计算机图形学课程设计 - 完整技术报告

目录 (Table of Contents)

1. 项目介绍

1.1 什么是计算机图形学?

计算机图形学是研究如何用计算机生成、处理和显示图形图像的学科。简单来说，就是研究"电脑怎么画图"。

当你玩游戏、看3D电影、使用CAD软件时，背后都有计算机图形学的技术在支撑。

1.2 本项目的目标

本项目的目标是：**从零开始，不使用任何高级3D图形库（如OpenGL、DirectX、Unity），仅用Java基础绘图功能，手动实现一个完整的图形学系统。**

这意味着：

- 每一条直线，我们都要自己计算应该点亮哪些像素
- 每一个3D物体，我们都要自己计算它在2D屏幕上的投影位置
- 每一个颜色变化，我们都要自己计算光照效果

1.3 项目实现的功能

本项目实现了以下功能模块：

序号	功能模块	具体内容
1	直线绘制	Bresenham算法、DDA算法
2	圆绘制	Bresenham中点圆算法、正负法
3	多边形填充	扫描线填充算法
4	区域填充	4连通种子填充算法
5	几何变换	平移、旋转、缩放、对称、错切
6	裁剪	Cohen-Sutherland直线裁剪
7	投影	透视投影、平行投影
8	消隐	Z-Buffer、后向面消除
9	光照	Phong光照模型
10	3D场景	机器人模型、无限地板
11	交互功能	鼠标旋转、缩放、文字输入

---

2. 技术环境与工具

2.1 开发语言：Java

Java是一种面向对象的编程语言。本项目使用Java 17版本。

为什么选择Java?

- Java自带的 `java.awt` 包提供了基础的2D绑图功能
- 跨平台，可以在Windows、Mac、Linux上运行
- 不需要安装额外的图形库

2.2 构建工具：Maven

什么是Maven?
Maven是一个项目管理工具，它可以帮助我们：

- 自动下载项目需要的依赖库
- 自动编译Java代码
- 自动打包成可执行程序

Maven的配置文件：pom.xml

```
<!-- 这是pom.xml的核心部分 -->
<project>
    <groupId>com.graphics</groupId>      <!-- 项目组织名 -->
    <artifactId>cg-course-project</artifactId>  <!-- 项目名称 -->
    <version>1.0</version>               <!-- 版本号 -->

    <properties>
        <maven.compiler.source>17</maven.compiler.source>  <!-- Java版本 -->
        <maven.compiler.target>17</maven.compiler.target>
    </properties>
</project>
```

2.3 项目目录结构

```
cg_final/                          <-- 项目根目录
|
├── pom.xml                        <-- Maven配置文件
```

```
├── README.md                    <-- 项目说明文件
├── ProjectReport.md             <-- 本技术报告
│
└── src/                         <-- 源代码目录
    └── main/
        └── java/
            └── com/
                └── graphics/    <-- 所有Java代码都在这里
                    ├── MainFrame.java        <-- 主窗口
                    ├── Scene3DPanel.java     <-- 3D渲染面板
                    ├── Canvas2DPanel.java    <-- 2D绘图画布
                    ├── Robot.java            <-- 机器人模型
                    ├── Matrix4.java          <-- 矩阵运算
                    ├── Polygon3D.java        <-- 3D多边形
                    ├── Stage.java            <-- 舞台/地板
                    └── *Dialog.java          <-- 各种对话框(10+个)
```

---

3. 项目结构说明

3.1 主要类的功能

MainFrame.java - 主窗口类

这是程序的入口，负责：

- 创建程序窗口
- 创建菜单栏（光栅图形、变换、裁剪、投影...）
- 创建工具栏（快捷按钮）
- 管理2D画布和3D场景的切换

```java
// MainFrame.java 核心代码
public class MainFrame extends JFrame {

    private Scene3DPanel scene3DPanel;  // 3D场景面板
    private Canvas2DPanel canvas2DPanel; // 2D画布面板

    public MainFrame() {
        super("计算机图形学课程大作业");  // 窗口标题
        initializeUI();  // 初始化界面
    }

    private void initializeUI() {
        setSize(1400, 900);          // 设置窗口大小
        setJMenuBar(createMenuBar()); // 创建菜单栏
        add(createToolBar(), BorderLayout.NORTH); // 添加工具栏
        // ... 更多初始化代码
    }
}
```

Canvas2DPanel.java - 2D画布类

这个类负责所有2D图形的绘制，包括直线、圆、多边形等。

核心概念：像素绘制

```java
// 这是最基本的像素绘制方法
// 参数: x坐标, y坐标, 颜色
public void setPixel(int x, int y, Color color) {
    // 检查坐标是否在画布范围内
    if (x >= 0 && x < canvas.getWidth() && y >= 0 && y < canvas.getHeight()) {
        // getRGB()方法将Color对象转换为整数颜色值
        canvas.setRGB(x, y, color.getRGB());
    }
}
```

Matrix4.java - 矩阵运算类

这个类提供所有3D变换需要的矩阵运算。

为什么需要矩阵?
在计算机图形学中，平移、旋转、缩放等变换都用4x4矩阵表示。通过矩阵乘法，可以将多个变换合并。

```
单位矩阵（不做任何变换）:
| 1  0  0  0 |
| 0  1  0  0 |
| 0  0  1  0 |
| 0  0  0  1 |

平移矩阵（移动tx, ty, tz）:
| 1  0  0  tx |
| 0  1  0  ty |
| 0  0  1  tz |
| 0  0  0  1  |
```

Robot.java - 机器人模型类

这个类定义了3D机器人的结构，包括身体、头、手臂、腿等部件。

```java
public class Robot {
    // 机器人的位置
    private double posX = 0, posY = 0, posZ = 0;

    // 机器人的旋转角度
    private double rotationY = 0;

    // 机器人由多个部件组成
    private List<RobotPart> parts = new ArrayList<>();

    // RobotPart是内部类，表示一个部件（如手臂、腿）
    private static class RobotPart {
        String name;        // 部件名称
        double[] position;  // 相对位置
        double[] size;      // 尺寸
        Color color;        // 颜色
    }
}
```

---

4. 核心算法详解

4.1 直线扫描转换算法

4.1.1 问题描述

问题： 已知直线的起点(x0, y0)和终点(x1, y1)，如何确定应该点亮屏幕上的哪些像素点?

难点： 屏幕是由离散的像素组成的，但直线在数学上是连续的。我们需要找到最接近理想直线的像素点。

```
理想直线（数学上）        实际绘制（像素点）
      *                        ■
     *                       ■
    *                      ■
   *                     ■
  *                    ■
```

4.1.2 DDA算法（数字微分分析器）

基本思想： 根据直线的斜率，每次x增加1，y增加斜率k。

```
// DDA直线算法 - 代码位置: Canvas2DPanel.java
public void drawLineDDA(int x0, int y0, int x1, int y1, Color color) {
    // 第1步：计算x和y方向的变化量
    int dx = x1 - x0;  // x方向变化量
    int dy = y1 - y0;  // y方向变化量

    // 第2步：确定步数（取dx和dy中较大的那个）
    // 这样可以保证每次至少有一个坐标增加1
    int steps = Math.max(Math.abs(dx), Math.abs(dy));

    // 第3步：计算每一步x和y的增量
    double xIncrement = (double) dx / steps;  // 每步x的增量
    double yIncrement = (double) dy / steps;  // 每步y的增量

    // 第4步：从起点开始，逐步绘制
    double x = x0;
    double y = y0;

    for (int i = 0; i <= steps; i++) {
        // 四舍五入取整，然后绘制像素
        setPixel((int) Math.round(x), (int) Math.round(y), color);
        // 移动到下一点
        x += xIncrement;
        y += yIncrement;
    }
}
```

算法分析：

- 优点：思路简单直观
- 缺点：使用浮点运算，速度较慢

4.1.3 Bresenham直线算法

基本思想： 完全使用整数运算，通过"误差项"来判断下一个像素应该在哪里。

详细步骤解释：

```
// Bresenham直线算法 - 代码位置: Canvas2DPanel.java
public void drawLineBresenham(int x0, int y0, int x1, int y1, Color color) {

    // ========== 第1步：计算基本参数 ==========
    int dx = Math.abs(x1 - x0);  // x方向的距离（取绝对值）
    int dy = Math.abs(y1 - y0);  // y方向的距离（取绝对值）

    // ========== 第2步：确定步进方向 ==========
    // sx: x每次移动的方向（+1向右，-1向左）
    int sx = (x0 < x1) ? 1 : -1;
    // sy: y每次移动的方向（+1向下，-1向上）
```

```
    int sy = (y0 < y1) ? 1 : -1;

    // ========== 第3步：初始化误差项 ==========
    // 误差项用于判断下一个点是水平移动还是对角移动
    // 初始值为 dx - dy
    int err = dx - dy;

    // ========== 第4步：循环绘制每个像素 ==========
    while (true) {
        // 绘制当前点
        setPixel(x0, y0, color);

        // 如果到达终点，结束循环
        if (x0 == x1 && y0 == y1) {
            break;
        }

        // 计算误差项的两倍（避免浮点运算）
        int e2 = 2 * err;

        // 判断是否需要在x方向移动
        // 如果 e2 > -dy，说明误差偏向x轴
        if (e2 > -dy) {
            err = err - dy;  // 更新误差项
            x0 = x0 + sx;    // x坐标移动一步
        }

        // 判断是否需要在y方向移动
        // 如果 e2 < dx，说明误差偏向y轴
        if (e2 < dx) {
            err = err + dx;  // 更新误差项
            y0 = y0 + sy;    // y坐标移动一步
        }
    }
}
```

图解说明：

```
假设要画从(0,0)到(5,3)的直线：

步骤  当前点    err    e2    移动决策
----------------------------------------
1    (0,0)     2      4     x+1, y+1  → (1,1)
2    (1,1)    -1     -2     x+1       → (2,1)
3    (2,1)     2      4     x+1, y+1  → (3,2)
4    (3,2)    -1     -2     x+1       → (4,2)
5    (4,2)     2      4     x+1, y+1  → (5,3) 到达终点
```

4.2 圆扫描转换算法

4.2.1 问题描述

问题： 已知圆心(xc, yc)和半径r，如何绘制圆？

关键技巧：八路对称性
圆具有很强的对称性。如果知道圆上一点(x, y)，可以直接推出其他7个对称点：

4.2.2 Bresenham中点圆算法

```java
// Bresenham画圆算法 - 代码位置: Canvas2DPanel.java
public void drawCircleBresenham(int xc, int yc, int r, Color color) {

    // ========== 第1步：初始化 ==========
    int x = 0;      // 从(0, r)点开始
    int y = r;
    int d = 3 - 2 * r;  // 初始决策参数

    // 绘制初始的8个对称点
    drawCirclePoints(xc, yc, x, y, color);

    // ========== 第2步：循环计算圆上的点 ==========
    // 只需要计算1/8的圆弧，其他部分通过对称性获得
    while (y >= x) {
        x++;  // x每次加1

        // 根据决策参数判断y是否需要减1
        if (d > 0) {
            // 如果d>0，中点在圆外，选择下方的点
            y--;
            d = d + 4 * (x - y) + 10;
        } else {
            // 如果d<=0，中点在圆内或圆上，选择右方的点
            d = d + 4 * x + 6;
        }

        // 绘制8个对称点
        drawCirclePoints(xc, yc, x, y, color);
    }
}

// 辅助方法：绘制8个对称点
private void drawCirclePoints(int xc, int yc, int x, int y, Color color) {
    setPixel(xc + x, yc + y, color);  // 第1象限
    setPixel(xc - x, yc + y, color);  // 第2象限
    setPixel(xc + x, yc - y, color);  // 第4象限
    setPixel(xc - x, yc - y, color);  // 第3象限
    setPixel(xc + y, yc + x, color);  // 交换x,y后的4个点
    setPixel(xc - y, yc + x, color);
    setPixel(xc + y, yc - x, color);
    setPixel(xc - y, yc - x, color);
}
```

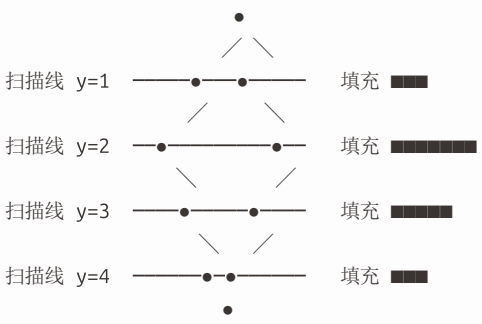
4.3 多边形扫描填充算法

4.3.1 问题描述

问题： 已知多边形的各个顶点坐标，如何填充多边形内部?

基本思想： 按照y坐标从上到下扫描，找出每条扫描线与多边形边的交点，然后填充交点之间的像素。

扫描线填充示意图：



4.3.2 算法实现

```java
// 多边形扫描填充 - 代码位置: Canvas2DPanel.java
public void scanLineFill(int[] xPoints, int[] yPoints, int n, Color color) {

    // ========== 第1步：找出多边形的y坐标范围 ==========
    int yMin = Integer.MAX_VALUE;
    int yMax = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        yMin = Math.min(yMin, yPoints[i]);
        yMax = Math.max(yMax, yPoints[i]);
    }

    // ========== 第2步：对每条扫描线进行处理 ==========
    for (int y = yMin; y <= yMax; y++) {

        // 存储该扫描线与多边形边的所有交点的x坐标
        List<Integer> intersections = new ArrayList<>();

        // ========== 第3步：计算扫描线与每条边的交点 ==========
        for (int i = 0; i < n; i++) {
            // 获取边的两个端点
            int x1 = xPoints[i];
            int y1 = yPoints[i];
            int x2 = xPoints[(i + 1) % n];  // 下一个顶点,最后一个连回第一个
            int y2 = yPoints[(i + 1) % n];

            // 检查扫描线y是否与这条边相交
            if ((y1 <= y && y < y2) || (y2 <= y && y < y1)) {
                // 计算交点的x坐标（使用线性插值）
                int x = x1 + (y - y1) * (x2 - x1) / (y2 - y1);
                intersections.add(x);
            }
        }

        // ========== 第4步：对交点排序 ==========
        Collections.sort(intersections);

        // ========== 第5步：两两配对填充 ==========
        // 第1个到第2个之间填充，第3个到第4个之间填充...
        for (int i = 0; i < intersections.size() - 1; i += 2) {
            int xStart = intersections.get(i);
            int xEnd = intersections.get(i + 1);

            // 填充这一段
            for (int x = xStart; x <= xEnd; x++) {
                setPixel(x, y, color);
            }
        }
    }
}
```

4.4 区域填充算法（种子填充）

4.4.1 问题描述

问题： 给定一个封闭区域内的一点（种子点），如何填充整个区域？

基本思想： 从种子点开始，向四个方向（上、下、左、右）扩展，如果邻居点不是边界且未被填充，就填充它并继续扩展。

4.4.2 算法实现

```java
// 4连通种子填充算法 - 代码位置: Canvas2DPanel.java
public void seedFill4(int x, int y, Color fillColor, Color boundaryColor) {

    // 使用栈来存储待处理的点
    // 为什么用栈而不是递归? 因为递归深度太大会导致栈溢出
    java.util.Stack<Point> stack = new java.util.Stack<>();

    // 将种子点入栈
    stack.push(new Point(x, y));

    // ========== 循环处理栈中的每个点 ==========
    while (!stack.isEmpty()) {
        // 取出栈顶的点
        Point p = stack.pop();

        // 边界检查: 如果点在画布外, 跳过
        if (p.x < 0 || p.x >= canvas.getWidth() ||
            p.y < 0 || p.y >= canvas.getHeight()) {
            continue;
        }

        // 获取该点当前的颜色
        Color currentColor = new Color(canvas.getRGB(p.x, p.y), true);

        // 如果已经是填充色或边界色, 跳过
        if (currentColor.equals(fillColor) || currentColor.equals(boundaryColor)) {
            continue;
        }

        // ========== 填充当前点 ==========
        setPixel(p.x, p.y, fillColor);

        // ========== 将四个邻居入栈 ==========
        stack.push(new Point(p.x + 1, p.y));  // 右
        stack.push(new Point(p.x - 1, p.y));  // 左
        stack.push(new Point(p.x, p.y + 1));  // 下
        stack.push(new Point(p.x, p.y - 1));  // 上
    }
}
```

4.5 几何变换算法

4.5.1 齐次坐标系

在3D图形学中，我们使用**4维向量**来表示3D点:

- 点 (x, y, z) 表示为 (x, y, z, 1)
- 第4个分量w=1表示这是一个点

为什么需要4维? 因为平移变换无法用3x3矩阵表示，但可以用4x4矩阵统一表示所有变换。

4.5.2 各种变换矩阵

```java
// 代码位置: Matrix4.java

// ========== 1. 平移矩阵 ==========
// 将点沿(tx, ty, tz)方向移动
public static double[][] translate(double tx, double ty, double tz) {
    return new double[][] {
        { 1, 0, 0, tx },   // x' = x + tx
        { 0, 1, 0, ty },   // y' = y + ty
```

```
        { 0, 0, 1, tz },   // z' = z + tz
        { 0, 0, 0, 1  }
    };
}

// ========== 2. 缩放矩阵 ==========
// 将点缩放(sx, sy, sz)倍
public static double[][] scale(double sx, double sy, double sz) {
    return new double[][] {
        { sx, 0,  0,  0 },   // x' = x * sx
        { 0,  sy, 0,  0 },   // y' = y * sy
        { 0,  0,  sz, 0 },   // z' = z * sz
        { 0,  0,  0,  1 }
    };
}

// ========== 3. 绕Y轴旋转矩阵 ==========
// 将点绕Y轴旋转angle弧度
public static double[][] rotateY(double angle) {
    double c = Math.cos(angle);  // 余弦值
    double s = Math.sin(angle);  // 正弦值
    return new double[][] {
        {  c, 0, s, 0 },   // x' = x*cos + z*sin
        {  0, 1, 0, 0 },   // y' = y (y不变)
        { -s, 0, c, 0 },   // z' = -x*sin + z*cos
        {  0, 0, 0, 1 }
    };
}
```

4.5.3 矩阵乘法

矩阵乘法的意义： 将两个变换合并为一个变换。

例如：先平移、再旋转 = 旋转矩阵 × 平移矩阵

```
// 矩阵乘法 - 代码位置: Matrix4.java
public static double[][] multiply(double[][] A, double[][] B) {
    double[][] result = new double[4][4];

    // 结果矩阵的第i行第j列 = A的第i行 点乘 B的第j列
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            result[i][j] = 0;
            for (int k = 0; k < 4; k++) {
                result[i][j] += A[i][k] * B[k][j];
            }
        }
    }

    return result;
}
```

图解矩阵乘法：

```
      B的第j列
         ↓
A   [ a b c d ]   B   [ . . j . ]      result[i][j] =
    [ e f g h ]       [ . . j . ]      a*j + b*j + c*j + d*j
    [ i j k l ] ←i行  [ . . j . ]      (点乘)
    [ m n o p ]       [ . . j . ]
```

4.6 裁剪算法

4.6.1 Cohen-Sutherland直线裁剪

问题： 给定一个矩形窗口和一条直线，如何只保留窗口内的部分？

基本思想：

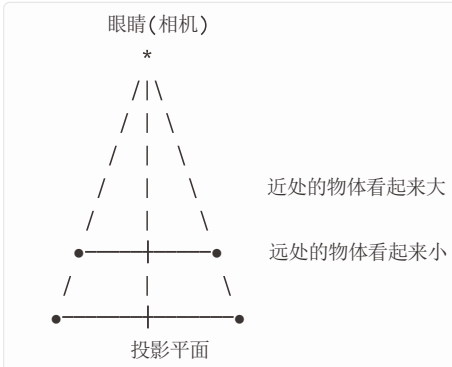
1. 给直线两端点编码（表示它相对于窗口的位置）
2. 根据编码快速判断直线是否完全在窗口内/外
3. 对于部分在内的直线，计算交点并裁剪

```
// Cohen-Sutherland裁剪 - 代码位置: ClippingDialog.java

// 区域编码（4位二进制）
// bit 3: 上边界外  bit 2: 下边界外  bit 1: 右边界外  bit 0: 左边界外
private int computeCode(double x, double y) {
    int code = 0;
    if (x < xMin) code |= 1;       // 左边界外: 0001
    if (x > xMax) code |= 2;       // 右边界外: 0010
    if (y < yMin) code |= 4;       // 下边界外: 0100
    if (y > yMax) code |= 8;       // 上边界外: 1000
    return code;
}

// 裁剪算法主体
public void cohenSutherlandClip(double x0, double y0, double x1, double y1) {
    int code0 = computeCode(x0, y0);  // 起点编码
    int code1 = computeCode(x1, y1);  // 终点编码

    while (true) {
        // 情况1: 两端点编码都为0，完全在窗口内
        if ((code0 | code1) == 0) {
            drawLine(x0, y0, x1, y1);  // 绘制整条线
            break;
        }

        // 情况2: 两端点编码AND不为0，完全在窗口外同一侧
        if ((code0 & code1) != 0) {
            break;  // 不绘制
        }

        // 情况3: 需要裁剪
        // ... 计算与边界的交点并更新端点
    }
}
```

4.7 三维投影变换

4.7.1 透视投影

原理： 模拟人眼看物体的效果—近大远小。




```
// 透视投影矩阵 - 代码位置: Matrix4.java
public static double[][] perspective(double fov, double aspect, double near, double far) {
    // fov: 视野角度（Field of View）
    // aspect: 宽高比
    // near: 近裁剪面距离
    // far: 远裁剪面距离

    double tanHalfFov = Math.tan(fov / 2);
    double f = 1.0 / tanHalfFov;

    return new double[][] {
        { f/aspect, 0,  0,                          0                          },
        { 0,        f,  0,                          0                          },
        { 0,        0,  (far+near)/(near-far),      (2*far*near)/(near-far)    },
        { 0,        0,  -1,                         0                          }
    };
}
```

4.8 消隐算法

4.8.1 后向面消除

原理： 摄像机看不到的面（背对摄像机的面）不需要绘制。

判断方法：计算面的法向量与视线方向的点积

- 点积 > 0：面朝向相机，需要绘制
- 点积 <= 0：面背对相机，不绘制

```
// 后向面消除 - 代码位置: Scene3DPanel.java
private boolean isFrontFacing(double[] v0, double[] v1, double[] v2, double[] viewDir) {
    // 计算面的法向量 (v1-v0) × (v2-v0)
    double[] edge1 = {v1[0]-v0[0], v1[1]-v0[1], v1[2]-v0[2]};
    double[] edge2 = {v2[0]-v0[0], v2[1]-v0[1], v2[2]-v0[2]};

    // 叉积计算法向量
    double[] normal = {
        edge1[1]*edge2[2] - edge1[2]*edge2[1],
        edge1[2]*edge2[0] - edge1[0]*edge2[2],
        edge1[0]*edge2[1] - edge1[1]*edge2[0]
    };

    // 点积判断朝向
    double dot = normal[0]*viewDir[0] + normal[1]*viewDir[1] + normal[2]*viewDir[2];

    return dot > 0;  // 正向朝着相机
}
```

4.9 光照模型

4.9.1 Phong光照模型

Phong模型将光照分为三个部分：

1. **环境光（Ambient）**：场景中的均匀光照，没有方向
2. **漫反射（Diffuse）**：物体表面均匀散射的光
3. **镜面反射（Specular）**：物体表面的高光点

```
最终颜色 = 环境光 + 漫反射 + 镜面反射
         = Ka * Ia + Kd * Il * (N·L) + Ks * Il * (R·V)^n

其中：
- Ka, Kd, Ks：材质的环境/漫反射/镜面反射系数
- Ia, Il：环境光强度，光源强度
- N：表面法向量
- L：光源方向
- R：反射方向
- V：视线方向
- n：光泽度（越大高光越集中）
```

---

5. 系统功能演示

5.1 操作说明

鼠标操作：

- 左键拖拽：旋转3D场景
- 右键拖拽：平移场景
- 滚轮：缩放场景

菜单操作：

- 光栅图形：切换2D/3D模式，绘制直线、圆、多边形
- 变换：对选中的物体进行平移、旋转、缩放
- 投影：切换透视/平行投影
- 光照明模型：调整光源和材质

5.2 典型使用流程

1. **绘制2D图形：**
   菜单 → 光栅图形 → 切换到2D画布 → 直线绘制 → 在画布上点击两点
2. **查看3D机器人：**
   菜单 → 光栅图形 → 切换到3D场景 → 拖拽鼠标旋转视角
3. **设计自定义形状：**
   菜单 → 机器人动画 → 形状设计器 → 添加形状 → 生成到场景

---

6. 开发过程中遇到的困难

6.1 深度排序问题

问题描述：
在绘制多个3D物体时，远处的物体有时会画在近处物体上面，造成遮挡关系错误。

解决方案：

1. 实现后向面消除，剔除背对相机的面
2. 按照物体到相机的距离对多边形排序，先画远的再画近的
3. 分层渲染：先渲染地板，再渲染机器人

6.2 机器人关节层级问题

问题描述：
机器人的手臂应该跟随身体移动，但手臂自己也可以独立旋转。如何实现这种层级关系？

解决方案：
使用**场景图(Scene Graph)**结构：

- 每个部件记录相对于父节点的变换
- 渲染时，将父节点的变换矩阵乘以子节点的变换矩阵
- 这样当身体移动时，手臂自动跟随；手臂旋转时，只影响自己

```
机器人层级结构：
Robot（根节点）
├── Body（身体）
│   ├── Head（头）
│   ├── LeftArm（左臂）
│   │   └── LeftHand（左手）
│   ├── RightArm（右臂）
│   │   └── RightHand（右手）
│   ├── LeftLeg（左腿）
│   └── RightLeg（右腿）
```

6.3 坐标系转换问题

问题描述：
Java Swing的坐标系原点在左上角（Y向下），而图形学标准坐标系原点在中心（Y向上）。

解决方案：
在视口变换阶段进行Y轴翻转：

```
screenY = height - worldY;
```

---

7. 编译与运行指南

7.1 环境要求

- **Java JDK 17** 或更高版本
- **Maven 3.6** 或更高版本

7.2 检查环境

打开命令行，输入：

```
java -version    # 应显示 java version "17.x.x"
mvn -version     # 应显示 Maven 3.x.x
```

7.3 编译项目

```
# 进入项目目录
cd f:\desktop\cg_final

# 编译项目
mvn clean compile
```

如果看到 `BUILD SUCCESS`，说明编译成功。

7.4 运行项目

```
mvn exec:java
```

程序窗口将自动弹出。

7.5 常见问题

Q：提示"找不到Java"
A：需要安装JDK并配置环境变量JAVA_HOME

Q：提示"找不到mvn命令"
A：需要安装Maven并将其bin目录添加到PATH

---

8. 总结与心得

8.1 项目收获

通过本项目的开发，我深入理解了：

1. 计算机图形学的底层原理，不再是"调用API就能画图"的黑盒
2. 3D渲染管线的完整流程：建模→变换→投影→光栅化
3. 矩阵在图形变换中的核心作用
4. 软件工程实践：模块化设计、代码组织、调试技巧

8.2 可改进之处

1. 可以实现更高级的着色算法（如纹理映射）
2. 可以支持更多的3D模型格式导入
3. 可以使用GPU加速渲染（OpenGL/DirectX）