基于手势识别的游戏制作

金镇雄、姜恒、王悦

一、实验器材

电脑摄像头

二、模块要求

本实验中使用了 python 之 cv2、pgzrun、numpy、paddlex 模块，安装方法如下：

```
pip install opencv-python
pip install pgzero
pip install numpy
pip install paddlex -I https://mirror.baidu.com/pypi/simple
```

三、实验原理和内容

1. 图像预处理

图像预处理的主要目的是消除图像中无关的信息，恢复有用的真实信息，增强有关信息的可检测性和最大限度地简化数据，从而改进特征抽取、图像分割、匹配和识别的可靠性。本实验中，首先对摄像头画面进行左右翻转，然后用双边滤波器去除图像中的高斯白噪声。之所以使用双边滤波器，是因为高斯滤波、维纳滤波等降噪方法容易模糊图片的边缘细节，对于高频细节的保护效果并不明显。相比较而言，双边滤波器可以很好的边缘保护，即可以在去噪的同时，保护图像的边缘特性。另外，为了提高人手检测和手势识别的可靠性，我们还使用了 cv2.createBackgroundSubtractorMOG2 除去背景。

2. 人手检测以及二值化

针对在复杂环境背景下肤色、光照变化等对动态手势分割的干扰，结合帧差法和肤色椭圆模型的动态手势分割方法。该方法首先把视频的 RGB 颜色空间转为 YCbCr 颜色空间或 HSV 颜色空间，利用肤色检测初步确定肤色区域，然后结合帧间差分方法对手势进行检测并实现动态手势分割，以解决光照变化、类肤色背景等因素对手势分割的干扰问题。

A. 肤色分割（Skin Segmentation）

肤色是人类皮肤重要特征之一，在检测人脸或手等目标时常采用肤色检测的方法，将相关区域从图像中分割出来。肤色检测方法有很多，但无论是基于不同的色彩空间还是不同的肤色模型，其根本出发点在于肤色分布的聚集性，即肤色的颜色分量一般聚集在某个范围内。通过大量的肤色样本进行统计，找出肤色颜色分量的聚集范围或用特殊的分布模型去模拟肤色分布，进而实现对任意像素颜色的判别。

肤色模型是根据大量样本的统计数据建立以确定肤色的分布规律，进而判断像素

的色彩是否属于肤色或与肤色相似程度的模型。本实验中使用了阈值模型，是用数学表达式明确肤色分布范围的建模方法。这类方法依据肤色分布范围进行检测，判断简单、明确、快捷，但需要选择合适的颜色空间及合适的参数。

YCbCr 颜色空间是一种常用的肤色检测的色彩模型，其中 Y 代表亮度（为了消除光照的影响一般放弃亮度通道），Cr 代表光源中的红色分量，Cb 代表光源中的蓝色分量，人体的皮肤的颜色集中在色度的较小区域内。肤色的 CbCr 平面分布在近似椭圆的区域内，通过判断当前像素点的 CbCr 值是否落在肤色分布的椭圆区域内，就可以很容易确认当前像素点是否属于肤色。据资料显示，正常黄种人的 Cr 分量大约在 133 至 173 之间，Cb 分量大约在 77 至 127 之间。

在 HSV 颜色空间也是一种典型的肤色分割的色彩模型。此模型中颜色的参数分别是色调（H）、饱和度（S）和明度（V）。其肤色分割类似于基于 CbCr 空间模型的分割，进行一定的阈值处理后可得到很准确的肤色区域。据资料显示，正常黄种人的 H 分量大约在 0 至 20 之间，S 分量大约在 48 至 255 之间，V 分量大约在 80 至 255 之间。

两种基于不同颜色空间上的肤色分割均可选，其漏检率以及检测率都很高。通过肤色分割得到的二值化图像不仅用于人手跟踪，还会在手势识别中。基于 YCbCr 空间的肤色分割代码如下：

```
ycbcr = cv2.cvtColor(img,cv2.COLOR_BGR2YCR_CB) # 转换到 YCbCr 空间上
lower = np.array([0, 133, 77])
upper = np.array([255, 173, 127])
skin = cv2.inRange(ycbcr,lower,upper) # 范围筛选
```

B. 运动目标分割（Moving Object Segmentation）

运动目标分割算法主要包括帧差法和背景差法。

背景差法是将当前帧与背景图像进行差分来得到运动目标区域。通过不含运动的的帧来获取背景图像。常用的背景构建方法有时间平均法、单高斯法、混合高斯法等。但背景差分法结合肤色分割缺陷在于实验环境的实时性决定光照条件一直处于变动状态，背景差分中的背景更新问题较为复杂。因此，本实验中采取了帧差法。

帧差法是基于运动图像序列中相邻帧图像具有较强的相关性而提出的检测方法，不同帧对应的像素点相减，判断灰度差的绝对值，当绝对值超过一定阈值时，即可判断为运动目标，从而实现目标的检测功能。代码实现如下：

```
prevGray = cv2.cvtColor(prevFrame,CV_BGR2GRAY); # 灰度处理
currGray = cv2.cvtColor(currFrame,CV_BGR2GRAY);
diff = cv2.absdiff(prevGray,currGray); # 帧差
```

fdb = cv2.threshold(diff,25,255,CV_THRESH_BINARY); # 二值化

C. 定义人手区域

从肤色分割和运动目标分割可分别获取二值化图像 skin 和 fdb。我们结合了这两个二值化图像，对它们进行逻辑与运算得到二值化图像 SM，完成了动态手势分割。

$$SM = skin \,\&\, fdb$$

人手区域可以根据下面的公式被定义：

$$ROI = rectangle(SM, [m, n, w, h])$$

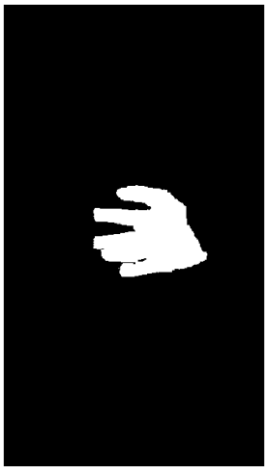
$$w = max[C] - min[C]; h = max[R] - min[R]; m = min[C]; n = min[R]$$

其中 C 和 R 分别是二值化图像 SM 中值为 1 的像素点的行坐标和列坐标集合。

D. 检测结果

图一为从电脑摄像头获取的原始图像；图二为对原始图像做预处理后进行的基于 CbCr 颜色空间检测获得的二值化图像；图三为基于帧差法的运动目标分割的结果；图四为对上述两个分割进行逻辑与运算后得到的二值化图像；图五和图六中红色边框为按照 ROI 定义公式获取的人手区域。

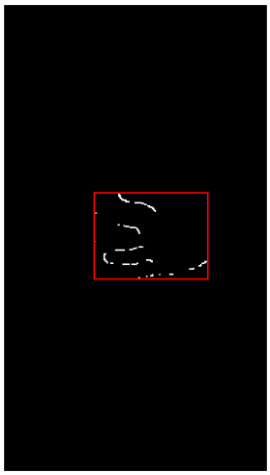
(1) original



(2) skin

(3) fdb

(4) SM



(5) ROI

(6) ROI+RGB

3. 基于自适应卡尔曼滤波器的人手跟踪

卡尔曼滤波器可以根据上一帧的人手区域的运动状态预测当前帧的运动状态（位置和速度）。然而，因为人手运动速度实时改变，一般是非线性的，所以具有固定参数的卡尔曼滤波器在很多情况下效果不太理想。这一问题可以由根据当前加速度实时改变系统噪声和观测噪声的协方差矩阵 Q 和 R 的方法来有效解决，而这就是自适应卡尔曼滤波器的原理。自适应卡尔曼滤波是指在利用测量数据进行滤波的同时，不断地由滤波本身去判断系统的动态是否有变化，对模型参数和噪声统计特性进行估计和修正，以改进滤波设计、缩小滤波的实际误差。

A. 系统建模

定义观测向量：

$$Z = [x, y]^T$$

定义被观测的状态向量：

$$X = [x, y, \dot{x}, \dot{y}]^T$$

其中x, y 分别表示人手在图像中的位置（行坐标和列坐标）；$\dot{x}, \dot{y}$表示其微分，即人手在第 k 帧时的相对速度。从而可得到t_k时刻被估计状态的状态方程和线性观测方程：

$$\begin{cases} X_k = AX_{k-1} + BU_{k-1} + w_{k-1} \\ Z_k = HX_k + v_k \end{cases}$$

其中，w_k和v_k分别是高斯白噪系统噪声和观测误差（真实状态与预测状态之间的误差），A是状态转移矩阵，H是观测矩阵。

$$\mathrm{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

根据下列卡尔曼滤波器的五个基本迭代公式，我们可以通过“预测”与“更新”两个过程来对系统的状态进行最优估计。

$$\hat{X}_k^- = A\hat{X}_{k-1} + BU_{k-1}$$

$$P_k^- = AP_{k-1}A^T + Q$$

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1}$$

$$\hat{X}_k = \hat{X}_k^- + K_k\left(Z_k - H\hat{X}_k^-\right)$$

$$P_k = (I - K_k H)P_k^-$$

B. 自适应卡尔曼滤波

根据求解卡尔曼滤波增益的公式：$K_k = \frac{P_k^- H^T}{HP_k^- H^T + R}$，可发现观测噪声的协方差矩阵 R 的大小会直接影响卡尔曼增益 K 值：R 越大，K 值越小，滤波结果更加接近于由系统

状态估计值给出的递归结果；R 越小，K 值越大，滤波结果更加接近于观测值所反算出来的状态变量。同理根据公式 $P_k^- = AP_{k-1}A^T + Q$ ，Q 越小，K 越大，滤波结果更加接近由系统状态估计值；Q 越大，K 越小，滤波结果更加接近于观测值。在上述构建的典型的卡尔曼滤波器中我们假设人手运动是线性的，但实际运动却不是这样的。因为人手运动时其速度的大小和方向经常改变，整个运动过程是一个非线性的，而根据当前加速度实时调整 Q 和 R 可解决这一问题。

首先根据下列公式求当前 x 和 y 方向的加速度，

$$a_x = \frac{\dot{x}_k - \dot{x}_{k-1}}{\Delta t};\ a_y = \frac{\dot{y}_k - \dot{y}_{k-1}}{\Delta t}$$

然后计算两个协方差矩阵 Q 和 R：

$$\mathrm{Q} = \omega_Q \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{R} = \omega_R \begin{bmatrix} f & \frac{f}{40} \\ \frac{f}{40} & \frac{f}{4} \end{bmatrix}$$

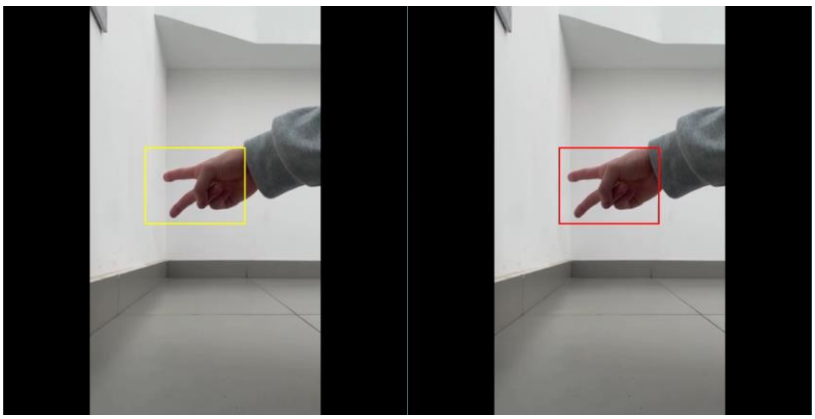
其中，f的值为 0.1845，是通过实验得到的[2]；ω_Q和ω_R分别是 Q 和 R 的权重，我们调整这两个权重来控制 Q 和 R 的大小，其调整方式如下：

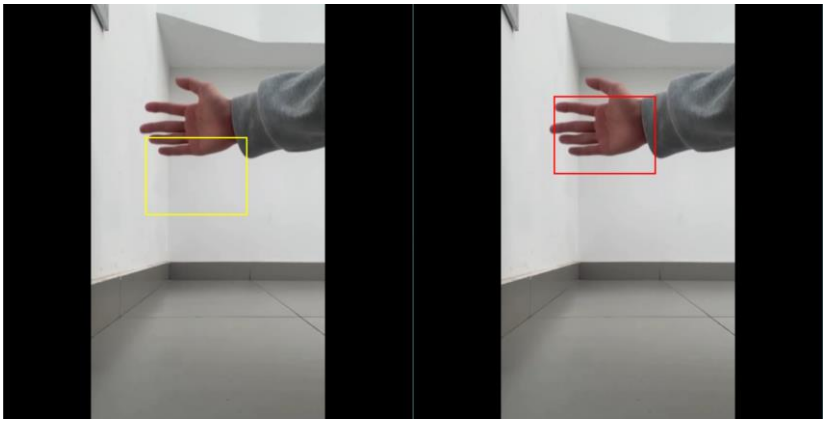
$$\omega_Q = \begin{cases} \frac{0.75}{\mathrm{T}_a}, |a_x| \ or \ |a_y| \geq \mathrm{T}_a \\ \frac{0.25}{\mathrm{T}_a}, else \end{cases}$$

$$\omega_R = \begin{cases} \frac{7.5}{{\mathrm{T}_a}^2}, |a_x| \ or \ |a_y| \geq \mathrm{T}_a \\ \frac{7.5}{\mathrm{T}_a}, else \end{cases}$$

也就是说，当手的当前加速度大于阈值T_a时，我们放大系统噪声的权重ω_Q而降低观测噪声的权重ω_R；否则，我们放小系统噪声的权重ω_Q而放大观测噪声的权重ω_R。

C. 实验结果与分析





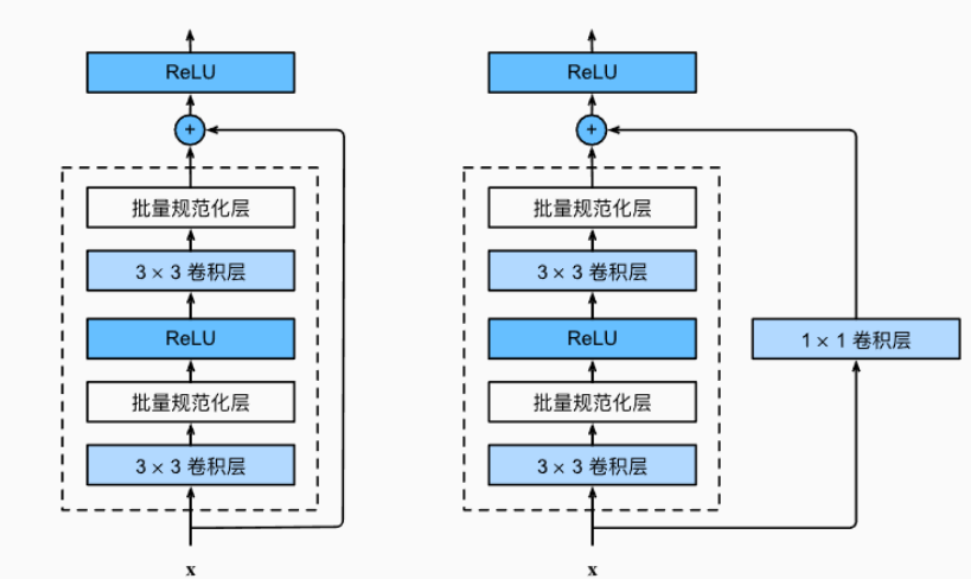
以上两个图中左侧为直接根据人手检测得到的人手区域，右侧为用自适应卡尔曼滤波器滤波后得到的人手区域。第一个图为人手做类似于匀速直线运动时的跟踪结果，第二个图为人手移动速度（大小以及方向）改变时的跟踪结果。可以发现，转换移动速度时，仅靠人手检测的定位误差比较大，跟踪结果与实际位置不一致，但卡尔曼滤波之后，有效地去除了人手检测错误所导致的误差，提高了人手跟踪的可靠性。

然而，我们最终放弃了人手跟踪部分。针对手臂、人脸等肤色非手势区域或类肤色区域由于微小的运动被误检为目标色块而被分割出来，对后续的跟踪和识别造成干扰的问题。也就是说，露出手臂的条件下，人手和手臂均被判定为人手区域，影响了人手检测与跟踪的可靠性。而我们没有找到同时满足时间复杂度和稳定性的，可以从包含人手和手臂的“人手区域”中只提取人手的算法，考虑到实时性，最后就放弃了人手跟踪。因此，为了避免摄像头拍到手臂造成干扰，在摄像头画面中绘制手势框作为手势识别区域，之后的处理都只对手势框中的部分进行（二值化以及手势识别）。

4. 手势识别

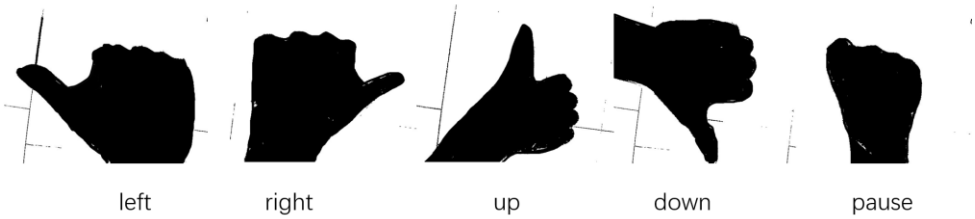
A. Resnet-18 模型

首先采用基于神经网络的手势识别法，使用的网络模型是经典的 Resnet-18 模型，使用残差块进行跨层数据线路传播，ResNet 沿用了 VGG 完整的 3×3 卷积层设计。残差块里首先有 2 个有相同输出通道数的 3×3 卷积层。 每个卷积层后接一个批量规范化层和 ReLU 激活函数。 然后我们通过跨层数据通路，跳过这 2 个卷积运算，将输入直接加在最后的 ReLU 激活函数前。这样的设计要求 2 个卷积层的输出与输入形状一样，从而使它们可以相加。如果想改变通道数，就需要引入一个额外的 1×1 卷积层来将输入变换成需要的形状后再做相加运算。如下图。



使用 resnet 进行图像识别的代码存放在./tools/中。训练集通过摄像头拍摄，统一尺寸进行上、下、左、右、与停止的分类，同时使用数据增强（旋转、对称）方式对数据集进行扩充，使用扩充后的数据集进行 resnet 训练。网络在测试集上的表现较好，但存在一定的过拟合现象，在实际配合手部位置识别时使用效果不佳。由于未寻找到包含我们需要种类的手势的手部数据集，所以没有使用其他数据集进行较好的训练。考虑到测试表现与实际使用情况，决定使用简单的模板匹配对手势进行识别，即在图片中进行模板的平移，对二者重合区域的相似度进行计算，得到在图像上所有平移区域的正确率矩阵。

实验中我们将人手检测中得到的二值化图像与预设好的尺寸稍小的模板进行识别比对。其中模板为较为理想的情况下经过拍摄获得，同时经过轮廓上的填充修补。预设好的模板同样分为上、下、左、右、停止五种类别（见下图），选择在全图中模板匹配值最大的类别作为判断，最终输出判断的位置种类。



B. 识别结果分析

在实际使用中效果较好，但该方法收到环境的干扰较大，如果背景颜色变化较大或背景颜色与手部颜色接近相同，则模板匹配的准确率会进一步下降。可能的解决方法是对各个方向种类进行不同条件下模板的扩充。

5. 游戏制作

A. 选题及创意介绍

我们的游戏继承了超级马里奥中的角色，敌人，金币等元素，同时增加了存档点等功能，并构建了一张内容丰富的地图。

马里奥

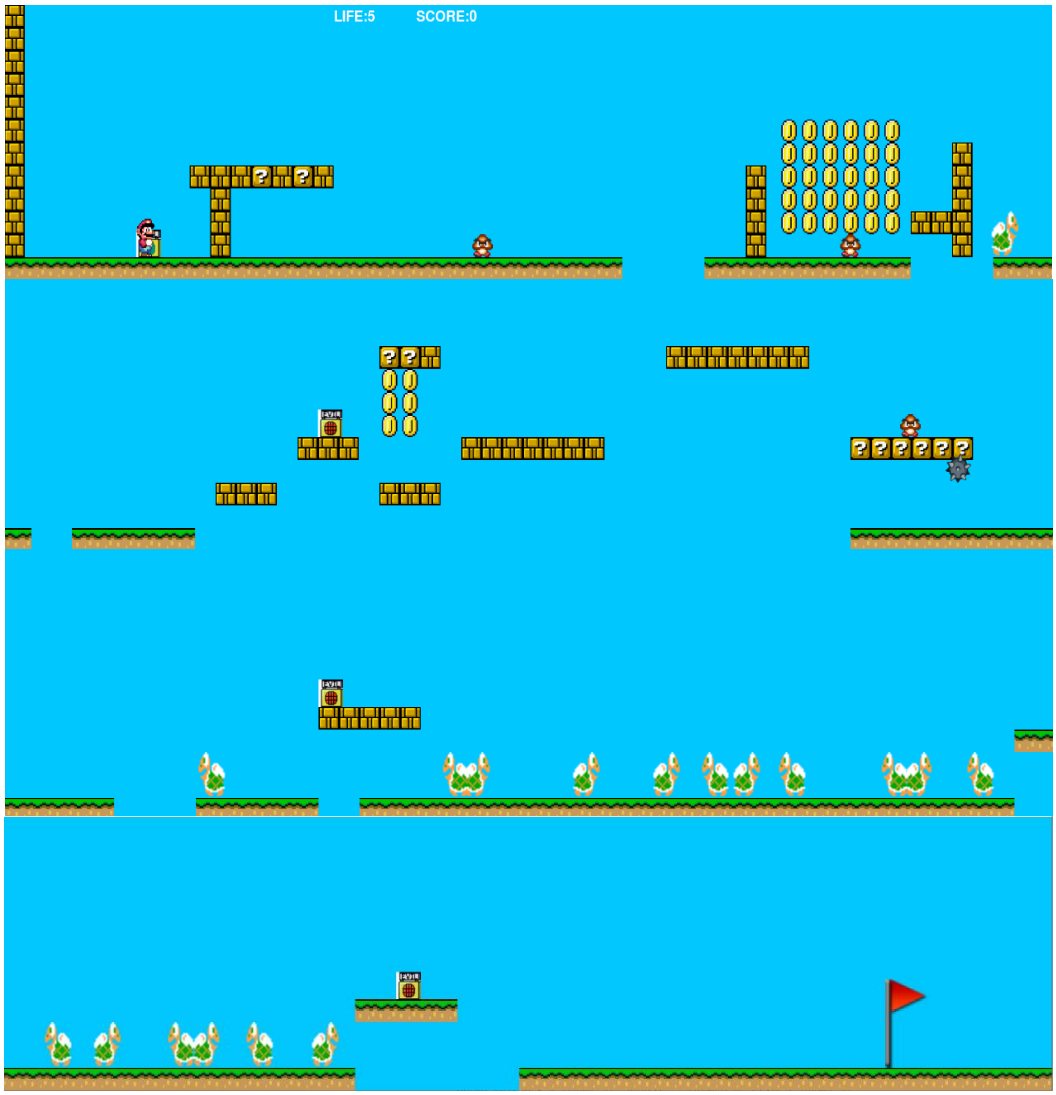
敌人

金币

炸弹

存档点

整体关卡设置由以下四张长图依次拼接而成：



B. 设计方案

Pgzero 模块用于绘制及更新游戏界面

Time 模块用于提供时间相关的函数

enemy 模块定义了 enemy 类，蘑菇怪、乌龟和炸弹等均属于 enemy 类，它可以控制 enemy 类的变量以设定的方式运动，并随时间或坐标变化改变它们的状态。

set 模块含有创造及重置地图内容的功能。

mario 模块创建了 mario 及 mario_die，并且能够控制它们的运动和动画效果。

判定模块用于处理 mario 和其他精灵在运动过程中发生的碰撞判定，并能在碰撞发生时改变 mario 及相关精灵的状态。

死亡、复活、胜利及重新开始模块，能够在 mario 死亡时清零分数，播放动画，并调用以上模块将 mario 复活至上一个存档点；当 mario 胜利或生命降低到 0 时，会出现相应文字，并允许玩家重新开始游戏。

主程序部分初始化全局变量，调用以上模块在屏幕上 draw 相应的内容并 update 各精灵的状态

C. 游戏框架及代码分析

我们的代码使用了如下的类和函数来实现游戏中的元素以及元素之间的联系。首先使用键盘进行控制，之后将按键操作重定义改为手势控制

0. 全局变量

```
last_action = 'pause' #上一轮动作
action = 'pause' #本轮动作
    vx = 0  # x速度
    vy = 0  # y速度
    maxv = 6  # 最大速度
    g = 0.5  # 重力加速度
    jump = 15  # 跳跃初速度
    walking = 0  # 是否在走路
    direction = "right"#方向
    mario_jump=False#马里奥是否在空中
    jumptimes=0#马里奥跳跃次数
```

```
gd = []  # 所有地面
gifts = []#所有礼物
enemies = []  # 小兵
traps = []  # 陷阱
flags = []#所有旗子（结束点）
saves = []#所有存档点

life = 5  # 命数
score = 0  # 得分
savepoint = -1
win = False#是否胜利
move_bg = 0#是否在移动背景
ground_now = 600
```

设置了一些参数，包括速度等参数，手势识别结果，记录游戏状态的一些变量以及存储精灵对象的列表

1. 马里奥的实现

使用 pygame zero 中自带的 Actor 类作为马里奥的类型，并加入以下函数来实现马里奥的移动与动画效果

马里奥的图片一共五张

1.1. walk()

```
def walk():
    if mario.image == "mario_big":
        mario.image = "mario_walk"
    elif mario.image == "mario_walk":
        mario.image = "mario_big"
    elif mario.image == "mario_walkleft":
        mario.image = "mario_left"
    elif mario.image == "mario_left":
        mario.image = "mario_walkleft"
```

通过改变马里奥的图片实现了马里奥的行走动画，当马里奥位置移动时，每间隔 0.1 秒调用，调用后马里奥的图像变为同方向行走的另一张分解动作

set_stop()

```
def set_stop():
    if direction == "right":
        mario.image = "mario_big"
    else:
        mario.image = "mario_left"
    clock.unschedule(walk)
```

停止了 walk 函数的调用并把马里奥恢复成站立状态

1.2. move_mario()

这个函数控制了马里奥的运动，当马里奥存活的时候，根据当前的按键状态决定马里奥的运动方向，并且根据马里奥是否落地，是否与其它背景块碰撞等一系列复杂的判定来调节运动状态。详见代码中的 move_mario()函数

```
if on_ground() and action == 'up':  # 只有落地可以跳跃
    vy = -jump
    mario_jump=True
    jumptimes+=1
    last_bottom = mario.bottom
    mario.bottom += vy
    clock.schedule_unique(set_mario_normal,1)
```

其中跳跃操作也插入在此函数中，只有马里奥在地面上且手势为向上时才可以跳跃。

1.3. check_underground()

由于刷新频率过低可能出现一帧之后马里奥的下部直接掉到地面以下，这个函数检验了马里奥是否在地面以下，如果为真则后续会上调马里奥位置至地面

1.4. get_gift()

```
def get_gift():
    global score,life
    for i in gifts:
        if i.image == "block1":
            if (
                mario.colliderect(i) and i.top < mario.top <= i.bottom < mario.bottom
            ):  # 只有从下面接触才会产生金币
                i.image = "coin"
                i.top -= 50
        elif i.image == "coin":
            if mario.colliderect(i):
                score += 1
                if score>=50 and score%50==0:
                    life+=1
                gifts.remove(i)
```

控制马里奥吃金币，共须两步，首先把碰撞礼物变出金币，第二步碰撞金币获取得分。每得到 50 分可以额外获得一条命

2. 敌人的实现

游戏中的敌人有小兵，乌龟，炸弹三种不同种类，但我们使用了同一个 enemy 类来进行封装，通过 image 的不同以及游戏中的分组来区别敌人的种类

class enemy(Actor)

```
def __init__(self, **kwargs):
    super().__init__(image="enemy1", **kwargs)
    self.leftlimit = 0
    self.rightlimit = 2000
    self.uplimit = 100
    self.downlimit = 500
    self.v = 3
    self.use = True
    self.direction = "left"
    self.vy = 0
```

Enemy 继承自 Actor 类，新增的变量有敌人移动范围（四个 limit），移动速度（v,vy），移动方向，与是否显示（use）,在创建对象时可以进行设定

```
if self.direction == "left":
    if self.left >= self.leftlimit + self.v:
        self.left -= self.v
    else:
        if self.image == "turtle_left":
            self.image = "turtle_right"
        self.direction = "right"
```

enemy 类中的 update 函数主要实现里敌人在活动区间内的移动，到达区间终点则改变移动方向。图为左移代码，其他方向同理

2.1. 小兵

是 enemy 类的基础，可以左右进行移动，移动代码如上图所示。

当马里奥从上部踩小兵时，小兵会死亡，其他方向碰撞马里奥会死亡

```
elif mario.colliderect(i):
    if mario.bottom <= i.top + 15:
        if i.image == "enemy1":
            # 敌人死亡
            i.image = "enemy3"
            score += 1
```

```
else:
    die()
```

2.2. 乌龟

其他与小兵类似，但踩死乌龟需要两次。第一次踩乌龟后触发复活函数，如果五秒内不踩第二次则会调用复活函数

```
elif i.image == "turtle_left" or i.image == "turtle_right":
    i.image = "turtle_hide"
    v0 = i.v
    i.v = 0
    vy = -15
    mario.bottom-=15
    clock.schedule_unique(i.turtle_recover, 5)
elif i.image == "turtle_hide":
    vy = -15
    i.image = "turtle_died"
    mario.bottom-=15
    score += 2
```

```
def turtle_recover(self):
    if self.image=='turtle_hide':
        self.image='turtle_left'
        self.v=3
```

2.3. 炸弹

分为上下移动与左右移动两种，移动代码与小兵和乌龟相同。炸弹在触发之前不可见，须把 use 设置为 false，并用 nearby 函数检测马里奥是否接近炸弹，若接近则把 use 设置为 true 使其可见

```
def nearby():
    global mario
    for i in traps:
        if i.image == "bomb1" and mario.distance_to(i) <= 150 and i.use == False:
            i.v = 3
            i.direction = "up"
            i.use = True
```

3. 碰撞判定

碰撞判定一共四个函数，分别为：

on_ground()

top_collide()

left_collide()

right_collide()

分别检测上下左右是否与地面或者墙面碰撞，以左部碰撞检测为例

```
def left_collide():  # 判断左边碰撞
    global gd, mario
    for i in gd:
        if (
            mario.colliderect(i)
            and i.left < mario.left <= i.right < mario.right
            and not last_bottom <= i.top + 1
        ):
            mario.left += 7
            return True
    return False
```

其中使用背景块与马里奥的两个边界同时进行比较是为了避免出现在马里奥整体在左方却判断左部碰撞的问题，并增加了重叠立即弹开的效果以确保马里奥不会穿墙

4. 场景 set

一共使用七个函数，将场景，礼物，敌人，炸弹，旗子与存档点的设定分别封装到六个函数里，在初始化与重新开始时调用。代码内容为全部为精灵对象与 enemy 对象的初始化，在此不一一列出

set_bg()

set_gift(savepoint_x=320)

set_enemy(savepoint_x=320)

set_trap(savepoint_x=320)

set_flag()

set_save()

最后使用 set_all()调用以上全部函数并加入音乐的播放，简化了场景设置的过程

5. 死亡与复活

5.1. 死亡

死亡分为两种方式，第一种为碰到敌人，会调用 hurt 函数

马里奥碰到 enemy 从而使自己或者 enemy 受到伤害，具体伤害须经过碰撞方向判定，enemy 类型判定等一系列复杂判定，若碰到炸弹或正面遭遇敌人会触发死亡

第二种为跌落

```
def fall():  # 失足坠落
    global life
    if mario.bottom >= 700 and life > 0:
        die()
```

无论哪种死亡方式都会调用 die（）函数并播放死亡动画，停止其他一切动作并把马里奥切换成死亡图像

```
def die():  # 死亡
    global vx, vy, vy_die
    mario.image = "mario_die"
    mario_die.bottomleft = mario.left, mario.bottom
    a = gd[0].left
    mario.bottomleft = a - 1200, 550
    vy_die = -20
    vx = 0
    vy = 0
    stop_move()
    walking = 0
    clock.unschedule(walk)
    sounds.theme.stop()
    sounds.die.play()
```

```
def death_animation():  # 死亡动画
    global vy_die
    if life > 0:
        vy_die += 0.5
        mario_die.bottom += vy_die
        if mario_die.bottom >= 700:
            refresh()
            return
```

当马里奥跌落至屏幕以下 100 像素且还有复活机会时，将启动复活

5.2.复活

当马里奥死亡但仍有复活机会时，会调用复活函数，其中 savepoint 使复活之后马里奥回到上一个保存点，而不是一定回到出发点。

```
def refresh():  # 复活
    global life, vx, vy, vy_die, score, savepoint
    life -= 1
    if life > 0:
        vx = 0
        vy = 0
        vy_die = 0
        distance_back = 320 - saves[savepoint].left
        savepoint_height = saves[savepoint].bottom
        savepoint_x = saves[savepoint].left - gd[0].left
        for i in gd + flags + saves:
            i.left += distance_back
        score = 0
        walking = 0
        reset_lives(savepoint_x)
        mario.image = "mario_big"
        mario.bottomleft = 320, savepoint_height
        mario_die.bottomleft = gd[0].left - 1200, 550
        sounds.theme.play(-1)
    else:
        sounds.fail.play()
```

reset_lives(savepoint_x)

复活之后重新设置可活动的对象，包括所有小兵，乌龟，实现方法为全部清除并调用 set 函数

6.背景滚动

move_all():

```
if mario.left > 900 and mario.image != "mario_die":
    move_bg = 1
    clock.schedule(stop_move, 1)
elif mario.left < 300 and mario.image != "mario_die":
    move_bg = 2
    clock.schedule(stop_move, 1)
if move_bg == 1:
    mario.left -= 10
    for i in gd + gifts + enemies + traps + flags + saves:
        i.left -= 10
    for i in enemies + traps:
        i.leftlimit -= 10
        i.rightlimit -= 10
```

当马里奥到达边缘时触发。用全局变量 move_bg 标记移动方向通过移动所有精灵对象达到屏幕滚动的效果

stop_move()函数将 move_bg 重新设置为 0，从而达到停止移动的效果

7.失败与胜利

```
def game_over():
    clear_all()
    screen.blit(images.trap_card, (200, 30))
    screen.draw.text("GAME OVER!", (400, 300), color="red", fontsize=96)
    screen.draw.text("Press R to RESTART", (950, 10), color="white", fontsize=36)
    if keyboard.r:
        reset_all()
```

失败与胜利均会绘制失败或胜利图像并提示重新开始，当按下 r 键则调用 reset_all()，重新调用初始化部分,开始新游戏

D. 游戏的控制

在 update 函数中插入手势识别代码，不断循环调用识别当前手势并使用全局变量 action 记录当前识别出的手势。

```
    max_type = 4
    max_all = 0
    for i in range(5):
        res = cv2.matchTemplate(skin, template[i], cv2.TM_CCOEFF_NORMED)
        min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(res)
        if max_val > max_all:
            max_all = max_val
            max_type = i
    last_action = action
    action = directions[max_type]
    if action == 'down':
        action = 'pause'
    print(action)
```

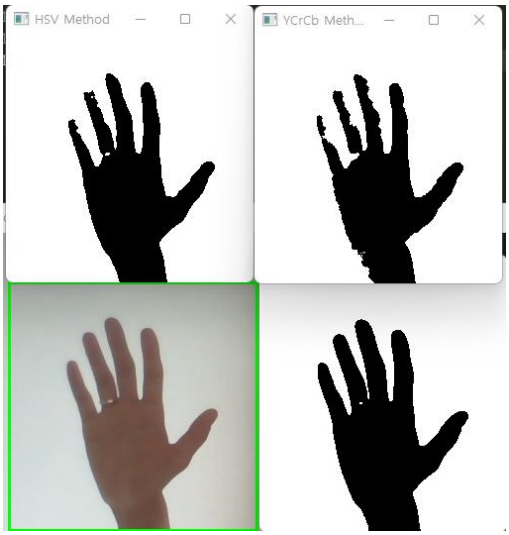
重新定义 pgzero 模块中的 on_key_up 与 on_key_down 函数的触发方式，改为当手势识别为向左时触发原来的按下左键操作，手势识别为向右时触发原来的按下右键操作，上一循环手势识别为左而本轮不为左时触发原来的抬起左键操作，上一循环手势识别为右本轮循环不为右时触发原来的抬起右键操作。

```
if last_action != action:
    if last_action == 'left':
        on_key_up(keys.LEFT)
    if last_action == 'right':
        on_key_up(keys.RIGHT)
if action == 'left':
    on_key_down(keys.LEFT)
if action == 'right':
    on_key_down(keys.RIGHT)
```

四、实验结果

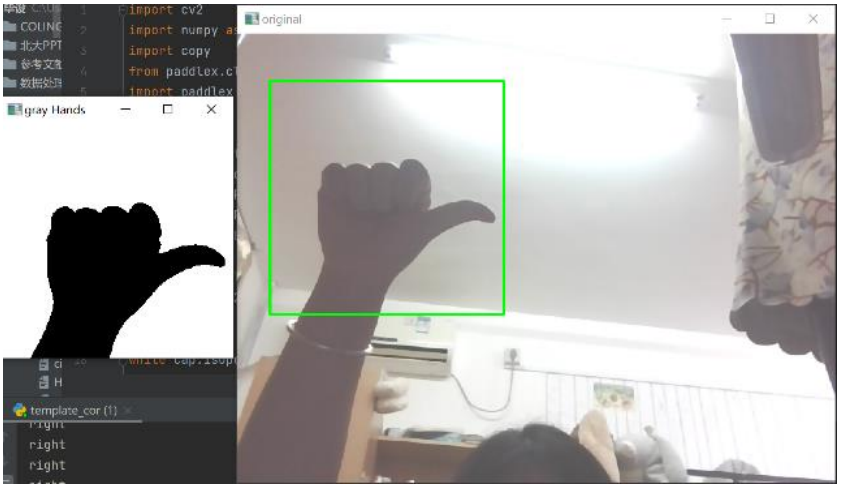
1. 人手检测与二值化

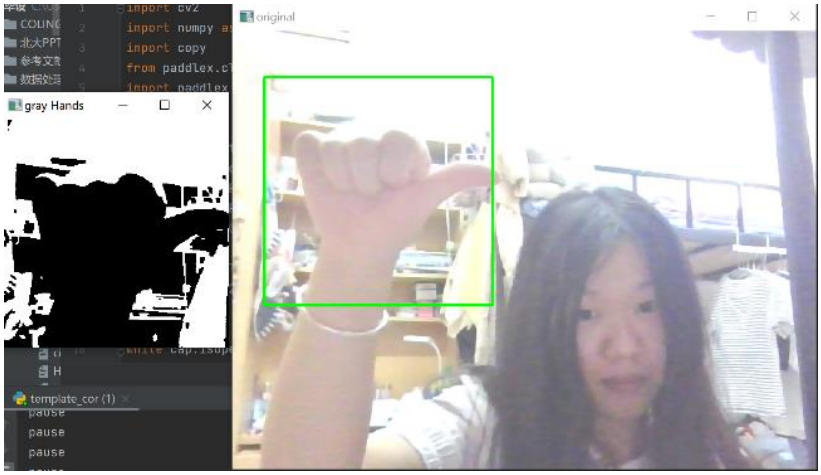
在单色背景中，基于 HSV 和 YCbCr 颜色空间的人手检测以及 Otsu 图像分割的二值化结果都比较合理，且 Otsu 算法检测结果最为平滑。因此，背景比较单一时可以使用 Otsu 算法来进行人手检测，而在比较复杂的环境中使用基于颜色空间模型的肤色分割算法来进行二值化是更合理的。



2. 手势识别

当手势框中的背景较为简单时，手势识别的准确率较高，但手势框中背景复杂且接近肤色时，手势识别准确率较低。





考虑到手势识别需要一定的时间，我们把游戏的速度调整的比较低。我们录制了完整游戏的视频，可以看到当手势框中的背景为白墙且游戏速度设置较慢时，手势识别可以很好的控制马里奥的运动，并顺利通过关卡。视频详见附件中的 supermario.mp4

五、参考文献

[1] 张忠军.基于肤色椭圆边界模型和改进帧差分算法的手势分割方法.福建电脑.2018,34(05)

[2] Mohd Shahrimie，Shahrel Azmin Suandi. Hand Gesture Tracking System Using Adaptive Kalman Filter. (2010)

六、小组分工

金镇雄：图像预处理、人手检测与跟踪、PPT 和书面报告整合

姜恒：手势识别

王悦：游戏制作、实验结果