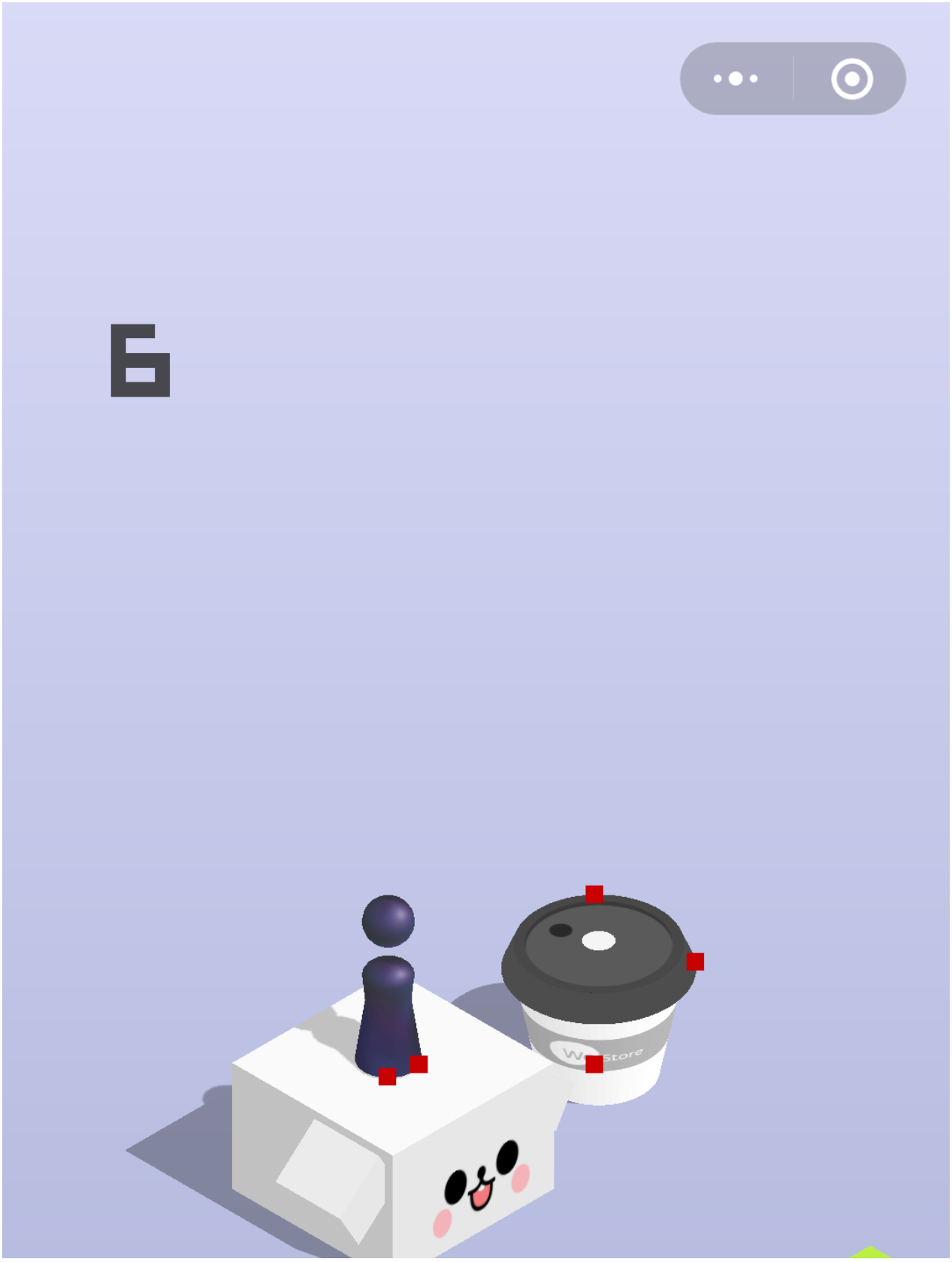
`无法`处理棋盘过小的情况

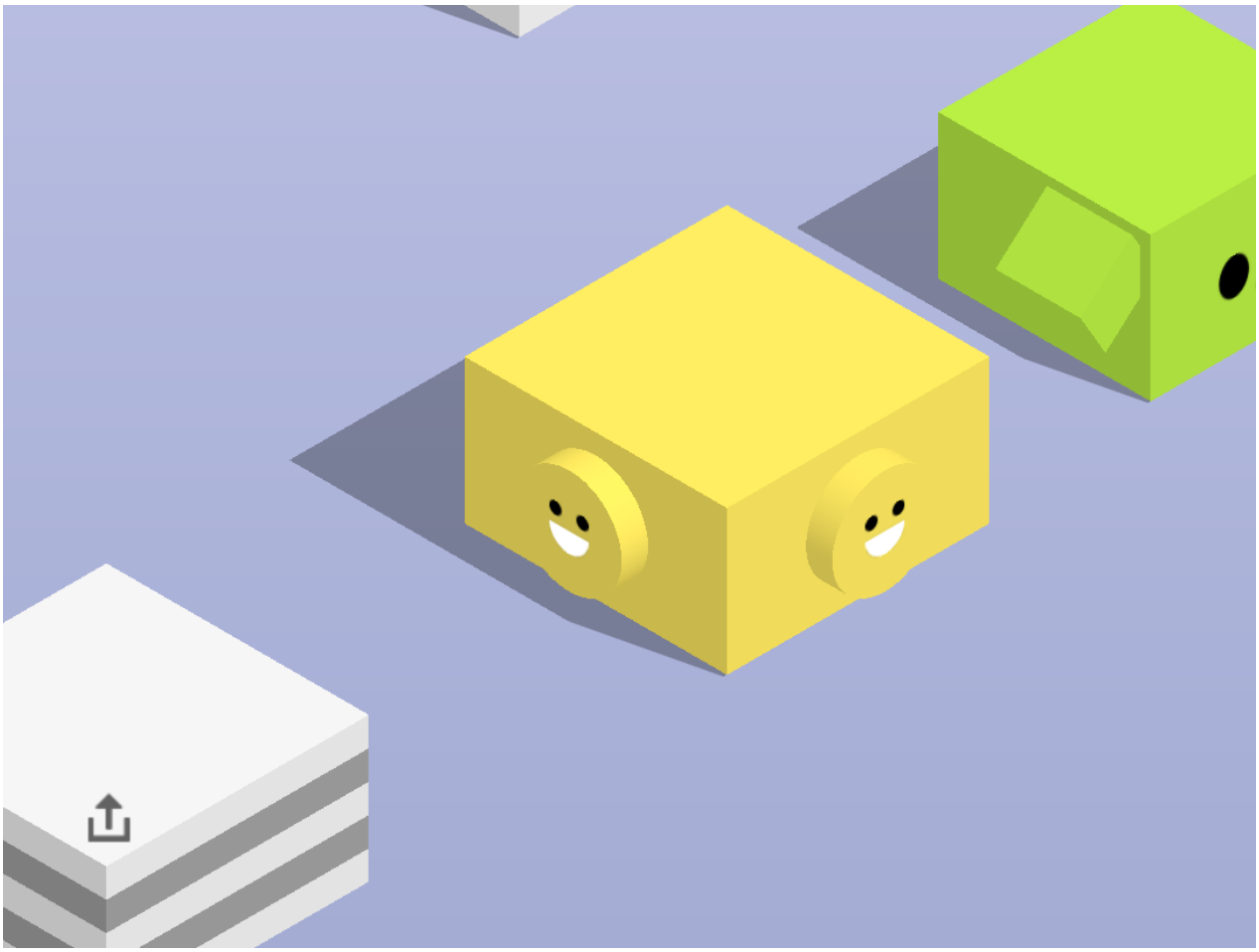
搜索范围一定要大，不然对于正常的棋盘无法找到左右顶点。




35
+1

但是棋盘过小而且和棋子靠的很近时，就会找到搜索范围之外的点。





从中间向两边搜索

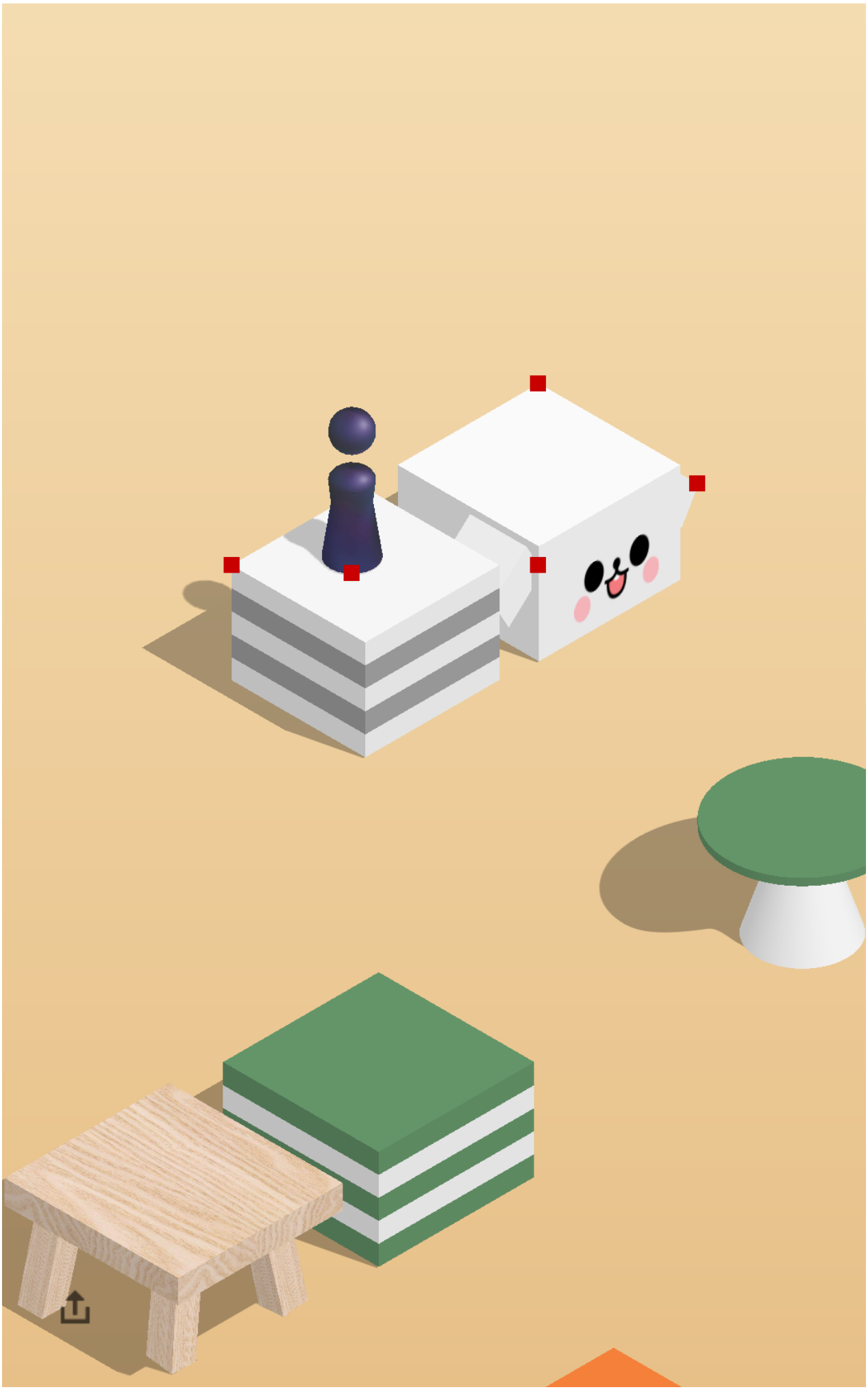
从顶点分别向两边一行一行的搜索，当遇到背景色的时候停止本行的搜索。

两个棋盘十分靠近时

对于图中的情况，当其向左搜索的时候，一直碰不到背景色。

这个时候我们是由于左顶点和右顶点和中心点距离偏差太远，于是我们认为棋子没法完全显示在棋盘内部。

于是取了左顶点的Y值和上顶点的X值作为中间点。



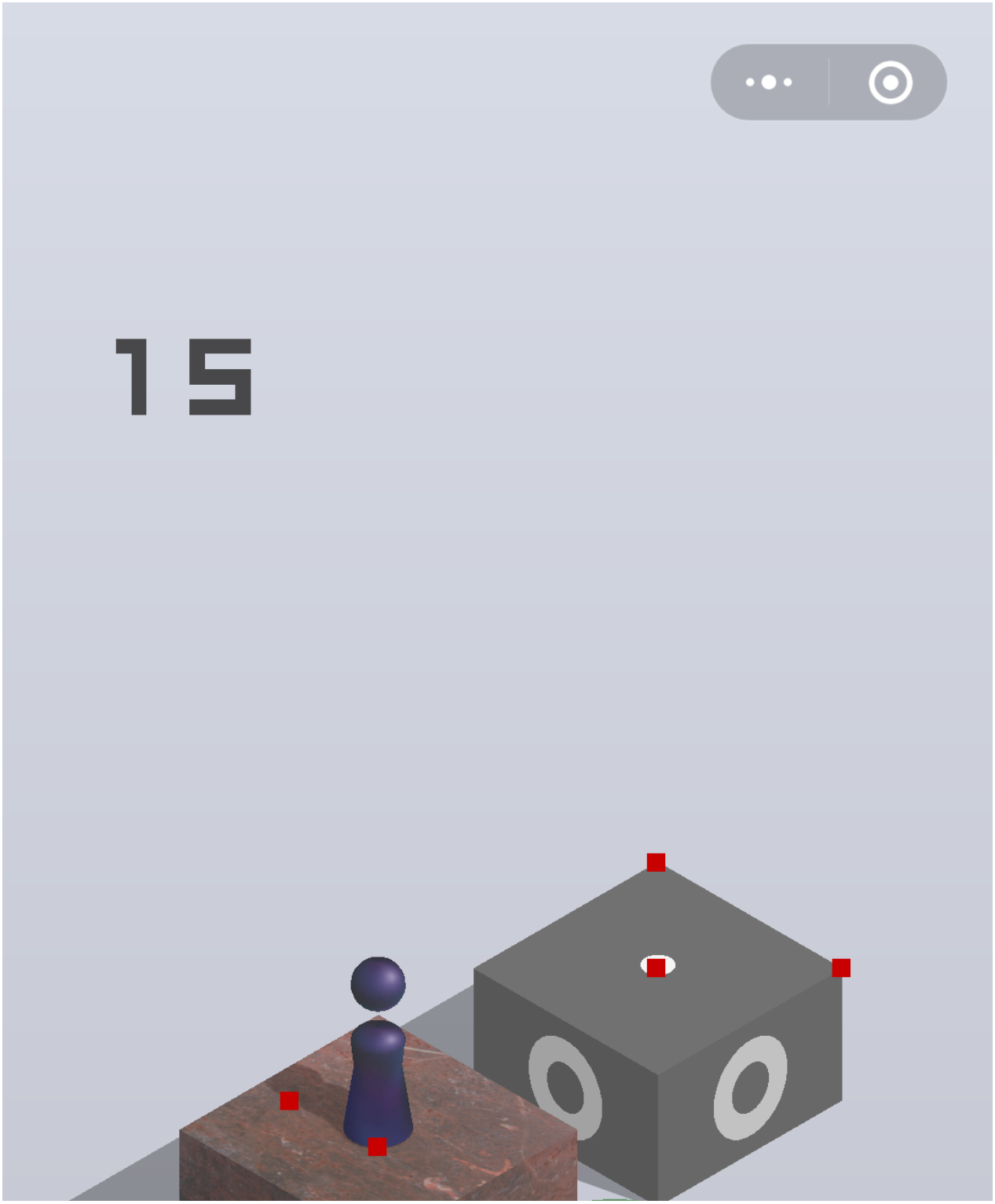
ⓘ 事实上，我们计算出来的点和实际的点对棋子来说差别不大，因此棋子一般都能跳过去；但是为了严谨，我们仍然做了一定优化

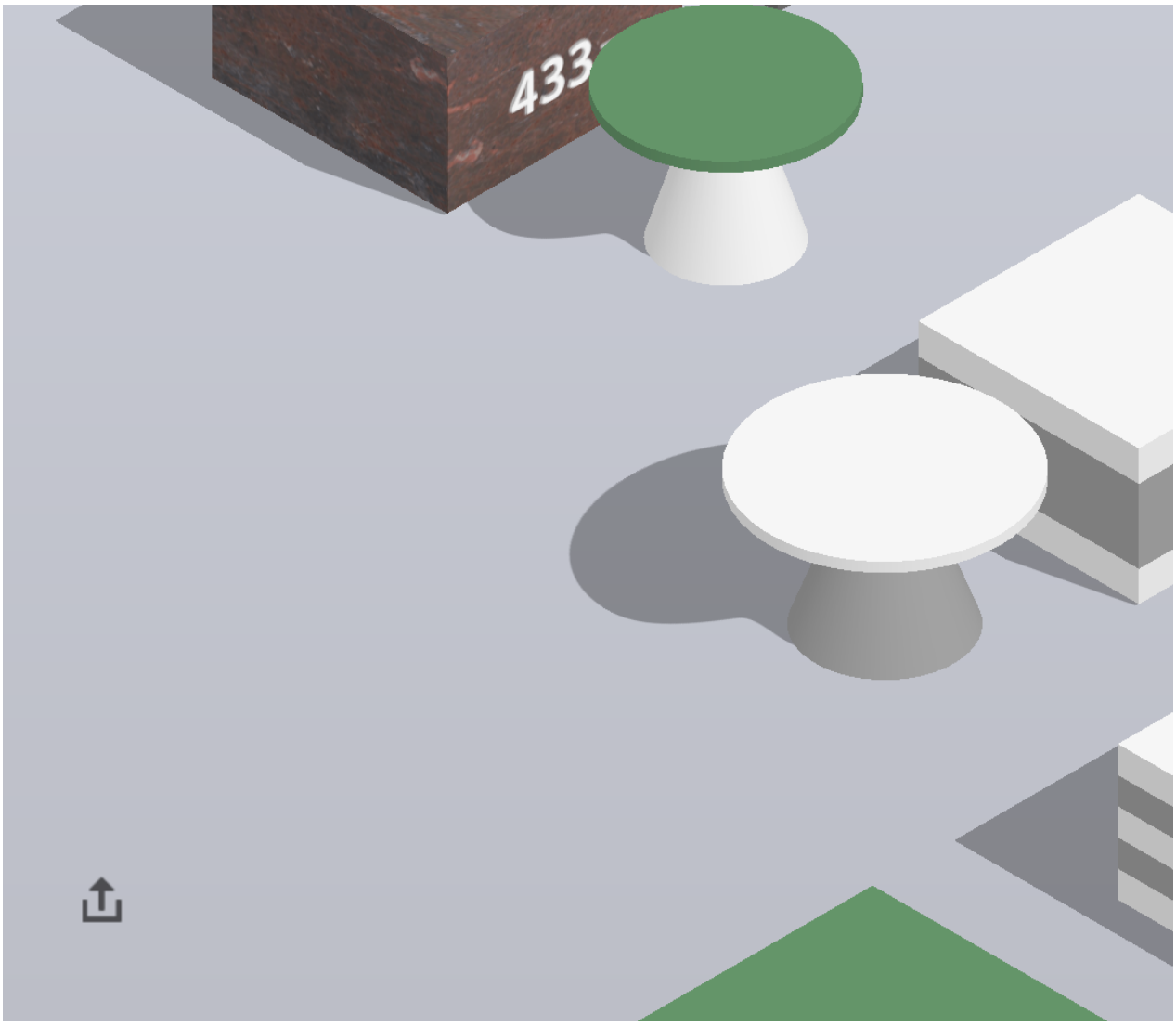
检测距离过近的情况

当左顶点和右顶点和中心点距离偏差太远时。

如果左右顶点距离照片左右边界在一定范围内，则我们认为它是距离过近；

反之则是两个棋盘距离过近。

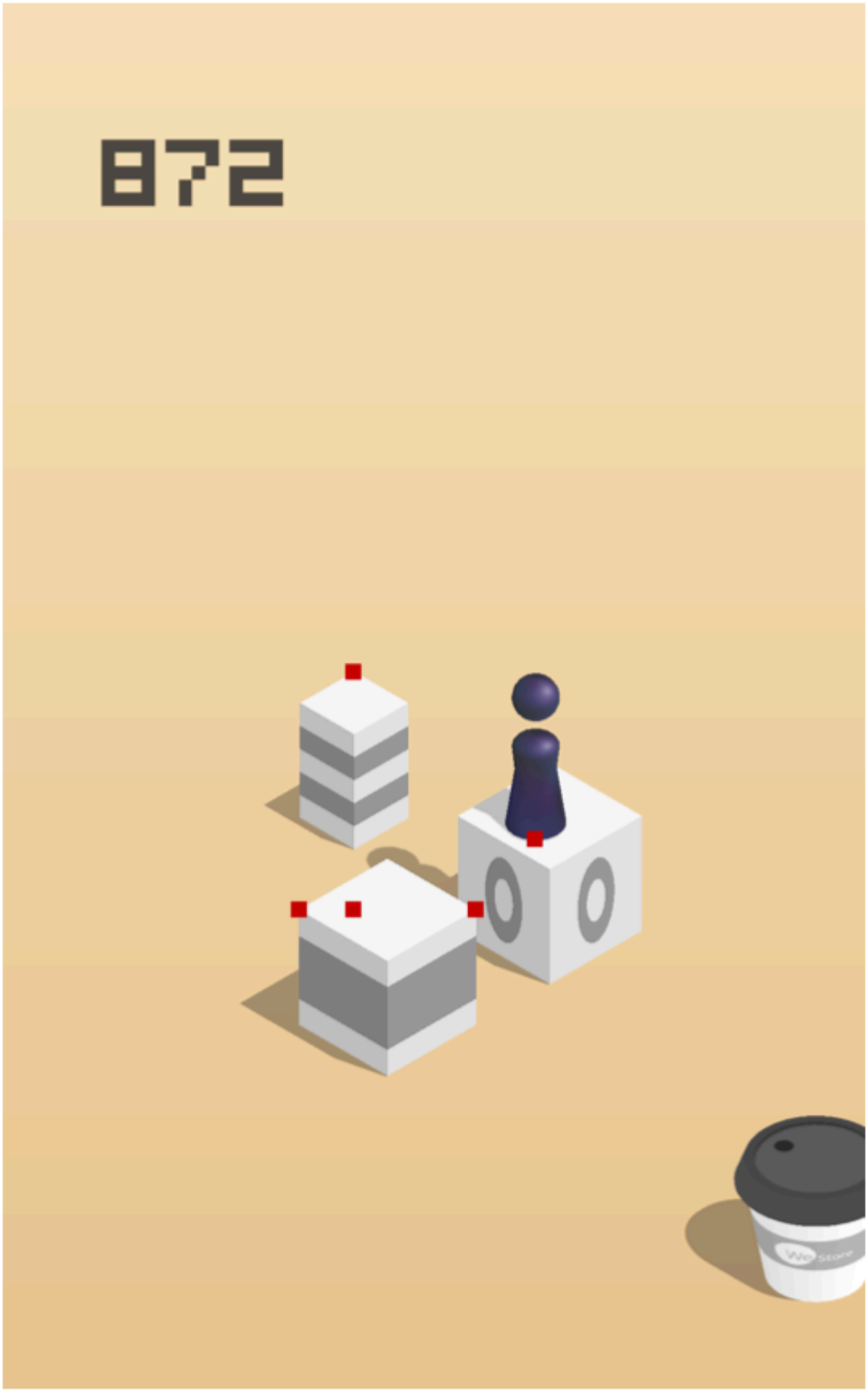




场中棋子混杂时

对于图中这种情况，是由于上下两颗棋盘太近导致的。

因此我们尝试检测棋子过近这种情况。

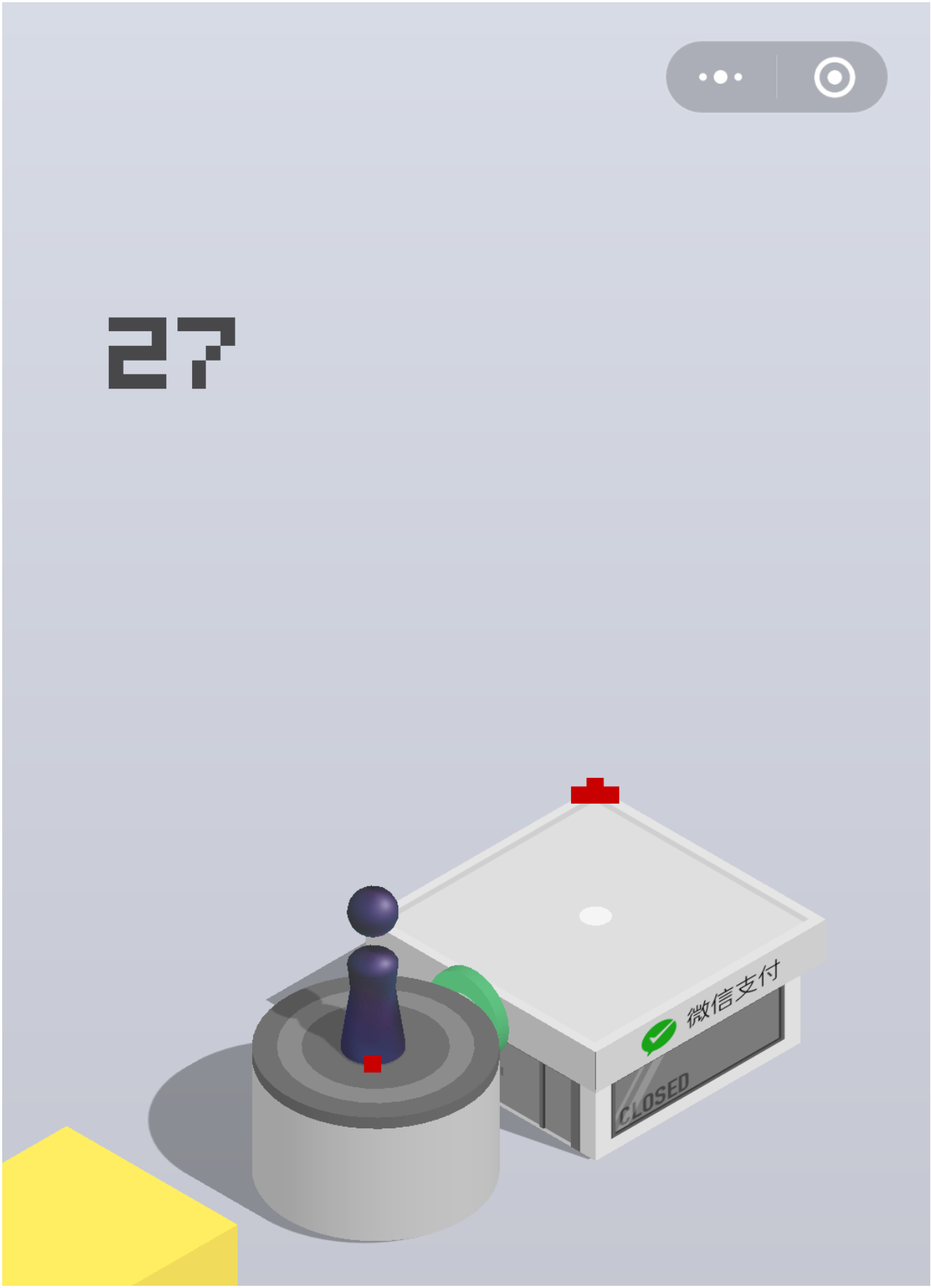


检测上下棋盘距离过近

当我们从上顶点往下，向左右两侧扩散寻找左右顶点的时候。

如果我们发现上顶点往下的点，有背景色的时候，我们就认为已经跳出了当前的棋盘

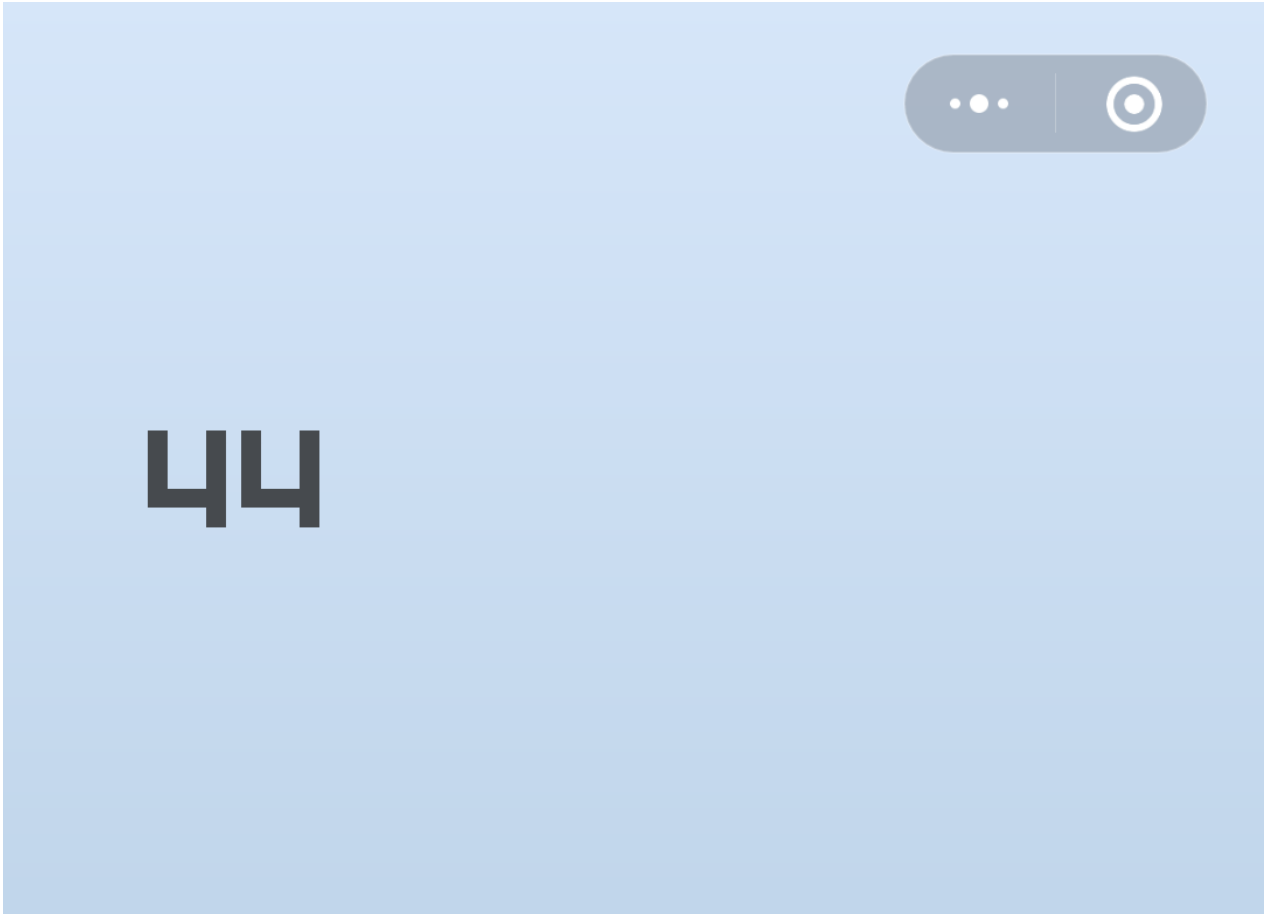
> ⓘ 当棋盘内部颜色存在背景色时，会混淆



因此我们不会一碰到背景色就跳出循环。

而是当y轴上的背景色长度大于某个范围时才会跳出循环。

棋盘和棋子距离很近的情况



如何找到棋盘中心点

从上到下，扫描每一行，对于和背景色不一致的点，我们认为是顶点。

> ⓘ 为了避免扫描到数字，实际扫描的顶点会偏小一点

渐变色的背景

对渐变色的背景，顶部颜色浅，底部颜色深。

我们会采样顶点的颜色。

> ⓘ 由于顶部接近白色，所以实际上我们会采样距离顶部一段距离的点

以及底部出现次数最多的颜色。

> ⓘ 最低部的点事实上不会出现，而且如果采样最低部的话，当背景为粉色的时候，识别不了白色的棋盘。因此采样点距离底部一定距离。并且为了提升速度，会做缓存

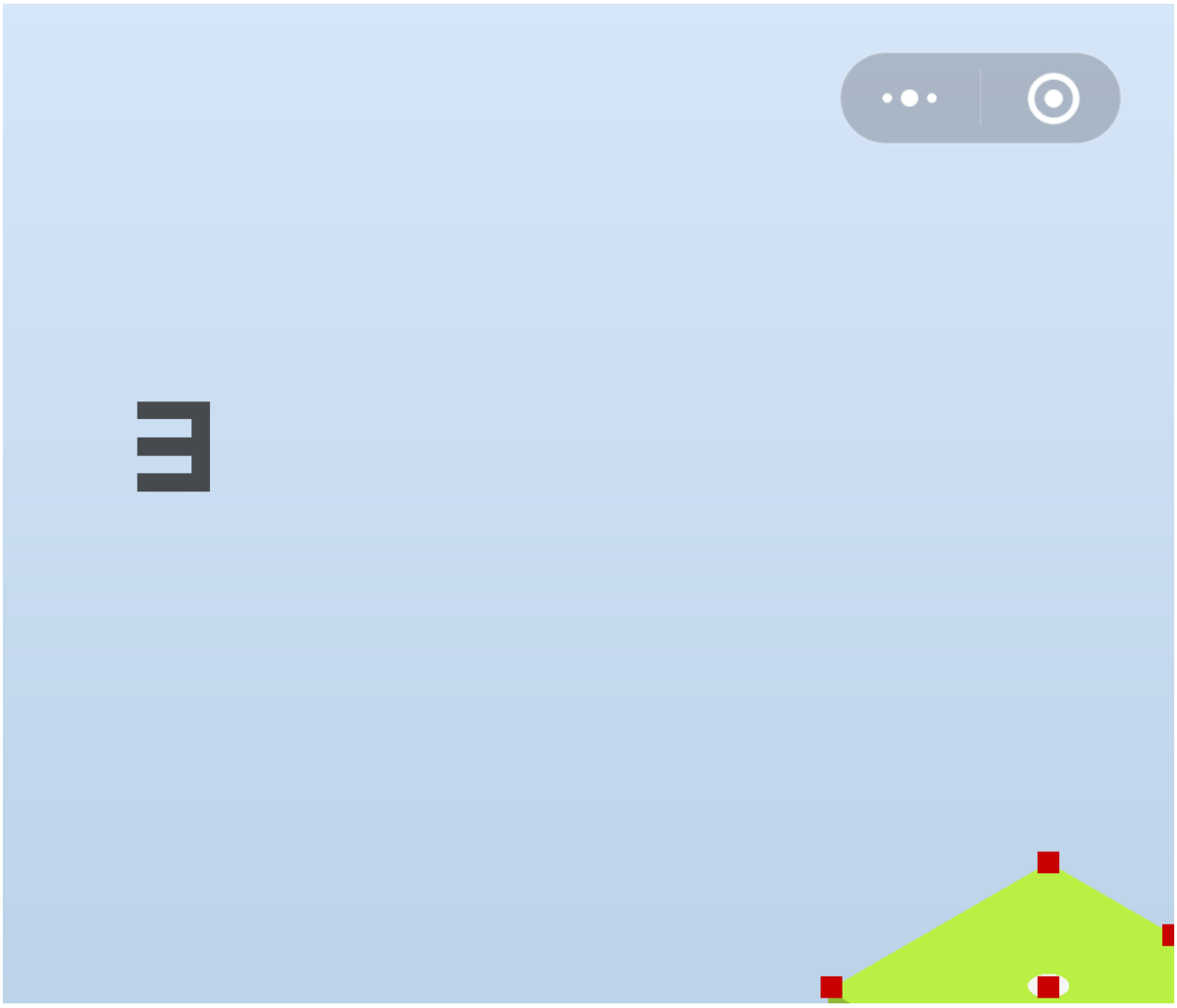
将这个范围称之为背景色。

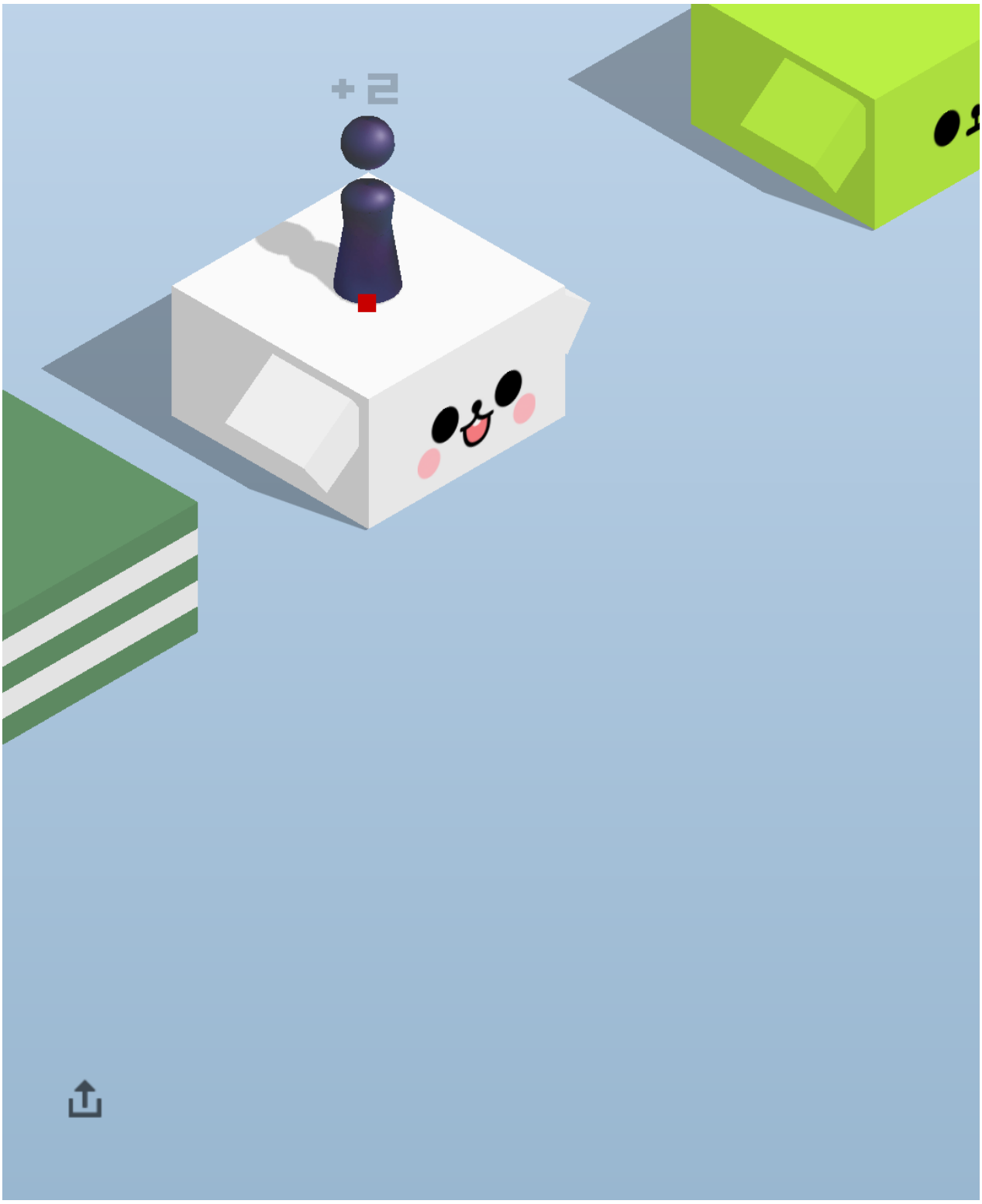
照片显示不了整个棋盘的情况

当左右两个顶点与顶点的x轴距离的差距的绝对值大于一定的数值后。

我们认为棋盘在照片之外。

对于图中的情况，我们直接使用左顶点的Y和顶点的X当作中心点



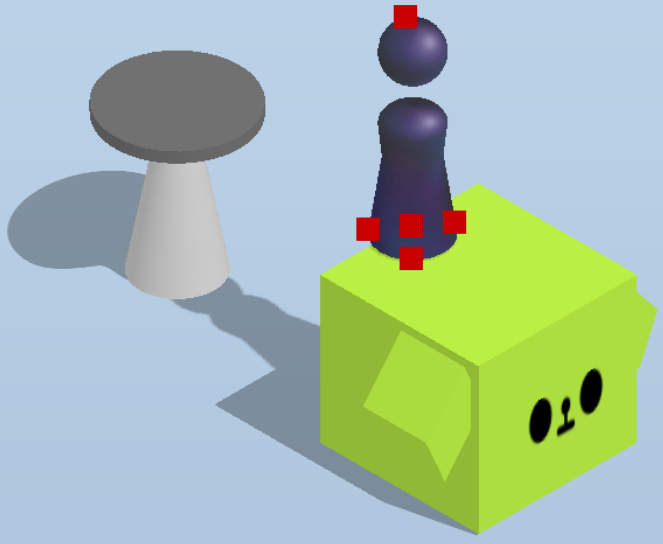


棋子顶点比棋盘高

对于这种情况，我们首先尝试搜索时会过滤棋子的颜色。

> ⓘ 棋子顶部亮光区域和底部颜色差距很大，无法通过颜色过滤

437



过滤距离棋子一定范围内的像素点

我们以棋子的底部为中心，过滤掉所有和棋子底部的距离小于棋子长度的点。

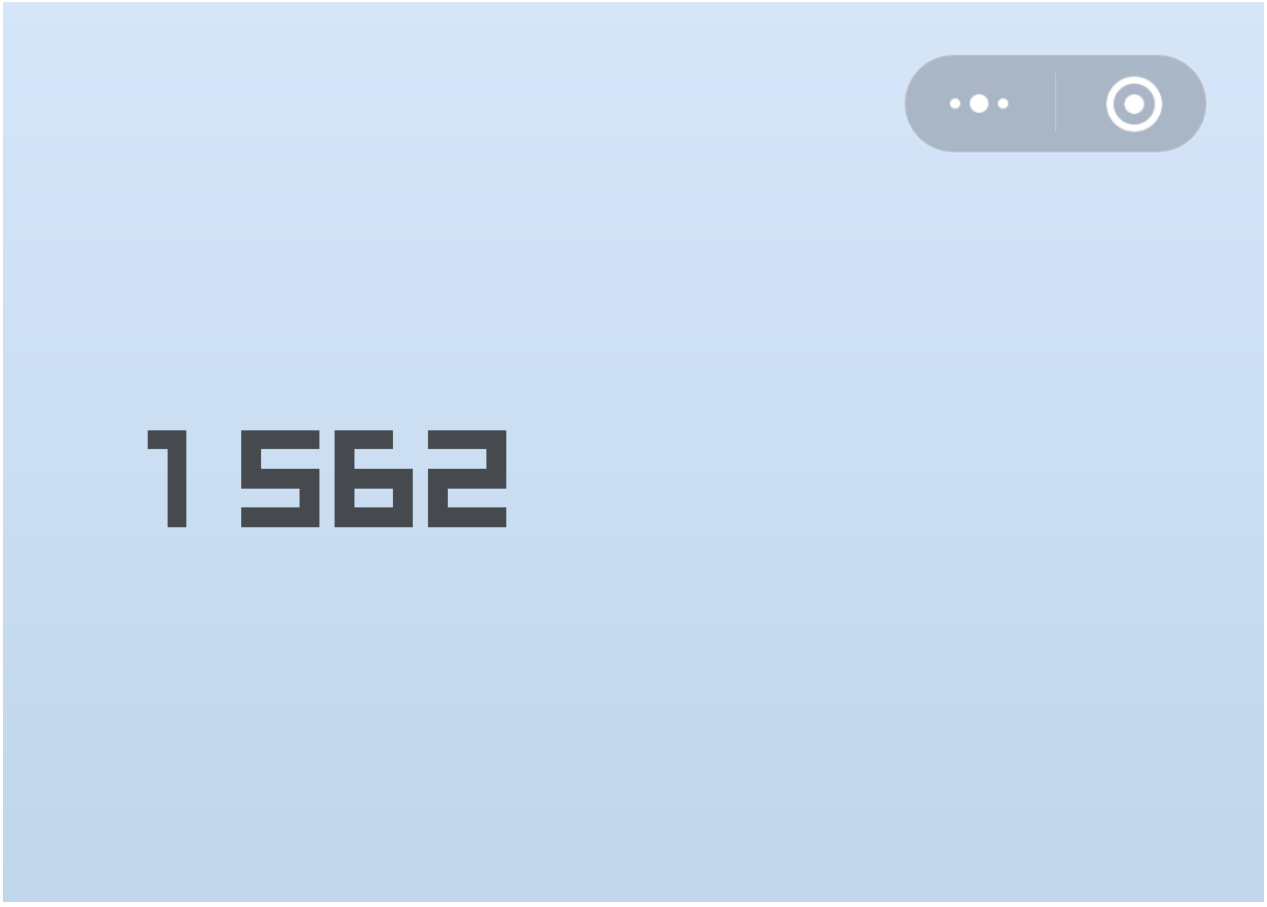
> ⓘ 当棋盘过小且过近时会出错

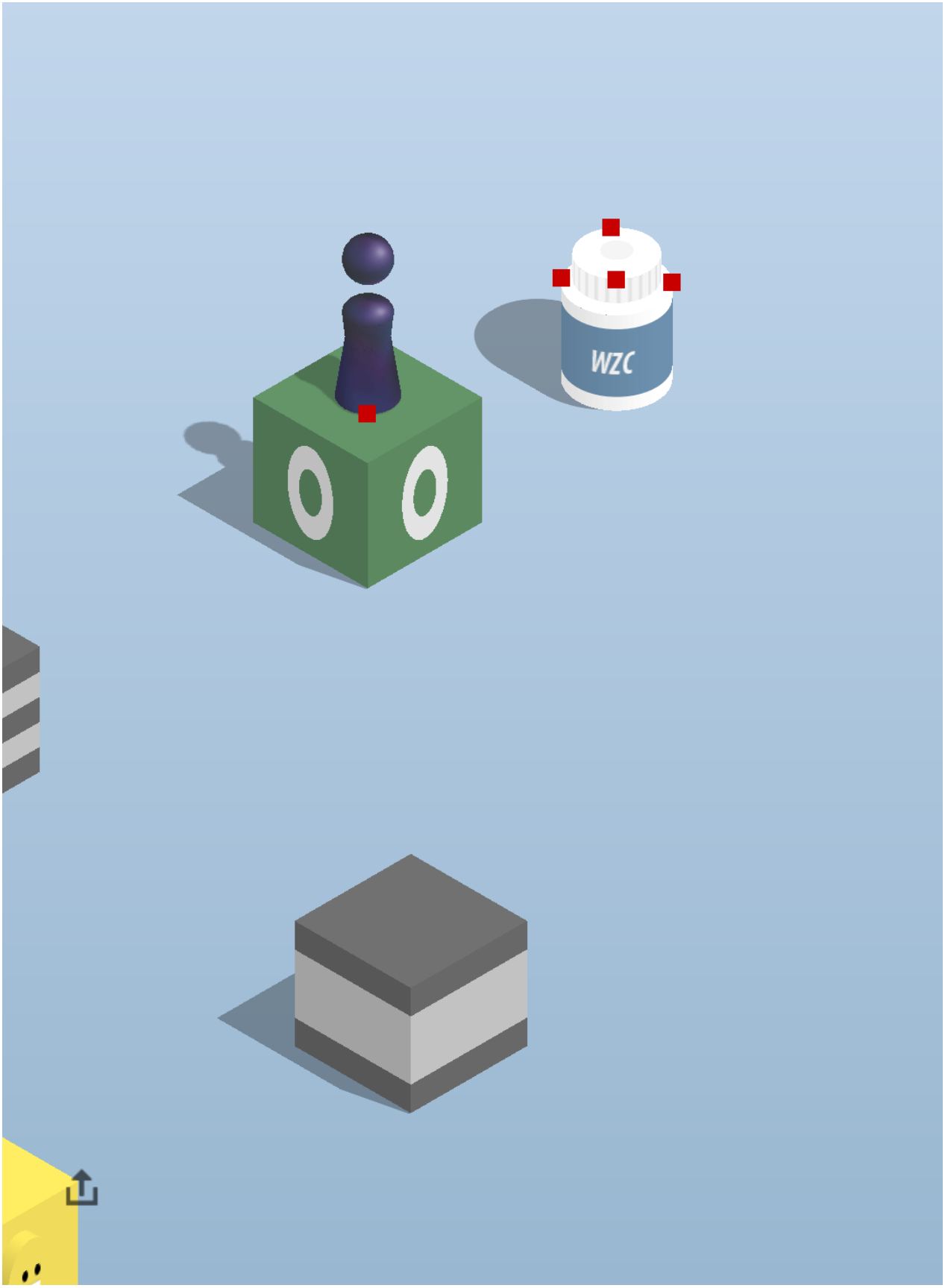
因此我们转变过滤像素点的范围。

过滤掉，以棋子底部为矩形中心，宽为棋子宽度，长为棋子长度的两倍的矩形范围的点。

特殊棋盘

当遇到这种特殊瓶子的时候，无法正常识别。





对于这种特殊的棋盘，我们只能专门开发出一种算法来识别。

当上顶点是瓶盖的颜色的时候

我们会对上顶点距离一定位置的两个点采样。

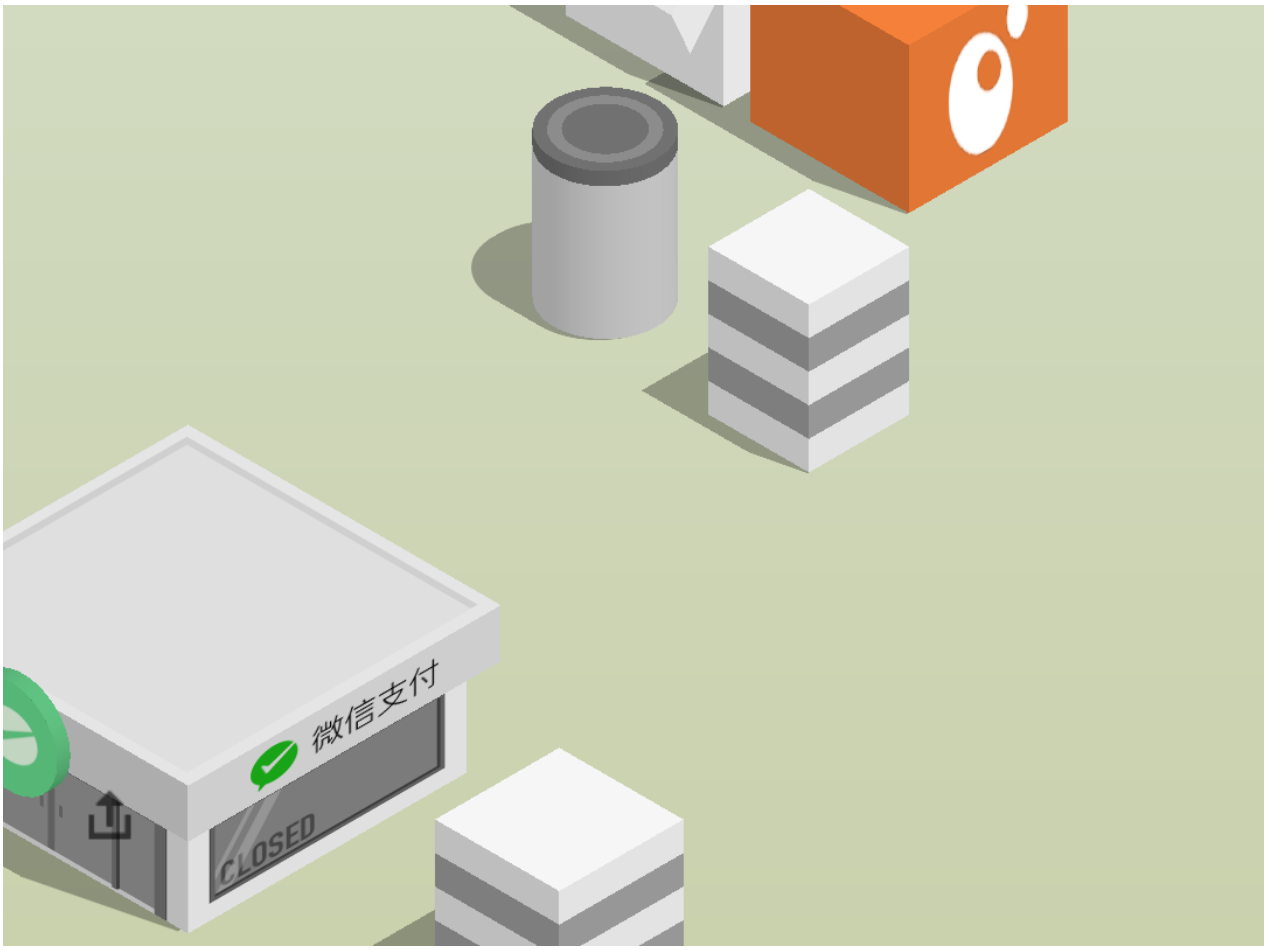
如果都是瓶身的颜色。

那我们就认为这个是特殊的瓶子。

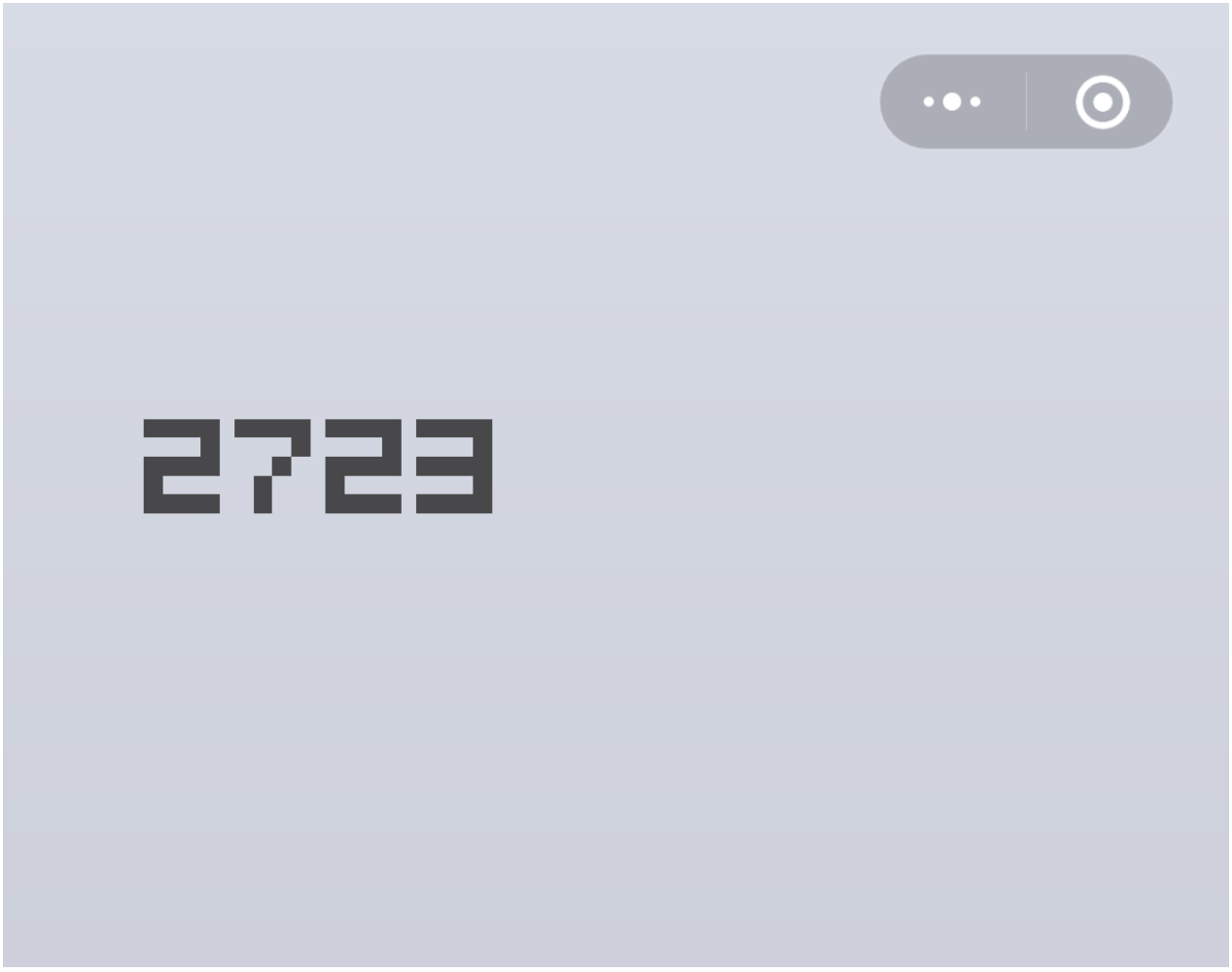
因此我们的中心点为距离上顶点固定位置的一个点。

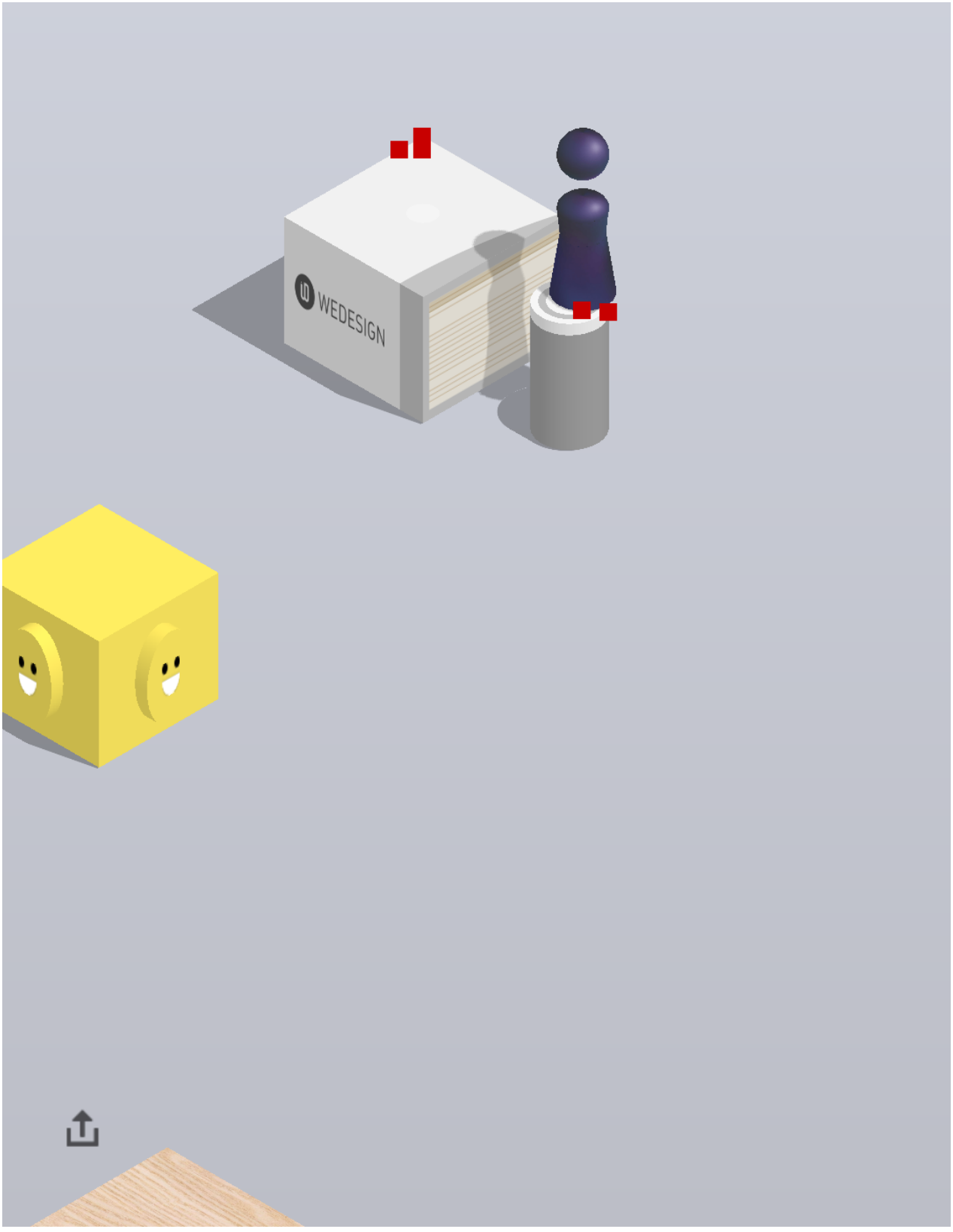
待解决的情况

上下棋盘过于靠近



奇怪的情况?





提高运行速度

虽然我们写了限制搜索范围的部分。

但那更多是为了防止bug，而不是为了提升速度

数据并行化

我们的程序对照片的数据处理主要有三部分

1. 获取照片长宽的
2. 找到棋子底部的点
3. 找到棋盘中心的点

最开始，这三部分并没有数据上的依赖关系。

于是我开启了三个线程分别处理这三个任务。

主线程等待另外三个线程结束任务。

因此主线程等待时间为三个线程中最大的。

同步

对于第三个任务，其分为

1. 找上顶点
2. 找左右顶点
3. 计算中心
4. 标记所有点

由于第三个任务的第一个子任务，需要得到棋子中间的点的位置。

而这是第二个任务的结果。

因此第三个任务在执行第一个子任务前，会等待第二个任务的信号。而收到信号前线程会挂起等待。

等待时间

因此主线程的实际等待时间为第三个任务的运行时间。