浙江大学

课程名称：计算机动画

姓　　名：猜猜

学　　院：计算机学院

专　　业：数字媒体技术

学　　号：

指导教师：

2020 年 1 月 6 日

浙江大学实验报告

课程名称：计算机动画 实验类型：综合

实验项目名称：群组动画

学生姓名： 专业：数字媒体技术 学号：

同组学生姓名：无 指导老师：

实验地点：玉泉 实验日期：2020 年 1 月 6 日

一、 实验目的和要求

1.掌握相互速度障碍物避障法（RVO 算法）的思路和算法。

2.掌握并实践相互速度障碍物避障法（RVO 算法）的具体思路和程序代码的对应关系。

3.了解群组动画的概念，了解生成一组群组动画的基本原理和方法，提高相关动画编程能力。

二、 实验内容和原理

实验内容：

实现群组动画（相互速度障碍物避障），给每一个点定义其初始位置和最终位置，实现每一个点从初始点到达终点的效果，在点运动的过程中，需要实现点与点之间的避障，并满足躲避静态障碍物。

实验原理：

速度障碍物算法（VO 算法）的具体解释如下（如图 Fig-1 所示）假设场景中有两个个体 A，B，P_A 和 P_B 分别代表其位置，V_A 和 V_B 分别代表其速度。个体 A，B 把彼此当作具有一定速度的障碍物，根据位姿空间理论，个体 A 可以被缩小为一个点，个体 B 可以被扩大为半径为 R_A+R_B 的圆，则 A 相对于 B 的速

度为 V_A-V_B，其中浅灰色的三角锥部分即为碰撞圆锥，如果相对速度在浅灰色区域内，则个体 A 就会与个体 B 相撞，如果相对速度不在该浅灰色区域内，则物体 A 与物体 B 不会发生碰撞。该算法实现了个体之间的避碰，但是由于下一时刻速度的选择具有随意性，可能会导致新旧速度方向不断变化，出现振荡现象，从而影响仿真的真实性。

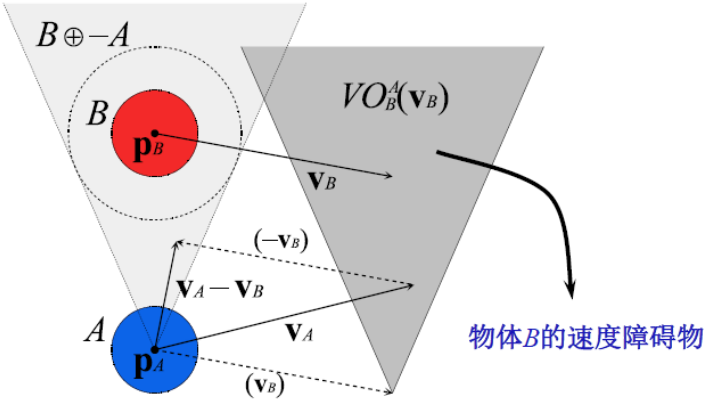


Fig-1

相对速度避障法（RVO）不仅可以解决个体与个体之间的避碰，还可以实现个体与障碍物之间的避碰（如图 Fig-2 所示），该算法中 A 相对于 B 的速度为 V_A `=V_A-1/2(V_A+V_B)，算法的原理相当于个体 A 和个体 B 都避开彼此的速度障碍物，如果相对速度在浅灰色区域内，则个体 A 就会与个体 B 相撞，如果相对速度不在该浅灰色区域内，则物体 A 与物体 B 不会发生碰撞，该算法减少了振荡的发生，使路线更真实。

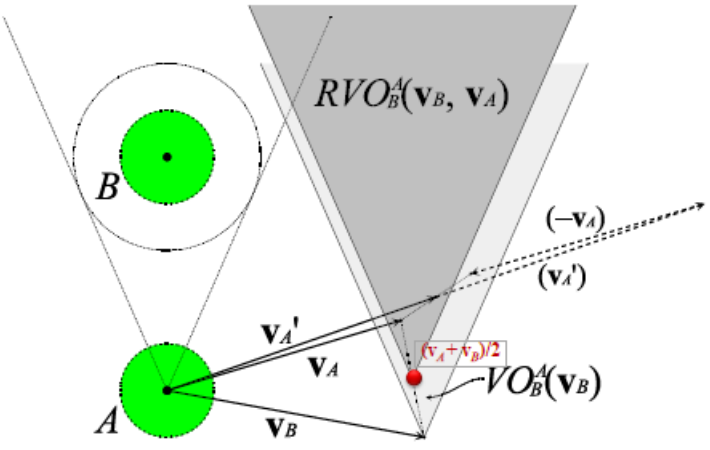


Fig-2

三、 实验器材

实验编译运行平台：

Windows10

Pycharm（Python 3.7.4）

OPENCV 3.4.2

实验工具库

matplotlib

四、 实验步骤

（一）整体框架

1.首先规定好四种模式提供给用户选择，第一种模式为无障碍物模式的简单的左右两列（每列 10 个点）进行位置互换（如图 Fig-3）；第二种模式为无障碍物模式的一个圆上的 36 个点进行与对面位置的点的位置互换（如图 Fig-4）；第三种模式为有方形障碍物模式的位置互换（如图 Fig-5），第四种模式为有圆形障碍物模式的位置互换（如图 Fig-6）。

Fig-3

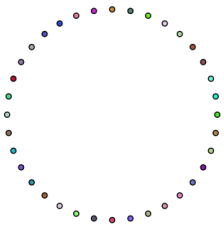


Fig-4

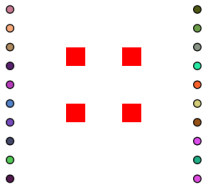


Fig-5

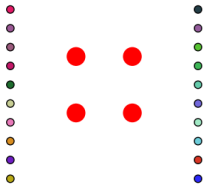


Fig-6

2.通过用户指定的模式选择对应的模式。首先需要定义的有关于个体：个体的当前位置（初始化为初始位置）、目标位置、每个个体的半径、每个个体的当前速度（初始化为 0），每个个体的最大速度、个体的活动范围；关于障碍物：障碍物的类型（矩形或圆形），障碍物的半径（或长的 1/2 和宽的 1/2），障碍物的位置。

3.定义停止运动条件：当所有的点距离目标位置的距离都小于某一距离时，停止运动，这里我设定的阈值为 0.1，具体代码如下：

```
# 给定所有点当前的坐标和最终的坐标，判断是否所有点全部到达
def isArrived(current_position, goal_position):
    for i in range(len(current_position)):
        distance = getDistance(current_position[i], goal_position[i])
        if distance > 0.1:
            return False
    return True
```

4.当还在运动状态下时，首先计算首选的速度，之后选择新的速度，并根据新的速度更新个体的位置，并在一定时间步长后输出当前位置的图片，最后生成视频。具体代码如下：

```
while not isArrived(robot['current_position'], robot['goal_position']):
    pref_velocity = getPrefVelocity(robot)
    new_velocity = getNewVelocity(robot, obstacle, pref_velocity)
    t += 1
    for i in range(len(robot['current_position'])):
        robot['current_position'][i][0] += new_velocity[i][0] * 0.04
        robot['current_position'][i][1] += new_velocity[i][1] * 0.04
        drawProcess(size, robot, obstacle, color, new_velocity, t)
```

接下来将展示具体算法部分

（一）具体算法

1.计算首选速度。

计算现在所在位置与目标位置之间的距离和相差的横坐标、纵坐标，接下来计算首选速度，当现在所在位置已经接近终点（即距离目标位置的距离小于 0.1）时，首选速度为 0；当尚未接近终点时，首选速度采用横纵坐标之差×最大速度÷距离来表示。计算出来所有点的首选速度列表为 pref_v。具

体代码示例如下：

```
def getPrefVelocity(robot):
    pref_v = []  # 首选速度
    for i in range(len(robot['current_position'])):
        distance = getDistance(robot['current_position'][i], robot['goal_position'][i])  # 现在所在位置与目标位置之间的距离
        difference_x = robot['goal_position'][i][0] - robot['current_position'][i][0]  # 现在所在位置与目标位置之间横坐标的差
        difference_y = robot['goal_position'][i][1] - robot['current_position'][i][1]  # 现在所在位置与目标位置之间纵坐标的差
        if (distance <= 0.1):  # 已经接近终点
            pref_v.append([0.0, 0.0])
        else:  # 尚未接近终点
            difference = [difference_x, difference_y]  # 现在所在位置与目标位置之间横坐标、纵坐标的差的列表表示
            pref_v_x = difference[0] * robot['max_velocity'][i] / distance
            pref_v_y = difference[1] * robot['max_velocity'][i] / distance
            pref_v.append([pref_v_x, pref_v_y])
    return pref_v
```

2.选择新速度

遍历所有的点，选出所有点的新速度（这里把当前点称为点 A），首先算出点 A 的半径、当前位置和当前速度，然后考虑除本身外所有点和障碍物对其的影响。将除本身外的点和障碍物视作 B，计算 B 的半径、当前位置和当前速度，计算 A 与 B 当前的关系。

计算 A 与 B 当前的关系算法如下：计算 AB 之间的距离，计算 AB 圆心的连线与水平线之间的角度大小，如果 AB 之间的距离小于其半径之和，则用半径之和代替距离，计算切线与 AB 圆心连线的夹角，计算三角锥的左边界和右边界的角度，上面所说定义如下图所示（Fig-7）

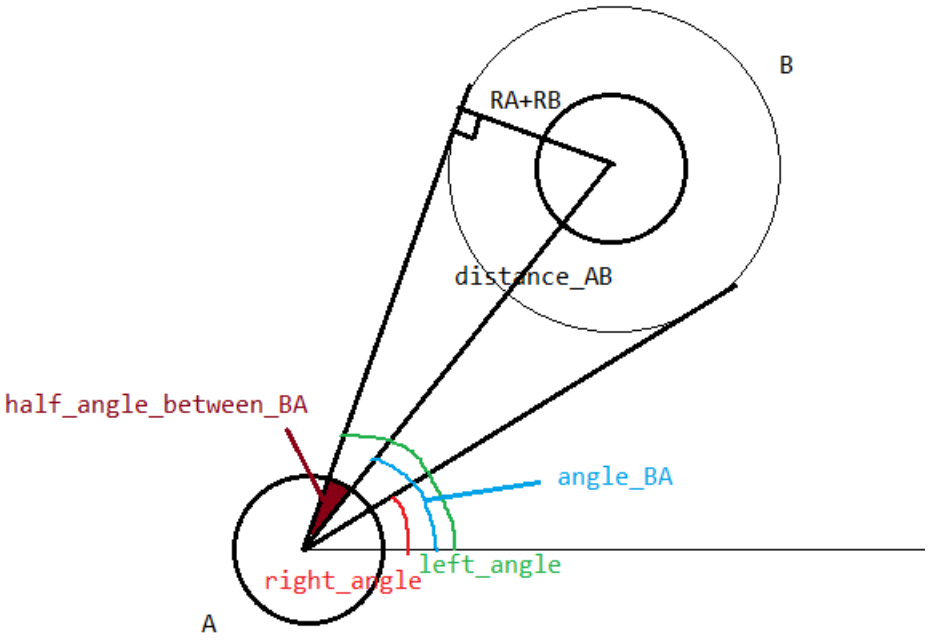


Fig-7

得到 AB 之间的关系具体代码示例如下：

```
def getRela(position_A, position_B, radius_A, radius_B, v_A, v_B):

    distance_BA = getDistance(position_A, position_B)  # AB 之间的距离
    angle_BA = math.atan2(position_B[1] - position_A[1], position_B[0] - pos
ition_A[0])  # AB 圆心的连线与水平线之间的角度大小

    # 如果 AB 之间的距离小于其半径之和，则用半径之和代替距离
    if distance_BA < radius_A + radius_B:
        distance_BA = radius_A + radius_B
    half_angle_between_BA = math.asin((radius_B + radius_A) / distance_BA)
# 切线与两线连线的夹角
    # 三角锥的左边界
    left_angle = angle_BA + half_angle_between_BA
    left_angle = math.atan2(math.sin(left_angle), math.cos(left_angle))  #
使角度一定处于 atan2 所允许的范围内（-pi，pi）
    # 三角锥的右边界
    right_angle = angle_BA - half_angle_between_BA
    right_angle = math.atan2(math.sin(right_angle), math.cos(right_angle))
# 使角度一定处于 atan2 所允许的范围内（-pi，pi）
    return left_angle, right_angle, radius_A + radius_B, distance_BA
```

得到 AB 之间的关系后，我们需要根据 AB 间的关系（主要是根据左右边界的角度）来选择新的速度。

选择新的速度的具体算法如下：首先算出从 0 到首选速度（常数大小）的五个插值的各个角度的速度，测试其是否符合要求，最后测试首选速度是否符合要求，如果符合要求则将其列入符合要求列表，如果不符合要求则舍去。测试其是否符合的标准如下：即测试新的速度减去 AB 当前速度的和的 1/2 所得到的速度（RVO 算法）的角度是否在上一步所得到的 AB 的左右夹角之间，如果其在左右夹角之间，说明会发生碰撞，则不符合要求，如果不在 AB 的左右夹角之间，则说明不会发生碰撞，符合要求，保留该新速度。当整个过程结束后，查看符合要求的速度列表，如果不为空，则选择列表中与首选速度最接近的新速度，如果为空则定义新速度为 0。选择新速度的具体代码如下：

```
def selectVelocity(position_A, pref_vA, rel_BA):
    suit_v = []  # 符合要求的速度列表
    pref_vA_cons = getDistance(pref_vA, [0.0, 0.0])  # A 首选速度的常数值
    # print(pref_vA_cons)

    # 首先测试所有的中间插值
    for angle in np.arange(0, 2 * math.pi, 0.1):
        for v in np.arange(0.02, pref_vA_cons + 0.02, pref_vA_cons / 5.0):
            new_v = [v * math.cos(angle), v * math.sin(angle)]
            # print(new_v)
            isSuit = judgeSuit(new_v, position_A, rel_BA)
            if isSuit:
                suit_v.append(new_v)
    # 测试首选速度
    new_v = pref_vA
    isSuit = judgeSuit(new_v, position_A, rel_BA)
    if isSuit:
        suit_v.append(new_v)

    # print('suit_v', suit_v)
    # 处理符合要求的速度和不符合要求的速度
    if suit_v:
        new_v = min(suit_v, key=lambda v: getDistance(v, pref_vA))
    else:
        new_v = [0.0, 0.0]

    return new_v
```

3.得到新的速度之后，根据时间步长计算新的位置，并在一定时间步长后输出当前位置的图片。

绘制图片这里采用了 matplotlib 库，从 matplotlib 导入 pyplot，首先通过 pyplot.figure()函数建立窗口 figure，之后再调用 add_subplot()函数添加子窗口，为了图片的整洁和美观，通过调用 pyplot.axis('off')隐藏坐标轴。之后开始绘制障碍物，通过调用 matplotlib.patches.Rectangle(self, xy, width, height, angle=0.0, **kwargs)函数来绘制方形的矩形，其中 xy 表示矩形的起点坐标，width 表示矩形的宽，height 表示矩形的高，这里我们给矩形填充为红色。通过调用 matplotlib.patches.Circle(self, xy, radius=5, **kwargs)函数来绘制圆形，其中 xy 表示圆心的坐标，radius 表示圆的半径，圆的填充颜色是随机得到颜色来绘制的。之后需要设置 x 轴和 y 轴的范围，使用的是 set_xlim()函数和 set_ylim()函数，之后保存图片到定义的路径中。

其中，对画布的处理和对矩形和圆的绘制的具体代码如下：

```
1.  def drawProcess(size,robot, obstacle, color, new_velocity, time):
2.      figure = pyplot.figure() #建立窗口
3.      ax = figure.add_subplot(1, 1, 1) #添加子窗口
4.      pyplot.axis('off') #隐藏坐标轴
5.      if obstacle: #如果存在障碍物
6.          if obstacle['type'] == 'rectangle':
7.              for i in range(len(obstacle['position'])):
8.                  ob = matplotlib.patches.Rectangle(
9.                      (obstacle['position'][i][0] - obstacle['radius'][i][0],
    obstacle['position'][i][1] - obstacle['radius'][i][1]),
10.                     2 * obstacle['radius'][i][0], 2 * obstacle['radius'][i][
    1],
11.                     facecolor='red',
12.                     fill=True,
13.                     alpha=1)
14.                 ax.add_patch(ob)
15.         if obstacle['type'] == 'circle':
16.             for i in range(len(obstacle['position'])):
17.                 ob = matplotlib.patches.Circle(
18.                     (obstacle['position'][i][0], obstacle['position'][i][1])
    ,
19.                     radius=obstacle['radius'][i][0],
20.                     facecolor='red',
21.                     fill=True,
22.                     alpha=1)
23.                 ax.add_patch(ob)
24.     for i in range(len(robot['current_position'])):
25.         r = matplotlib.patches.Circle(
```

```
            (robot['current_position'][i][0], robot['current_position'][i][1
]),
            radius=robot['radius'][0],
            facecolor=color[i],
            edgecolor='black',
            linewidth=1.0,
            ls='solid',
            alpha=1,
            zorder=2)
        ax.add_patch(r)

    ax.set_aspect('equal')
    ax.set_xlim(size[0], size[1])
    ax.set_ylim(size[0], size[1])

    str_time = str(time)
    time_final='{0:0>3}'.format(str_time)
    # './image/'+
    pyplot.savefig(time_final+'.png')
    pyplot.cla()
    pyplot.close(figure)
    return figure
```

随机获取颜色的代码如下：

```
#获取随机颜色
def getColor(N):
    color_arr = ['1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C',
 'D', 'E', 'F']
    color = []
    for i in range(N):
        c = '#'
        for i in range(6):
            c += color_arr[random.randint(0, 14)]
        color.append(c)
    return color
```

4.根据刚刚生成的图片制作视频。

生成视频需要用到 cv2.VideoWriter(filename, fourcc, fps, frameSize[, isColor])函数，其中 filename 为保存的文件的路径，这里我选择保存在 video 的文件夹下，fourcc 为编码器，其本身是一个 32 位的无符号数值，用 4 个字母表示采用的编码器，这里采用“XVID”，fps 为要保

存的视频的帧率，这里设为 25，frameSize 为要保存的视频的画面尺寸，这里设为（640，480），isColor 指示是黑白画面还是彩色画面。综上这里所运用的实例为

```
1.  out = cv2.VideoWriter(video_name, cv2.VideoWriter_fourcc(*'XVID'), fps,(640,
     480))
```

以初始化生成的视频。

之后读取给定路径中的文件名列表，读取每一张图片，将其写入视频中，即可得到完整视频。

```
1.  image_list = os.listdir(path)
2.  for i in range(len(image_list)):
3.      image = cv2.imread(path+image_list[i])
4.      out.write(image)
5.  cv2.destroyAllWindows()
```

五、 实验结果展示与分析（详见 result 文件夹下的图片和视频）

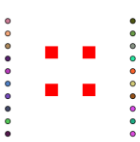
1.无障碍物，左右两列（每列 10 个点）进行位置互换。

初始状态　　中间状态　　结束状态

2.无障碍物，一个圆上的 36 个点进行与对面位置的点的位置互换

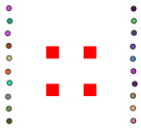
初始状态　　中间状态　　结束状态

3.有障碍物模式（矩形）的位置互换



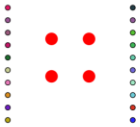
初始状态

中间状态



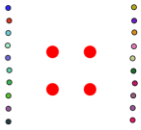
结束状态

4.有障碍物模式（圆形）的位置互换



初始状态

中间状态



结束状态

结果分析：因为在选择新速度时有对于速度比较全面的筛选和对于角度的严格判断，个体与个体的规避、个体与静态障碍物的规避在实验的结果中有比较好的体现，绝大多数个体可以有效地避碰。

但是在第二个样例接近末尾的时候我们可以看到（如图 Fig-8），两个小球重叠到了一起，同时在实验调试过程中我发现，在计算具体位置和速度的过程中，由于计算步骤的不同，有时会发生结果（float 类型）的细微差别，这可能是导致个体偶尔会发生重叠的原因。

Fig-8

另外，在实验中途做测试的过程中，会偶尔出现在障碍物附近个体卡住一直到最后都回不去的情况，这种情况很少发生，应该是因为被障碍物挡住，一直无法得到合适的速度导致的，这种情况值得注意，可以作为之后改进的一个方向。

六、 实验感想与心得

在实验过程中遇到了各式各样的问题，其中包括手误写错代码导致找了很长时间的 bug，以及遇到的几个算法上的问题，以下一一介绍：

第一个问题出现在计算首选速度上，因为不知道应该如何定义首选速度，最开始想到的就是用最大速度代替，但是方向当时并没有思路去解决，最后想到可以用初始位置到目标位置的方向来代替，于是首选速度即大小为最大速度，首选速度方向为初始位置到目标位置的方向。

第二个问题出现在判断新速度是否符合条件时，因为没有考虑下图（Fig-9）这种情况，即 left_angle 小于 0，right_angle 大于 0 的情况（所有的角度的范围为(-Π,Π)），于是导致在有些时候的速度一直很奇怪，经常会胡乱摆动。

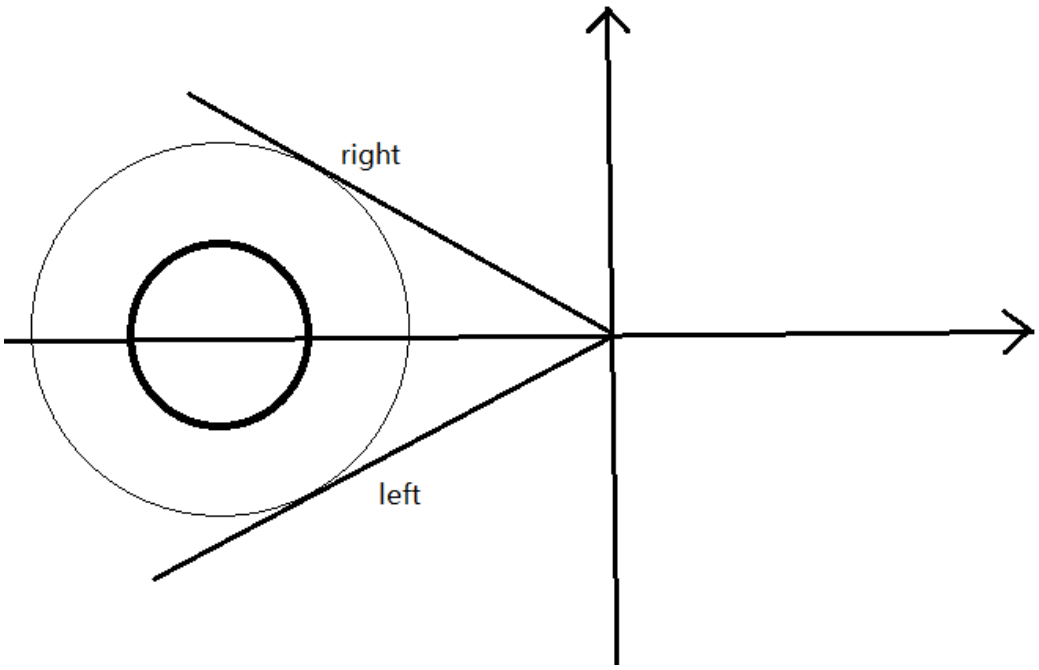


Fig-9

第三个问题出现在绘制图片上，刚开始打算用 Opencv 来绘制图片，但是用 Opencv 绘制的话就必须在整数值的像素点上进行绘制，这样的话会有一定的偏差，而且绘制起来也很不方便，需要经常进行数据格式的转变，后来发现 matplotlib 可以直接绘制，带来了很大的方便。

通过本次实验，我深入掌握了相互速度障碍物避障法（RVO 算法）的思路和算法，掌握并实践了相互速度障碍物避障法（RVO 算法）的具体思路和程序代码的对应关系，了解了群组动画的概念，了解了生成一组群组动画的基本原理和方法，在一定程度上提高了相关动画编程能力。通过将理论与实践相结合，我对 RVO 算法有了更深刻的认识。

同时，后续我继续将对代码进行修改，以解决偶尔出现的个体重叠问题和个

体在障碍物间卡住的问题。

七、 参考文献

1.Jur van den Berg, Ming Lin, Dinesh Manocha, “Reciprocal Velocity Obstacles for Real-Time Multi-Agent Navigation”, IEEE International Conference on Robotics and Automation (ICRA), 2008，pp. 1928-1935.

2. 李晓娜,孙立博,秦文虎.虚拟人群仿真的路径规划新算法[J].东南大学学报(自然科学版),2011,41(02):420-424.

3.此外在 csdn 上看了一些关于 RVO 的算法解释，参考了一些群组动画的基本构造和源代码。