# 作业一：三角形光栅化

## 一、 问题 1：实现 Look At 函数

### 1.1 实现过程

首先通过单位化相机到物体的向量，得到相机到注视点的方向向量 $\vec{f}$, 后单位化上方向量 $\vec{up}$ 得到相机上方的方向向量 $\vec{u}$，通过单位化 $\vec{f} \times \vec{u}$ 得到“右”方向上的单位向量，同时重新计算 $\vec{u} = \vec{s} \times \vec{f}$，确保它与 s 和 f 都垂直，并且仍然是相机的“上”方向。得到正交单位向量组后，创建 4*4 的单位矩阵，将 s、u、-f 赋给视图矩阵的前三列（由于 f 是相机到物体的方向，所以这里取负值）。最后计算相机坐标系的原点。它通过将 eye 向量与视图矩阵的前三列（代表相机坐标系的基向量）进行点积，然后取负值得到。这一步是为了将世界坐标系中的点转换到相机坐标系中。

### 1.2 Look At 代码

```
def LookAt(eye, center, up):
    f = (np.array(center) - np.array(eye)) / np.linalg.norm(np.array(center) - np.array(eye))
    u = np.array(up) / np.linalg.norm(np.array(up))
    s = np.cross(f, u)
    s = s / np.linalg.norm(s)
    u = np.cross(s, f)

    view = np.identity(4)
    view[0, :3] = s
    view[1, :3] = u
    view[2, :3] = -f
    view[:3, 3] = -np.dot(np.array(eye), view[:3, :3])

    return view
```

## 二、 问题 2：实现 Perspective 函数

### 2.1 实现过程

一、计算透视投影矩阵中的焦距 f。首先，将 fov 从度转换为弧度，然后计算 fov 的一半的正切值，最后取其倒数得到 f。并计算远近裁剪面之间的距离 depth。

二、创建一个 4x4 的零矩阵, 对透视投影矩阵中的元素进行设置，实现 x 轴方向上的缩放、y 轴方向上的缩放、深度方向上的缩放和深度方向上的偏移，同时确保在透视投影中，当 z 坐标为-1 时，点被映射到裁剪面上。

### 2.2 Perspective 代码

```
def Perspective(fov, aspect=1.0, near=0.1, far=10.0):
    f = 1.0 / np.tan(np.radians(fov) / 2)
    depth = far - near

    proj = np.zeros((4, 4))
    proj[0, 0] = f / aspect
    proj[1, 1] = f
    proj[2, 2] = (far + near) / depth
    proj[2, 3] = 2 * far * near / depth
    proj[3, 2] = -1

    return proj
```

## 三、 问题 3：实现 Ortho 函数

### 3.1 实现过程

一、创建 4*4 正交投影单位阵。

二、设置投影矩阵的 [0,0] 元素，它影响 x 轴上的缩放。这里计算的是视口宽度的倒数，乘以 2 是为了将视口映射到 [-1, 1] 的范围。

三、设置投影矩阵的 [1,1] 元素，它影响 y 轴上的缩放。这里计算的是视口高度的倒数，乘以 2 是为了将视口映射到 [-1, 1] 的范围。

四、设置投影矩阵的 [2,2] 元素，它影响 z 轴上的缩放。这里计算的是视口深度的倒数，乘以-2 是为了将视口映射到 [-1, 1] 的范围，并且取负值是为了将深度方向反转，使得近裁剪面在 z=-1 处，远裁剪面在 z=1 处。

五、设置投影矩阵的 [0,3] 元素，它影响 x 轴上的偏移。这里计算的是视口中心在 x 轴上的位置，然后将其映射到 [-1, 1] 的范围。

六、设置投影矩阵的 [1,3] 元素，它影响 y 轴上的偏移。这里计算的是视口中心在 y 轴上的位置，然后将其映射到 [-1, 1] 的范围。

七、设置投影矩阵的 [2,3] 元素，它影响 z 轴上的偏移。这里计算的是视口中心在 z 轴上的位置，然后将其映射到 [-1, 1] 的范围。

### 3.2 Ortho 代码

```
def Ortho(left, right, bottom, top, near, far):
    ortho = np.identity(4)
    ortho[0, 0] = 2 / (right - left)
    ortho[1, 1] = 2 / (top - bottom)
```

```
    ortho[2, 2] = -2 / (far - near)
    ortho[0, 3] = -(right + left) / (right - left)
    ortho[1, 3] = -(top + bottom) / (top - bottom)
    ortho[2, 3] = -(far + near) / (far - near)

    return ortho
```

# 四、 问题 4：实现 inside 函数

## 4.1 实现过程

首先获取三个顶点 A、B、C 的坐标，计算向量 $\vec{v_0}$□$\vec{v_1}$□$\vec{v_2}$，分别表示 A 到 C、A 到 B、和 A 到待判断点的向量。然后计算重心坐标 (u,v,w)，表示点在三角形内的相对位置，且满足 u+v+w=1。若 u、v、w 都大于等于 0，则点位于三角形内部或边界。

## 4.2 inside 代码

```
  def inside(self, x, y, z):
    A, B, C = self.vertices
    v0 = C - A
    v1 = B - A
    v2 = np.array([x, y, z]) - A

    dot00 = np.dot(v0, v0)
    dot01 = np.dot(v0, v1)
    dot02 = np.dot(v0, v2)
    dot11 = np.dot(v1, v1)
    dot12 = np.dot(v1, v2)

    invDenom = 1 / (dot00 * dot11 - dot01 * dot01)
    u = (dot11 * dot02 - dot01 * dot12) * invDenom
    v = (dot00 * dot12 - dot01 * dot02) * invDenom
    w = 1.0 - u - v

return u >= 0 and v >= 0 and w >= 0
```

# 五、 问题 5：实现 rotatenorm 函数

## 5.1 实现过程

### 5.1.1 rotationnorm

一、计算质心坐标 $center = \frac{A+B+C}{3}$。

二、计算 B-A 和 C-A 的叉积，得到法向量 norm 并标准化为单位法向量。

三、调用函数 rotation around axis(norm, thea) 来得到绕 norm 轴旋转 thea 角度的旋转矩阵。

四、进行旋转操作：1. 计算每个顶点相对于质心的位置。2. 将这些相对位置应用旋转矩阵.3. 将旋转后的顶点移回原始坐标系。

### 5.1.2 rotationaroundaxis

将输入的旋转轴 axis 标准化为单位向量。后计算旋转角度 theta 的余弦值和正弦值。最后构建一个 3*3 的旋转矩阵，用于绕 axis 旋转 thea 角度。其中的元素是根据罗德里格斯旋转公式计算的，其公式定义为

$$R = I + sin(\theta) \cdot [u]_\times + (1 - cos(\theta)) \cdot ([u]_\times)^2$$

其中，I 是单位矩阵，$[u]_\times$ 是轴向量 axis 的斜对称叉积矩阵。

## 5.2 rotatenorm 和 rotationaroundaxis 代码

```
def rotate_norm(self, thea):
  A, B, C = self.vertices
  center = (A + B + C) / 3.0
  norm = np.cross(B - A, C - A)
  norm = norm / np.linalg.norm(norm)
  rotation_matrix = rotation_around_axis(norm, thea)
  self.vertices = np.dot(self.vertices - center, rotation_matrix.T) + center
```

```
def rotation_around_axis(axis, theta):
  axis = axis / np.linalg.norm(axis)
  cos_theta = np.cos(np.radians(theta))
  sin_theta = np.sin(np.radians(theta))
  ux, uy, uz = axis

  rotation_matrix = np.array([
  [cos_theta + ux**2 * (1 - cos_theta), ux * uy * (1 - cos_theta) - uz * sin_theta, ux * uz
      * (1 - cos_theta) + uy * sin_theta],
```

```
    [uy * ux * (1 - cos_theta) + uz * sin_theta, cos_theta + uy**2 * (1 - cos_theta), uy * uz
        * (1 -  cos_theta) - ux * sin_theta],
    [uz * ux * (1 - cos_theta) - uy * sin_theta, uz * uy * (1 - cos_theta) + ux * sin_theta,
        cos_theta + uz**2 * (1 - cos_theta)]
    ])

return rotation_matrix
```

# 六、 ab

## 6.1 a: 透视投影图像和正交投影图像（已进行颜色插值）

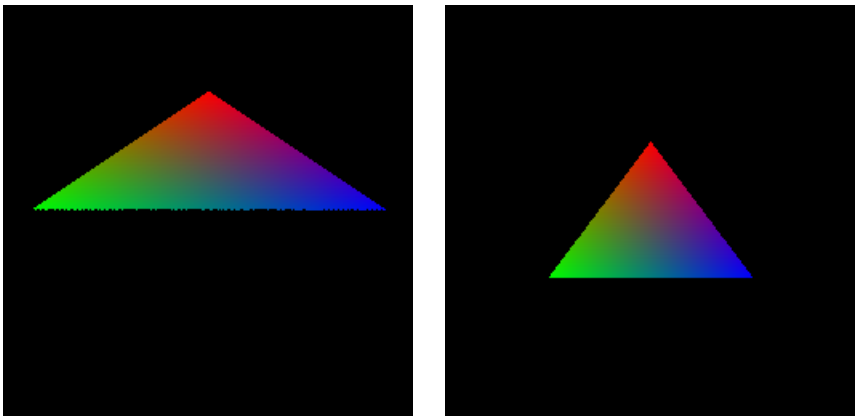

图 **1**　透视投影图像（左）和正交投影图像（右）

## 6.2 b: 三角形绕其自身中点旋转的透视投影图像（已进行颜色插值）

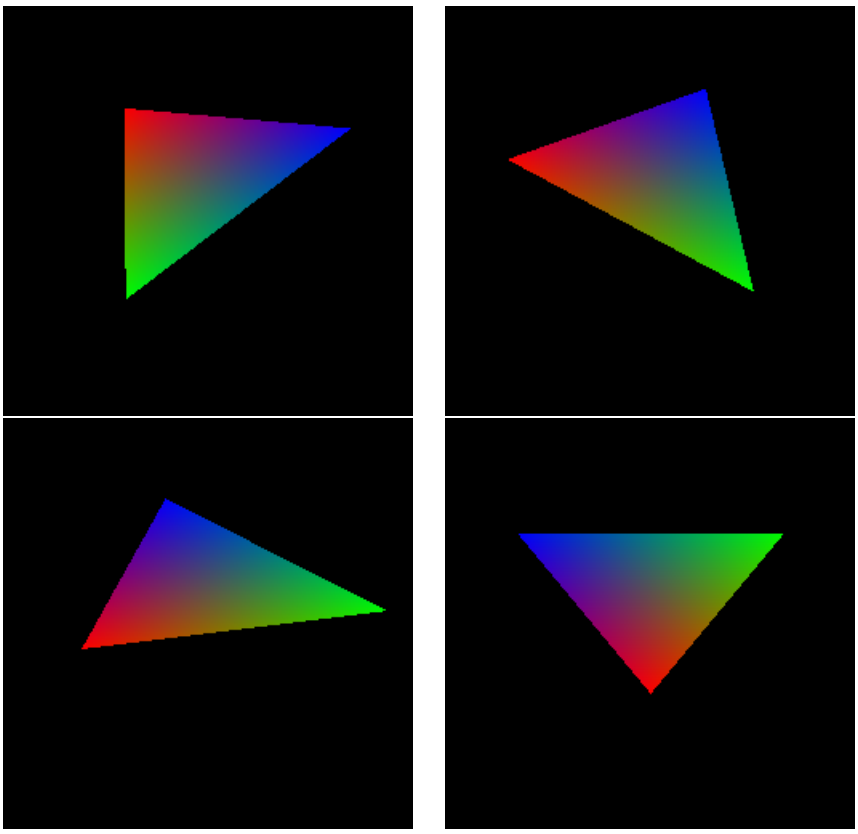

图 **2**　45°(左上) 90°(右上) 135°(左下) 180°(右下)

# 七、 cd: 颜色插值与深度测试

## 7.1 实现过程

### 7.1.1 颜色插值

利用 inside 函数中三角形重心的计算方法计算出所要操作点相对于三个顶点的重心坐标，取三个顶点的颜色利用该坐标取加权平均进行颜色插值。

### 7.1.2 深度测试

同样利用 inside 函数求出所要操作点相对于三个顶点的重心坐标，通过 $z = u * A[2] + v * B[2] + w * C[2]$ 计算出其深度值，并比较该点与深度缓冲区中当前像素位置的深度值，若该点深度值小于深度缓冲区中的值，这意味着点 P 更接近相机，因此应该替换深度缓冲区中的值，并更新颜色缓冲区以反映该点的颜色。

## 7.2 图像

c 题图像如下所示：

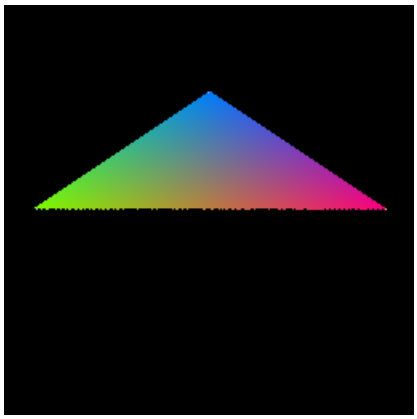

图 **3**　三个顶点的三个坐标的（**2.0**，**2.5**，**3.0**）三轮换

d 题图像如下所示：

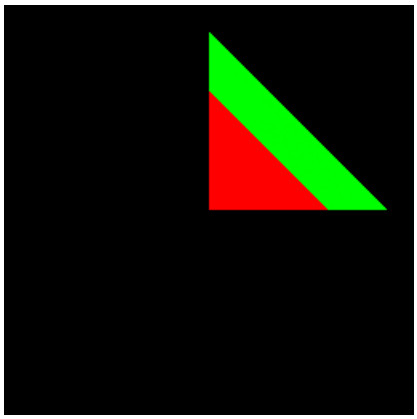

图 4　三角形 **T1**、**T2**

### 7.3 代码实现

```
for x in range(W):
    for y in range(H):
        if raster_t.inside(x, H - 1 - y, 0):
        A, B, C = raster_t.vertices
        v0, v1, v2 = B - A, C - A, np.array([x, H - 1 - y, 0]) - A
        d00, d01, d11 = np.dot(v0, v0), np.dot(v0, v1), np.dot(v1, v1)
        d20, d21 = np.dot(v2, v0), np.dot(v2, v1)
        denom = d00 * d11 - d01 * d01
        v = (d11 * d20 - d01 * d21) / denom
        w = (d00 * d21 - d01 * d20) / denom
        u = 1.0 - v - w
        z = u * A[2] + v * B[2] + w * C[2]
        if z < self.depth_buf[y][x]:
            self.depth_buf[y][x] = z
            self.color_buf[y][x] = (u * t.colors[0] + v * t.colors[1] + w * t.colors[2])
```

## 八、 e: 旋转插值

### 8.1 实现过程

首先创建初始和终止状态的三角形，然后利用 interpolate 函数对各时刻的三角形进行插值处理。

#### 8.1.1 interpolate 函数

一、利用时间 t 对两个三角形的颜色向量进行加权平均，实现颜色线性插值。

二、构建绕 x,y,z 轴旋转相应角度的旋转矩阵，并组合出最终的旋转矩阵。

三、将旋转矩阵转换成四元数，并且定义 [1,0,0,0] 为单位四元数，表示没有旋转。

四、使用之前定义的 quaternion slerp 函数在单位四元数（无旋转）和最终四元数之间进行球面线性插值。

```
def quaternion_slerp(q1, q2, t):
"""在两个四元数之间进行球面线性插值"""
q1 = np.array(q1)
q2 = np.array(q2)
# 计算四元数的点积
dot_product = np.dot(q1, q2)
# 如果点积为负，反转一个四元数以获得最短路径
if dot_product < 0.0:
q2 = -q2
dot_product = -dot_product
# 如果四元数非常接近，使用线性插值
if dot_product > 0.9995:
result = q1 + t * (q2 - q1)
return result / np.linalg.norm(result)
# 计算插值
theta_0 = np.arccos(dot_product)
sin_theta_0 = np.sin(theta_0)
theta = theta_0 * t
sin_theta = np.sin(theta)
s1 = np.cos(theta) - dot_product * sin_theta / sin_theta_0
s2 = sin_theta / sin_theta_0
return (s1 * q1) + (s2 * q2)
```

五、将颜色插值和旋转插值作用于原三角形，得到新的三角形。

## 8.2 代码实现

```
def interpolate(self, other_triangle, t):
"""在两个三角形之间进行插值"""
# 插值颜色
new_colors = (1 - t) * self.colors + t * other_triangle.colors
# 初始和最终旋转矩阵转换为四元数
identity_quaternion = [1, 0, 0, 0] # 初始没有旋转
x_rotation = rotation_around_axis([1, 0, 0], 30)
y_rotation = rotation_around_axis([0, 1, 0], 60)
z_rotation = rotation_around_axis([0, 0, 1], 30)
final_rotation_matrix = z_rotation @ y_rotation @ x_rotation
final_quaternion = R.from_matrix(final_rotation_matrix).as_quat()
# 插值四元数
interpolated_quaternion = quaternion_slerp(identity_quaternion, final_quaternion, t)
# 创建新的三角形
new_triangle = Triangle()
```

```
new_triangle.vertices = np.copy(self.vertices) # 复制初始顶点位置
new_triangle.colors = new_colors
# 应用插值后的四元数旋转
new_triangle.rotate_with_quaternion(interpolated_quaternion)
return new_triangle
```

### 8.3 图像

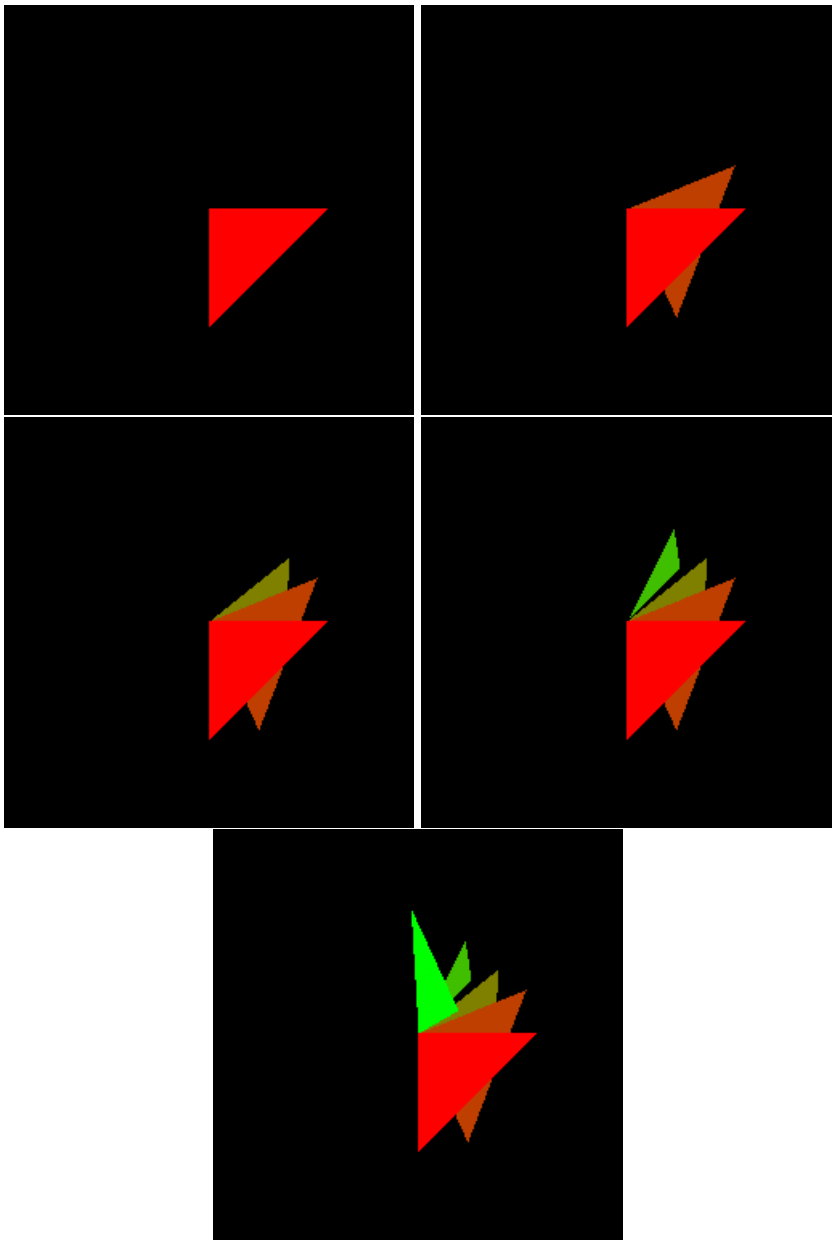

图 5　各时刻图像