计算机图形学复习笔记与算法书

吴建豪

1 光栅图形学基础

1.1 光栅化概念

- 光栅化是将几何图形转换为像素表示的过程
- 基本任务是将连续的几何体离散化为像素点阵
- 需要考虑抗锯齿、采样等问题

1.2 直线光栅化算法

1.2.1 DDA 算法 (数字微分分析器)

DDA 算法通过计算斜率并在每一步使用浮点数累加来模拟直线路径,本质上是用离散的点来逼近连续的直线.在每一步中,算法通过对浮点坐标进行四舍五入来确定最接近理想直线的整数像素坐标,这种简单的取整方式虽然可能在某些情况下不是最优选择,但保证了算法的简单性和直观性.

- 基本原理:
  - 基于浮点或定点的增量计算
  - 通过斜率计算逐步确定下一个像素位置
- 工作过程:
  1. 计算直线斜率 $k = \Delta y / \Delta x$
  2. 从起点开始,每次在 x 方向步进 1 个单位
  3. y 坐标按斜率增加:$y = y + k$
  4. 将(x,y)转换为最近的整数坐标
- 特点:
  - 实现简单直观
  - 需要浮点运算
  - 效率相对较低

```C++
void DDALine(int x1, int y1, int x2, int y2) {
    float dx = x2 - x1;
    float dy = y2 - y1;
    float steps = abs(dx) > abs(dy) ? abs(dx) : abs(dy);
    // 计算每步的增量
    float xInc = dx / steps;
    float yInc = dy / steps;
    float x = x1, y = y1;
    for (int i = 0; i <= steps; i++) {
        putPixel(round(x), round(y)); // 绘制像素
        x += xInc;
        y += yInc;
    }
}
```

1.2.2 Bresenham 算法

Bresenham 算法通过累积误差项来选择最接近理想直线的像素点,巧妙地将浮点运算转化为整数运算,是一种高效的增量式算法.在每一步中,算法通过比较误差项与 0 的关系来决定是选择水平方向相邻的像素还是对角线方向的像素,这种选择方式保证了所选像素始终是距离理想直线最近的点,同时避免了浮点运算.

- 基本原理:
  - 使用整数增量法
  - 通过误差判定选择最接近直线的像素
- 工作过程:
  1. 假设当前在像素 P(x,y)
  2. 下一个像素只有两个选择:
     - P1(x+1, y)
     - P2(x+1, y+1)
  3. 计算误差项:
     - 如果误差<0,选择 P1
     - 如果误差 ⩾0,选择 P2
  4. 更新误差项(只需整数加减)
- 特点:
  - 仅使用整数运算
  - 避免浮点运算开销
  - 效率高,适合硬件实现
  - 是最常用的直线光栅化算法

```cpp
void BresenhamLine(int x1, int y1, int x2, int y2) {
    int dx = abs(x2 - x1);
    int dy = abs(y2 - y1);
    int sx = x1 < x2 ? 1 : -1;
    int sy = y1 < y2 ? 1 : -1;
    int err = dx - dy;

    while (true) {
        putPixel(x1, y1); // 绘制像素
        if (x1 == x2 && y1 == y2) break;
        int e2 = 2 * err;
        if (e2 > -dy) {
            err -= dy;
            x1 += sx;
        }
        if (e2 < dx) {
            err += dx;
            y1 += sy;
        }
    }
}
```

1.2.3 中点算法 (Midpoint)

中点算法基于几何直观,通过判断候选像素中点与理想直线的位置关系来选择最优像素,提供了一种更易理解的方式来实现直线光栅化.算法在每一步都计算两个候选像素中点相对于理想直线的位置,如果中点在直线上方,说明直线更接近下方像素,反之则选择上方像素,这种基于几何关系的选择方式直观地保证了像素选择的最优性.

- 基本原理:
  - 使用中点判别函数
  - 根据中点位置决定像素选择

- 工作过程:
    1. 考虑当前像素 P(x,y)右边的两个候选像素
    2. 计算它们中点 M 的位置
    3. 判断 M 与理想��线的位置关系:
        - 如果 M 在直线上方,选择下方像素
        - 如果 M 在直线下方,选择上方像素
- 几何意义:
    - 通过中点到直线的距离判断
    - 保证选择的像素最接近理想直线
    - 本质上与 Bresenham 算法等价
- 特点:
    - 思想与 Bresenham 算法相近
    - 便于理解增量式判别思想
    - 可推广到圆和其他曲线的光栅化

```C++
void MidpointLine(int x1, int y1, int x2, int y2) {
    int dx = x2 - x1, dy = y2 - y1, d = dy - (dx / 2);
    int x = x1, y = y1;
    putPixel(x, y); // 绘制第一个像素
    while (x < x2) {
        x++;
        if (d < 0) d += dy;
        else {
            d += (dy - dx);
            y++;
        }
        putPixel(x, y); // 绘制像素
    }
}
```

2 着色器基础

2.1 着色器概念

- 定义: 着色器是运行在图形处理单元(GPU)上的小程序，用于控制图形渲染管线中的各个阶段。它们负责处理顶点、几何体和像素等数据，以生成最终的图像。
- 功能:
    - 顶点处理: 变换顶点位置、计算光照等。
    - 几何处理: 生成新的几何体、进行几何变换等。
    - 片元处理: 计算每个像素的颜色、处理纹理等。
- 编程语言: 常见的着色器编程语言包括 GLSL(OpenGL Shading Language)、HLSL(High-Level Shading Language, 用于 DirectX)、和 SPIR-V(用于 Vulkan)。
- 可编程性: 着色器提供了高度的灵活性，允许开发者自定义图形渲染效果，如实时光照、阴影、反射、折射等。

2.2 顶点着色器

- 职责:

- 顶点变换: 将顶点从模型空间(Model Space)转换到世界空间(World Space)、视图空间(View Space)和裁剪空间(Clip Space)。
- 法线计算: 计算和传递顶点的法线向量，用于后续的光照计算。
- 属性传递: 传递顶点属性，如颜色、纹理坐标等，供后续着色器使用。

- 输入:
  - 顶点属性: 位置(Position)、法线(Normal)、颜色(Color)、纹理坐标(Texture Coordinates)等。
  - 变换矩阵: 模型矩阵(Model Matrix)、视图矩阵(View Matrix)、投影矩阵(Projection Matrix)。
- 输出:
  - 裁剪空间位置: 用于光栅化阶段的顶点位置。
  - 顶点属性: 传递给几何着色器或片元着色器的属性。

2.3 几何着色器

- 职责:
  - 生成新几何体: 可以根据输入的几何体(如点、线、三角形)生成新的几何体，如额外的点、线段或三角形。
  - 几何变换: 进行复杂的几何变换，如法线细分、平滑、扩展等。
  - 特效生成: 实现几何体的动态效果，如爆炸效果、粒子生成等。
- 输入:
  - 基元类型: 顶点着色器输出的基元，如点(Points)、线(Lines)、三角形(Triangles)。
- 输出:
  - 新基元类型: 生成的基元类型，可以与输入基元类型相同或不同。
- 应用场景:
  - 几何体细分: 增加几何体的细节，提高渲染质量。
  - 动态几何体生成: 根据需要动态生成或修改几何体，如生成树枝、建筑结构等。

2.4 曲面细分着色器

- 职责:
  - 细分控制: 决定如何细分输入的几何体，控制细分级别和方式。
  - 细分评估: 计算细分后的新顶点的位置，生成更细腻的曲面。
- 组成部分:
  - 细分控制着色器(Tessellation Control Shader, TCS): 控制细分级别，决定每个细分区块的细分参数。
  - 细分评估着色器(Tessellation Evaluation Shader, TES): 计算细分后新顶点的位置，进行曲面评估。
- 应用场景:
  - 高精度曲面: 用于需要高精度和平滑曲面的场景，如角色模型、复杂地形等。
  - 动态细分: 根据视距或其他参数动态调整细分级别，提高渲染效率。

2.5 片元着色器

- 职责:
  - 颜色计算: 计算每个片元(像素)的最终颜色，包括光照、纹理、颜色混合等。
  - 纹理映射: 应用纹理图像到几何体表面，实现复杂的表面细节。
  - 光照模型: 实现不同的光照模型，如 Phong 光照、Blinn-Phong 光照、PBR(基于物理的渲染)等。
- 输入:
  - 插值属性: 从顶点着色器或几何着色器传递下来的插值属性，如颜色、法线、纹理坐标等。
- 输出:
  - 片元颜色: 最终显示在屏幕上的颜色值。
- 应用场景:
  - 真实感渲染: 通过复杂的光照和材质计算，实现高度真实感的图像。
  - 后处理效果: 实现各种后处理效果，如模糊、边缘检测、颜色校正等。

2.6 图形渲染管线

- 定义: 图形渲染管线是将三维场景转换为二维图像的过程，涉及多个阶段，每个阶段处理不同类型的数据，并在 GPU 上高效执行。
- 主要阶段:
  1. 顶点处理:
     - 任务: 处理顶点数据，进行变换和光照计算。
     - 涉及的着色器: 顶点着色器。
  2. 图元装配:
     - 任务: 将顶点组装成基本图元，如点、线、三角形。
  3. 几何处理:
     - 任务: 对图元进行进一步处理，如剔除、细分。
     - 涉及的着色器: 几何着色器(可选)。
  4. 光栅化:
     - 任务: 将图元转换为片元，准备进行像素级处理。
  5. 片元处理:
     - 任务: 计算每个片元的最终颜色，应用纹理、光照等效果。
     - 涉及的着色器: 片元着色器。
  6. 测试与混合:
     - 任务: 进行深度测试、模板测试，混合片元颜色，实现透明效果等。
- 渲染管线的可编程性:
  - 通过着色器程序，开发者可以在管线的不同阶段插入自定义的计算逻辑，实现各种视觉效果。
- 性能优化:
  - 并行处理: 管线的每个阶段都在 GPU 上并行执行，充分利用 GPU 的计算能力。
  - 管线效率: 通过减少不必要的计算和优化着色器代码，提高渲染效率。

2.7 着色器编程实例

为了更好地理解各类着色器的作用，以下提供一个简单的着色器编程实例，展示顶点着色器和片元着色器的配合。

顶点着色器示例(GLSL)

- 接收每个顶点的位置和颜色属性。
- 使用模型、视图和投影矩阵将顶点位置转换到裁剪空间。
- 将顶点颜色传递给片元着色器。

片元着色器示例(GLSL)

- 接收从顶点着色器传递过来的颜色。
- 将颜色赋值给片元的最终颜色输出，实现颜色插值效果。

渲染流程

1. 顶点着色器对每个顶点进行处理，计算其在屏幕上的位置，并传递颜色信息。
2. 光栅化将顶点组成的图元转换为片元。
3. 片元着色器对每个片元进行颜色计算，最终形成图像。

2.8 高级着色器技术

随着图形渲染需求的提升，着色器技术也不断发展，涌现出许多高级技术，提升图形质量和渲染效率。

1. 光照模型
   - Phong 光照模型: 基于反射定律，计算环境光、漫反射光和镜面反射光。
   - Blinn-Phong 光照模型: 对镜面反射进行了优化，通过计算半程向量来简化计算。
   - PBR(基于物理的渲染): 采用物理上准确的光照和材质模型，提供更真实的渲染效果。

2. 纹理技术
   - 多重纹理映射: 结合多个纹理，实现复杂的表面效果，如混合不同材质、法线贴图等。
   - 法线贴图(Normal Mapping): 使用法线贴图增加表面细节，模拟凹凸感而无需增加几何体。
   - 环境映射(Environment Mapping): 实现反射和折射效果，如镜面反射、水面反射等。
   - 纹理压缩: 通过压缩技术减少纹理内存占用，提高渲染性能。

3. 后处理效果
   - 抗锯齿(Anti-Aliasing): 减少边缘锯齿，提高图像质量。
   - 景深(Depth of Field): 模拟真实相机的景深效果，突出前景或背景。
   - 动态模糊(Motion Blur): 模拟快速移动物体的模糊效果，增强运动感。
   - HDR(高动态范围)渲染: 支持更宽的亮度范围，提升图像的亮部和暗部细节。
   - 屏幕空间反射(SSR): 实现基于屏幕空间的反射效果，提升反射的真实性。

4. 计算着色器(Compute Shaders)
   - 定义: 计算着色器是一种通用的着色器，用于执行并行计算任务，不限于图形渲染管线。
   - 应用场景: 物理模拟、图像处理、人工智能等。

5. 可编程几何着色器
   - 实例化: 在几何着色器中实例化图元，生成多个副本，节省内存并提高渲染效率。
   - 动态生成: 根据需要动态生成复杂的几何体，如草地、树木、粒子效果等。

6. 混合与透明度
   - 混合模式: 实现不同的混合效果，如加法混合、乘法混合、Alpha 混合等。
   - 透明度排序: 处理透明对象的渲染顺序，确保正确的视觉效果。
   - 双面渲染: 渲染几何体的双面，提高视觉效果，如叶片、薄膜等。

7. +着色器优化
   - 减少分支: 尽量减少条件判断，使用数学函数替代分支，提升性能。
   - 内存访问优化: 合理使用纹理缓存和共享内存，减少全局内存访问，提高数据访问效率。
   - 并行计算: 利用 GPU 的并行计算能力，优化算法以充分发挥硬件性能。