实验报告：三维图形的透视投影与消隐

1. 实验原理

1.1. 透视投影原理

透视投影是三维空间中的点投影到二维平面上的一种方式。其基本原理是根据视点（观察者位置）和投影平面之间的关系，通过投影矩阵将三维坐标转换为二维坐标。在本实验中，视点设定为 $(0,0,d)$，投影面为 $O-xy$ 平面，即 $z=0$ 平面。

投影公式为：

$$x^{'}=\frac{d\cdot x}{d-z}$$
$$y^{'}=\frac{d\cdot y}{d-z}$$

其中，(x,y,z)为三维空间中的坐标，$(x^{'},y^{'})$为二维平面上的坐标，d为视点到投影面的距离。

1.2 消隐原理

消隐技术用于隐藏不可见的面，确保只有朝向观察者的面被显示。为了判断面是否可见，可以通过计算该面的法向量与视点方向向量的点积。若点积为正，表示该面朝向视点，应该显示；若点积为负，表示该面背向视点，应当隐藏。

视点方向的向量为 $v = (0, 0, d)$，面的法向量计算公式为

$$
\begin{aligned}
a &= y_1 (z_2 - z_3) + y_2 (z_3 - z_1) + y_3 (z_1 - z_2) \\
b &= z_1 (x_2 - x_3) + z_2 (x_3 - x_1) + z_3 (x_1 - x_2) \\
c &= x_1 (y_2 - y_3) + x_2 (y_3 - y_1) + x_3 (y_1 - y_2) \\
N &= (a, b, c)
\end{aligned}
$$

所以，我们只需要判断 $v \cdot N$ 的正负性。

1.3 旋转变换

三维图形的旋转可以通过旋转矩阵实现。以绕 y 轴的旋转为例，旋转角度为 θ 时，旋转矩阵为：

$$
R_y(\theta) = \begin{bmatrix}
\cos(\theta) & 0 & \sin(\theta) \\
0 & 1 & 0 \\
-\sin(\theta) & 0 & \cos(\theta)
\end{bmatrix}
$$

这个矩阵可以用于将三维点绕 y 轴进行旋转，由于我们的坐标采用的是右手定则（x 轴向右，z 轴向前，y 轴向上），所以旋转正数度代表逆时针旋转，选择负数度代表顺时针旋转。

2. 实验过程

2.1 定义三棱锥顶点

三棱锥的顶点分别为：A $(0, 0, 20)$，B $(20, 0, 20)$，C $(20, 0, 0)$，D $(10, 20, 10)$，这些顶点表示三棱锥的四个角。

```
// 三棱锥顶点
Point3D pyramid[] = {
    {0, 0, 20},  // A
    {20, 0, 20}, // B
    {20, 0, 0},  // C
    {10, 20, 10} // D
};
```

2.2 投影计算

使用透视投影公式，将三维顶点投影到二维平面上。视点距离设定为 $d = 50.0f$，计算得到每个顶点的二维坐标。

```
// 投影函数，将三维点投影到二维平面
Point2D project(Point3D point, float d)
{
    Point2D result;
    result.x = d * point.x / (d - point.z);
    result.y = d * point.y / (d - point.z);
    return result;
}
```

2.3 三维旋转

使用绕 y 轴旋转的旋转矩阵对三棱锥进行旋转。旋转角度设定为 -30 度，表示顺时针旋转。

```
// 三维旋转函数
Point3D rotateY(Point3D point, float angle)
{
    float rad = angle * M_PI / 180.0;
    Point3D rotated;
    rotated.x = point.x * cos(rad) + point.z * sin(rad);
    rotated.y = point.y;
    rotated.z = -point.x * sin(rad) + point.z * cos(rad);
    return rotated;
}
```

2.4 消隐算法

通过计算每个面的法向量和视点方向向量的点积，判断该面是否可见。如果点积为正，说明该面朝向视点，应该绘制；否则，隐藏该面。

```
// 计算平面法向量
Point3D calculateNormal(Point3D p1, Point3D p2, Point3D p3)
{
    Point3D normal = {
        p1.y * (p2.z - p3.z) + p2.y * (p3.z - p1.z) + p3.y * (p1.z - p2.z),
        p1.z * (p2.x - p3.x) + p2.z * (p3.x - p1.x) + p3.z * (p1.x - p2.x),
        p1.x * (p2.y - p3.y) + p2.x * (p3.y - p1.y) + p3.x * (p1.y - p2.y)};
    return normal;
}
```

```
// 计算每个面的可见性
bool visibleEdges[6] = {false};
Point3D normals[4];
normals[0] = calculateNormal(rotatedPyramid[0], rotatedPyramid[1], rotatedPyramid[2]);
normals[1] = calculateNormal(rotatedPyramid[0], rotatedPyramid[1], rotatedPyramid[3]);
normals[2] = calculateNormal(rotatedPyramid[1], rotatedPyramid[2], rotatedPyramid[3]);
normals[3] = calculateNormal(rotatedPyramid[2], rotatedPyramid[0], rotatedPyramid[3]);

visibleEdges[0] = isFaceVisible(normals[1], viewVector) || isFaceVisible(normals[0], viewVector);
visibleEdges[1] = isFaceVisible(normals[2], viewVector) || isFaceVisible(normals[0], viewVector);
visibleEdges[2] = isFaceVisible(normals[3], viewVector) || isFaceVisible(normals[0], viewVector);
visibleEdges[3] = isFaceVisible(normals[1], viewVector) || isFaceVisible(normals[3], viewVector);
visibleEdges[4] = isFaceVisible(normals[1], viewVector) || isFaceVisible(normals[2], viewVector);
visibleEdges[5] = isFaceVisible(normals[2], viewVector) || isFaceVisible(normals[3], viewVector);
```

2.5 绘制结果

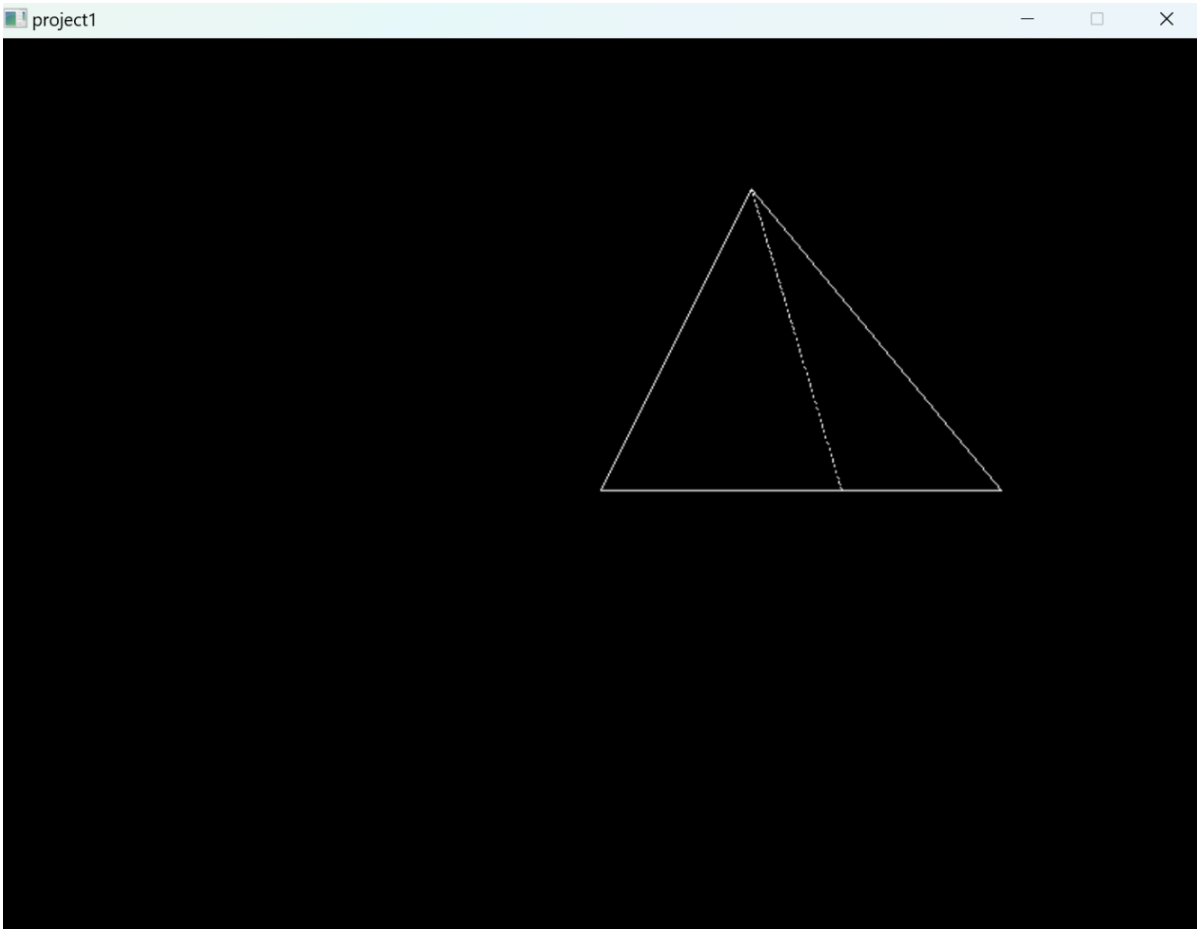
使用 `graphics.h` 库的函数绘制投影后的三棱锥。根据每条边的可见性，绘制实线或虚线。

```
// 绘制三棱锥投影
void drawPyramidProjection(Point2D projected[], bool visibleEdges[6])
{
    // 根据可见性绘制每条边
    setlinestyle(visibleEdges[0] ? PS_SOLID : PS_DASH, 1);
    line(projected[0].x, projected[0].y, projected[1].x, projected[1].y);

    setlinestyle(visibleEdges[1] ? PS_SOLID : PS_DASH, 1);
    line(projected[1].x, projected[1].y, projected[2].x, projected[2].y);

    setlinestyle(visibleEdges[2] ? PS_SOLID : PS_DASH, 1);
    line(projected[2].x, projected[2].y, projected[0].x, projected[0].y);

    setlinestyle(visibleEdges[3] ? PS_SOLID : PS_DASH, 1);
    line(projected[0].x, projected[0].y, projected[3].x, projected[3].y);

    setlinestyle(visibleEdges[4] ? PS_SOLID : PS_DASH, 1);
    line(projected[1].x, projected[1].y, projected[3].x, projected[3].y);

    setlinestyle(visibleEdges[5] ? PS_SOLID : PS_DASH, 1);
    line(projected[2].x, projected[2].y, projected[3].x, projected[3].y);
}
```

3. 实验结果

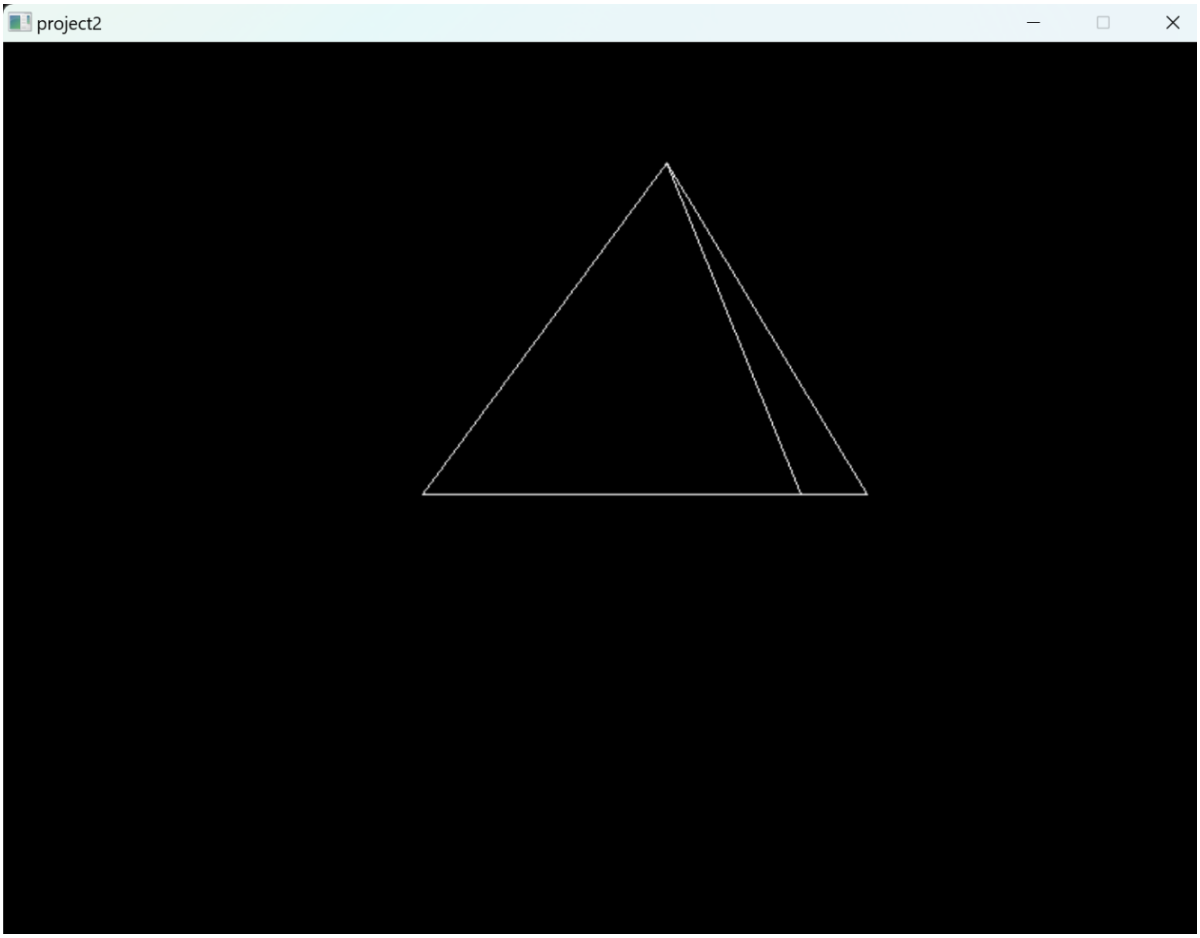
3.1 原始投影结果

在没有旋转的情况下，三棱锥的顶点通过透视投影转换为二维坐标，并显示在屏幕上。消隐过程隐藏了背向视点的面，确保只绘制可见的边。



3.2 旋转后的投影结果

通过绕 y 轴旋转 -30 度（顺时针旋转 30 度），三棱锥的形状发生了变化，顶点的位置被更新。旋转后的三棱锥再进行投影，显示在屏幕上，消隐算法继续确保只有朝向视点的面被绘制。



实验发现，经过绕 y 轴顺旋转 30 度后，原本中间哪一条不可视的直线现在可以看见了，所以原本的虚线现在变成了实线。

4. 实验错误点

4.1 坐标转换问题

在投影后的坐标转换过程中，未考虑坐标系的缩放和屏幕坐标的变换，导致图形的大小与期望不符。通过加入 `coordinates_transformation` 函数来进行适当的坐标缩放和转换。

```
// 数学坐标转屏幕坐标
void coordinates_transformation(Point2D &point)
{
    const float scale = 8.0f; // 放大倍数
    point.x = scale * point.x;
    point.y = scale * point.y;
}
```

4.2 消隐算法的问题

在初次实现消隐算法时，法向量计算与点积判断部分存在误差，导致部分可见的面被错误地隐藏。经过调试，修正了法向量的计算和点积判断逻辑，确保了消隐效果正确。

4.3 旋转算法的细节

旋转算法的角度问题可能会导致图形的错误旋转。特别是负角度表示顺时针旋转时，容易与期望的旋转方向发生偏差。经过修正，确保旋转正确。

5. 实验心得

5.1. 透视投影的重要性

透视投影是计算机图形学中非常基础的概念，它使我们能够将三维空间的图形准确地展示到二维平面上。理解透视投影的公式和实现方式，对于后续学习其他计算机图形学技术非常有帮助。

5.2. 消隐技术的应用

消隐技术能够有效地优化图形显示，减少计算量，并提升用户的视觉体验。在实际应用中，消隐是实时渲染和三维建模中的关键技术之一。

5.3 三维旋转的复杂性

三维旋转是图形学中常见的变换之一。虽然旋转矩阵的推导看起来简单，但在实际编程过程中，需要注意角度的单位（弧度与角度之间的转换）、旋转轴的选择（本实验为 y 轴）等细节问题。