现代计算机图形学入门

---赖武功（dionysoslai@163.com）

现代计算机图形学入门
- 绪论
- 第一部分 光栅化
  - 概述
  - 第一章 渲染管线
    - 1. 名词解释
      - 1.1 图像渲染
      - 1.2 逐顶点渲染
      - 1.3 逐片元渲染
      - 1.4 光栅化
      - 1.5 CPU与GPU
    - 2. 总概
    - 3. 应用阶段
      - 3.1 数据加载
      - 3.2 渲染状态设置
      - 3.3 绘制
    - 4. 几何阶段
      - 4.1 顶点着色器
      - 4.2 图元装配
      - 4.3 曲面细分
      - 4.4 几何着色器
      - 4.5 裁剪
      - 4.6 屏幕映射
    - 5. 光栅化阶段
      - 5.1 三角形设置
      - 5.2 三角形遍历
    - 6. 像素处理阶段
      - 6.1 像素着色
      - 6.2 像素合并
  - 第二章 理论篇--基础数学
    - 1. 向量
      - 1.1 向量坐标表示
      - 1.2 点乘
      - 1.3 叉乘
    - 2. 矩阵
      - 2.1 性质与计算
    - 3. 插值计算
      - 3.1 线性插值
      - 3.2 双线性插值
      - 3.3 三角形插值
        - 3.3.1 邻近插值法
        - 3.3.2 距离插值法
        - 3.3.3 重心插值法
    - 4. 泰勒展开
      - 4.1 皮亚诺余项
        - 4.2 拉格朗日余项
    - 5. 蒙特卡洛积分
    - 6. 傅里叶积分
    - 7. 卷积
    - 7. 总结
  - 第三章 理论篇--基础变换
    - 1. 基础变换
      - 1.1 缩放变换
      - 1.2 旋转变换
      - 1.3 错切变换
      - 1.4 对称变换
      - 1.5 平移变换
    - 2. 三维变换

2.1 二维到三维拓展
2.2 法线变换
3. 矩阵分解与逆变换
3.1 矩阵分解
3.2 逆变换
4. 投影矩阵
4.1 正交投影矩阵
4.2 透视投影矩阵
4.2.1 frstum 到长方体变换
4.2.2 长方体正交变换
4.2.3 一般应用
5. 视口变换
6. 总结
第三篇 实践篇--基础数学内容和基础变换
1. 向量实现
1.1 四则运算实现
1.2 归一化与齐次坐标化
1.3 测试
2. 矩阵实现
2.1 四则运算实现
2.2 转置与逆变换
2.3 测试
第四章 理论篇--基础绘制
1. 线段绘制
1.1 基本思路
1.2 DDA(数值微分)画线算法
1.3 中点画线算法
1.4 Bresenham 算法
2. 三角形绘制
2.1 X扫描线算法
2.2 X扫描线算法改进
2.3 ToLeft 算法
3. 三角形网格
3.1 网格定义
3.2 三角形网格实现
4. 总结
第四章 实践篇--基础绘制
第五章 理论篇--缓冲与消影算法
1. 颜色缓冲
1.1 overdraw 问题（延伸）
2. 双缓冲技术
3. 消影算法
3.1画家算法
3.2 z-buffer 消影算法
3.3 Warnock 消影算法
第五章 实践篇--缓冲与消影算法
第六章 理论篇--反走样
1. 走样现象
2. 反走样技术
2.1 分辨率
2.2 平滑
2.2.1 非加权区域采样方法
2.2.2 加权区域采样方法
3. 走样原因（采样）
4. 傅里叶变换
4.1 原理
4.2 滤波器
4.2.1 低通滤波器

4.2.2 高通滤波器
5. 卷积
6 反走样技术
第六章 实践篇--反走样
第七章 理论篇--光照模型
1. 物体颜色
2. 基础光照模型
2.1 环境光照
2.2 漫反射光照
2.3 镜面光照
2.3.1 Phong 反射模型
2.3.2 Blinn-Phong 反射模型
3. 着色频率
3.1 平面着色（flat shading）
3.2 顶点着色（Gouraud shading）
3.3 像素着色（Phong shading）
第七篇 实践篇--光照模型
1. 最简单光照场景
第八章 理论篇--网格
1. 网格定义
2. 网格化
2.1 三角剖分
2.2 ear clipping 算法
2.3 T-vertices 算法
3. 网格细分
3.1 一维细分
3.2 细分步骤
3.3 细分算法
3.3.1 Loop 细分
3.3.2 $\sqrt{3}$细分
3.3.3 Catmull-Clark 细分
4. 网格简化
4.1 概念与原因
4.2 层次简化（LOD）
4.3 拓扑结构
4.4 简化机理
4.5 简化基本操作
4.5.1 顶点删除方法
4.5.2 边坍缩方法
4.5.3 面收缩方法
4.6 长方体滤波算法
4.7 基于相邻面片和边界的局部平坦性原则
4.8 Cohen 局部判别准则
4.9 渐进的网格简化技术
4.10 基于二次误差度量的简化技术
第八章 实践篇--网格
第九章 理论篇--纹理映射
1. 纹理简介
2. 纹理使用
2.1 纹理采集
2.2 纹理贴图
2.3 纹理滤波
3. 纹理采集
3.1 过程纹理
3.1.1 白噪声
3.1.2 柏林噪声
3.2 纹理合成
3.2.1 基于像素纹理合成

3.2.2 基于块纹理合成
4. 纹理映射
4.1 自然参数化
4.2 手工指定
4.3 网格参数化
5. 纹理渲染
5.1 纹理放大
5.1.1 临近过滤算法
5.1.2 双线性插值算法
5.2 多级渐进纹理
5.2.1 各向同性滤波
5.2.2 各向异性滤波
第九章 实践篇--纹理映射
第十章 理论篇--阴影技术
1. 阴影定义
1.1 阴影定义
1.2 阴影类型
1.2.1 附着阴影
1.2.2 投射阴影
1.2.3 自阴影
1.3 硬阴影
1.4 软阴影
2. 阴影重要性
3. 平面阴影
3.1 阴影投影
4. 曲面阴影
5. 阴影算法
5.1 阴影域（shadow volumn）
5.1.1 步骤
5.1.2 模板缓存
5.1.3 缺陷
5.2 阴影图（shadow map）
5.2.1 步骤
5.2.2 优点与缺点
6. 阴影艺术
6.1 阴影壳
第十章 实践篇--阴影技术
第十一章 理论篇--裁剪
第十一篇实践篇--裁剪
第二部分 光线追踪
第三部分 曲线与曲面
第四部分 运动与模拟
推广
1. 颜色模型
2. 计算几何
3. 分形几何
4. 共形几何
附录

绪论

在讲实质性内容之前，请容许我谈谈两个对我影响很大的小故事，希望在本书中，也能贯穿他们。

故事一：这是我在读初中在《读者》杂志上看到的一篇小短文，短文中大概描述了这样一段历史：爱因斯坦刚刚从校园走向社会时，在经济方面颇为窘困，因此需要找一份工作维持生活。后来，爱因斯坦选择当一名家庭教师。他是这样在广告中写道：我将会为您的孩子解决学习上的困惑，对于孩子学到的内容，保证能够做到**完全理解。**

对于这段故事，很多细节我已经不是很清楚，甚至很肯是杜撰而来的历史。但，这并不是妨碍对我的影响：**完全理解。**我希望，本教材对于图形学入门介绍，对大家也能够完全理解目的，每个人能够有一个完整的、自上而下的全局把握，在大方向做到提纲挈领，在小知识做到心中有数。

故事二：这个并不能算是一个故事。而是清华大学邓俊辉教授在他算法课程上提到一句话。邓教授在讲解DFS 算法时，提到了下面一个图：

这是一个逻辑排序中：出度+DFS的问题，在课上邓教授提到这样一句话：我们很多人在大一的时候，都会在思考大一阶段我们需要学哪些内容，掌握哪些技能。启示正在聪明的人，是会先思考大四那一年我们需要学习什么，出了校园之后我们需要掌握哪些内容。以终为始，倒推前面我们应该需要学习的知识。

这的确是一个很聪明的方法，只有明白自己需要什么，才有动力或者方向才是正确，学习最初的内容。因此，本教材也会采用这个方法，每一大部分都会列一个出度图，贯穿整个知识体系。

为什么要写这本书呢?

- 理论与实践结合

目前，市场上其实并不缺少图形学相关书籍，比方虎书《Fundamentals Of Computer Graphics》、渲染圣经《Real-Time Rendering》，国内有清华大学张家广教授和胡事民教授编写《计算机图形学基础教程》等书籍，都是非常好的阅读材料。但这些书籍都明显有一个问题，理论内容偏重，缺乏实践即代码方面实现。

图形学相对其他学科来讲，有一个很明显的特点：理论和实践深入结合。通过具体实践，我们才会加深理论方面理解与推广。

- 初学者入门材料混乱

初学者往往会将图形学与图形学API混乱，将OpenGL、DirectX、Valkun与图形学内容挂钩。但实际图形学并不等于OpenGL，OpenGL等API是对图形学内容的实现与接口封装。作为初学者并不适合将这些API作为入门材料。

另一方面，当前游戏引擎非常完善，比方现在非常热门的Unity和Epic 引擎。初学者也往往将引擎学习当做图形学入门材料。这也无可厚非，学习一个游戏引擎，较小很快，我们可以快速的实现一款游戏。但游戏引擎是一个非常综合体系，图形渲染只是其中一块内容，初学者往往不能很好区分开来，陷入知识沼泽中。

- 为了更多志同道合朋友

很认同闫令琪大神一个观点：**教不为了让你了解这门学科，而是希望大家一起来研究它**（大概是这个意思，具体说法我需要再去了解下）。我也需要，能够有更多朋友通过本书，从此打开一扇新的窗口，并一起致力于将这门学科发展壮大。大家一起**Make the Graphic Great Again!**.

- 更好入门图形学

本人很喜欢计算机图形学这块内容，不过我本科就读的是电子类专业内容，在本科阶段，并不是很了解这门学科，甚至对于CS专业都没怎么接触过。毕业之后，阴差阳错之下，开始从事游戏开发相关工作，从这之后慢慢接触图形学相关内容，慢慢开始热爱它。但由于先天缺陷吧，入门曲线相当陡峭，很多时候对相关内容云里雾里。

因此，我希望你能够通过本书，尽量轻松、平滑入门图形学，花更少时间完全更多事情。时间是一个贵重的礼物。

本书目的与特点

- 目的

作为图形学入门教材，本书会尽可能覆盖图形学方方面面内容，用一些通俗语言将基本概念和知识点说清楚。用前面提到话来说，希望大家对这块内容能够完全理解，对图形学有一个完整的、自上而下的全局把握，大方向做到提纲挈领，小知识做到心中有数。

另一方面，我希望本教材能够成为你的第一本图形学入门书籍，启蒙读物。通过本书，尽量让学习曲线平滑，后期可以无缝衔接《Fundamentals Of Computer Graphics》、《Real-Time Rendering》等书籍。同时，也为你接下来OpenGL、DirectX、Unity等内容学习，提供扎实理论基础。

- 特点

由于图形学本书特点：理论与实践深入结合。因此，本教程将分为理论篇和实践篇两个部分。

理论篇章节，将侧重对理论方面讲解。比方第四章 基础绘制章节，我们将会重点讲个线段绘制算法、三角形绘制算法、网格内容。对一些实现具体问题并不会很深入讲解。

实践篇章节，则侧重具体代码实现。同样第四章 基础绘制章节，在代码实现时，我们首先就会讲解SDL库导入和基本操作问题。然后，在实现线段绘制算法时，我们就会看到一些细节问题。比方当斜率k不存在时，我们要如何处理。

同时，有些理论的实现时，是需要一些前提约数的。比方我们坐标系是使用左手法则还是使用右手法则。比方我们标准化坐标范围是$[-1,1]$还是$[0,1]$，屏幕坐标的原点是左下角标记为$[0,0]$还是右上角标记为$[0,0]$等问题。这些具体细节，在理论篇中不会花很大篇幅讲解，会放在实践篇中进行具体讲述。

另一方面，有些内容跟理论其实没什么关系。比方工具使用，例如三角形网格实现中，会提到obj文件读写格式问题。这些内容，也均会放在实践篇中提到。同时，并不是每一章内容，都对应有实践篇，比方第二章内容，由于实际内容过少，就不单独拎出来，而是跟第三章合在一起讲。

本书大纲

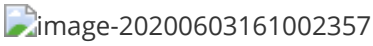
这块内容，留着后续内容完善再详细阐述。

第一部分 光栅化

概述

光栅化是一种将几何图元变为二维图像的过程，也是目前3D游戏成像主流算法，覆盖市面上几乎99%的游戏。这几年光线追踪技术发展很快，但依然还需要一段时间才能达到效率上的要求。相比来说，光栅化的计算速度与就比光线追踪快非常多，不过由于它并非基于对物理光线的传递计算，因此对于现实中复杂的光照效果的真实模拟方面，光栅化有点无能为力，特别是在阴影这块处理，显得不那么真实。

下图，大概列了一下光栅化知识体系整体出图，由于部分内容串接比较多，只能大概标下顺序。



光栅化大章节，我们也将按照这个图，依次讲解各个部分内容。不过有一点不同，我们将会在第一章，讲解目前图形API OpenGL、DirectX的渲染管线内容。由上到下给大家剖析当前图形整个渲染流程。如果这块内容，你看完之后还是对很多内容感到疑惑，可以跳过，后续你学习了OpenGL或者DirectX可以在回头看这块内容。

同时，我们不会讲一些很基础内容，比方矩阵基础运算、图形学含义、成像设备介绍等都不会讲述（主要是由于工作原因，时间不充裕，也许哪天我就会将这块内容补充完毕）😟

第一章 渲染管线

作为光栅化第一章节内容，我们并不像其他书籍一样，概述一下图形学内容，而是跟大家讲讲当前图形API整个渲染过程。本章会涉及，渲染整个流程，最初数据准备到后续顶点着色、片段着色器，再到最终像素合并阶段。同时，会适当的引用一些代码示例，让大家了解更加清楚。

1. 名词解释

在具体内容讲解之前，我们首先对一些基本名词进行解释，这些名词的理解有助于大家对后续内容理解。

1.1 图像渲染

我们最初准备的场景和实体一般都采用三维形式表示，这样更接近于现实世界，便于操纵和变换，而图形的显示设备大多是二维的光栅化显示器和点阵化打印机。因此，我们需要将这个由三维组成的的场景，转换为一个二维图像。在这其中会涉及一系列操作，这些操作我们就合称为图像渲染。

1.2 逐顶点渲染

游戏中图像是以点、线段、三角形等基本图元组成，每个图元都包含固定顶点数量，比方三角形有3个顶点组成。其中每个顶点数据，不仅仅包含坐标信息，同样包含颜色、纹理、法线等一些基本信息，这些信息合集我们称之为顶点数据(vertex data)。

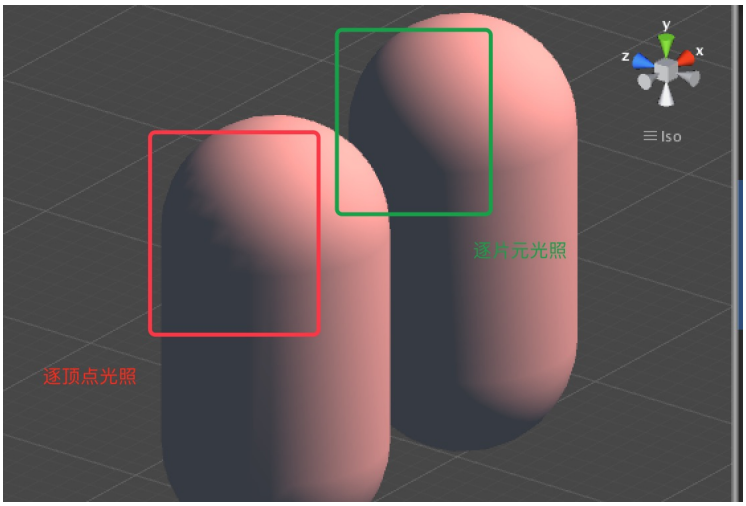
我们对这些顶点数据统一在顶点着色器中进行线性插值操作，从而得到每个片元的相关数据，这一过程我们称之为逐顶点渲染（per-vertex shading），也称为 Gougraud shading。这块内容在第七章第三节中会详细阐述。

1.3 逐片元渲染

首先，我们要区分片元与像素，二者并不相同。一个片元并不是真正意义上的像素。一个片元包含了很多状态的集合，这些状态包括屏幕坐标、深度信息、顶点信息等内容，这些信息都用于计算每个像素的最终颜色。

相对于逐顶点渲染数据全部在顶点着色器中计算；逐片元渲染的数据会在两个不同类型着色其中计算。我们将在顶点着色器中，对顶点位置和法线数据进行线性插值计算后，再在片元着色器进行光照（颜色）计算的这一过程称之为逐片元渲染（per-fragment shading)，也称之为 Phong shading。同样，这块内容会在第七章第三节中会详细阐述。

相对来说，逐片元渲染比逐顶点渲染，可以提高渲染的真实感，图像更加真实柔和。下图展示二者渲染不同：



1.4 光栅化

光栅化也可以叫成删格化、像素化。通俗点就是将矢量图形转化成像素点过程（或者说将三维空间中的几何形体显示到到屏幕的过程。如下图所示：将一个圆形图元栅格化过程。

image-20200414115139252

光栅化过程主要是2个两个部分工作：

1. 计算图元覆盖哪些像素（即计算窗口坐标系中哪些整形栅格区域被图元占用）；
2. 为这些像素计算它们的颜色和深度值；

总结起来，可以用一句话概括：光栅化就是产生片元过程。

1.5 CPU与GPU

图像渲染过程中，大部分都是在GPU上进行，这是因为GPU 渲染速度比CPU速度快非常多。那为什么GPU渲染速度会比CPU快呢？这里需要简单讲下，方便对后续具体渲染内容理解。

CPU和GPU差别非常大，是由于二者最开始设计目标就不相同。CPU需要很强的通用性来处理各种不同的数据类型，同时各种逻辑判断会引入大量的分支跳转和中断的处理。这些都使得CPU的内部结构异常复杂。而GPU面对的则是类型高度统一的、相互无依赖的大规模数据和不需要被打断的纯净的计算环境。

因此二者呈现出完全不同的架构图，如下所示：其中绿色的是计算单元，橙红色的是存储单元，橙黄色的是控制单元。

image-20200414141558721

可以看出来，CPU有大量的缓存单元，当计算单元并不是很多。相对而言，GPU则有大量的计算单元和非常简单的控制单元，并且省去了Cache。因此，GPU很适合一些重复计算内容，但不适合逻辑运算，这也是为什么写shader时，不建议使用if ... else ... 语句。

总体来说，以下类型的程序适合在GPU上运行：

- 计算密集型的程序。所谓计算密集型(Compute-intensive)的程序，就是其大部分运行时间花在了寄存器运算上，寄存器的速度和处理器的速度相当，从寄存器读写数据几乎没有延时。可以做一下对比，读内存的延迟大概是几百个时钟周期；读硬盘的速度就不说了，即便是SSD, 也实在是太慢了。
- 易于并行的程序。GPU其实是一种SIMD(Single Instruction Multiple Data)架构， 他有成百上千个核，每一个核在同一时间最好能做同样的事情。

图像渲染过程，正好就是这种类型数据，因此非常适合在GPU上运行（原先GPU设计，就是为了渲染图像，现在在深度学习方面也大量运用GPU计算）。

2. 总概

自从20世纪初，福特工程发明流水线生产之后，在工业上，流水线就被广泛应用，大大提高了生产力。一个典型流水线图，如下所示:

从图中，可以看出每个流水线阶段都做到分工明确、简化流程等特点。因此，采用流水线设计总体来说具有高效率、低成本、交易量大的优点：

- 分工合作，简化流程，降低复杂度；
- 效率高，每个阶段专注各自阶段事情；

在现代化图像渲染技术中，同样采用流水线概念进行设计。具体流水线图，如下所示：（这里图有问题，后续改下，同时还需要对图线颜色区别，确认具体阶段可编程、可配置、固定区别信息）

图中可以看到，图形渲染管线包含很多部分，其中每个部分的输入参数都是上一阶段的输出参数，其输出参数都是下一个阶段的输入参数。根据渲染管线各个阶段内容，我们分为大概四个大阶段：应用阶段、几何阶段、光栅化阶段、像素处理阶段。其中应用阶段在CPU中进行，几何阶段、光栅化阶段和像素处理阶段均在在GPU中进行。

3. 应用阶段

应用阶段是唯一一个在CPU中运行的阶段，主要任务：是把需要显示在屏幕上的几何信息输入GPU中，GPU再进行具体渲染工作。具体任务大致可以分为三个内容：

1. 准备场景数据，数据包括场景中的模型、光源、各种贴图信息等；
2. 粗粒度剔除工作，剔除不可见物体，减少非必要计算，提高GPU效率；这个阶段常用的剔除算法有：视锥体剔除（Frustum Culling）和遮挡剔除（Occlusion Culling）

   这里有个地方需要注意的是，剔除和裁剪的区别；

   剔除：将整个物体数据移除，不发送给GPU渲染；

   裁剪：物体并不一定全部在视锥体外面，即物体一部分在视锥体内，一部分在视锥体外部。这个阶段工作发生在GPU上。常见的算法有Cohen-Sutherland算法、中点分割法、Liang-Barsky算法等。

   二者主要区分在于粒度不同。
3. 设置渲染状态；
4. 输出图元信息，调用绘制命令；

总之，应用阶段的主要任务是准备好场景数据（包括场景中的模型、光源等等）；剔除不可见物体，减少不必要的计算；设置物体的渲染状态（物体使用的材质、shader等等）。最终，应用阶段将输出渲染图元（用于渲染物体的几何信息），并传递给几何阶段。

3.1 数据加载

这个阶段实质是将：需要渲染数据加载到显存中。这是由于实际渲染工作是在显卡中，显卡内部访问显存更快，而且实际上大多数显卡是不能直接访问系统内存的。

具体流程：从硬盘中读取数据——>加载到内存中，经过一些列数据操作（对有一款游戏来说，这里数据操作实际上就是游戏主要逻辑了）——>将网格和顶点数据加载到显存中。

数据加载到显存后就可以从系统内存中移除掉了，但是对于我们后续需要cpu访问的数据（如网格数据会用于碰撞检测）可能会继续保留，因为从硬盘加载这些数据也是很耗时的。

在具体的图形API中，我们以OpenGL为例，讲解操作具体流程。

OpenGL中通过顶点缓冲对象（Vertex Buffer Objects，VBO）管理这些内存，它会在显存中储存大量顶点。使用VBO好处是我们可以一次性的发送一大批数据到显卡上，而不是每个顶点发送一次。另外从CPU把数据发送到显卡相对较慢，所以只要可能我们都要尝试尽量一次性发送尽可能多的数据。当数据发送至显卡的内存中后，顶点着色器几乎能立即访问顶点，这是个非常快的过程。

下面是一个具体VBO才做例子：

```
float firstTriangle[] = {
    -0.9f, -0.5f, 0.0f,  // left
    -0.0f, -0.5f, 0.0f,  // right
    -0.45f, 0.5f, 0.0f,  // top
};

unsigned int VBOs[1], VAOs[1];
glGenVertexArrays(1, VAOs);
glGenBuffers(1, VBOs);
// 第一个三角形
glBindVertexArray(VAOs[0]);
// 把顶点数组复制到缓冲中供OpenGL使用
glBindBuffer(GL_ARRAY_BUFFER, VBOs[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(firstTriangle), firstTriangle,
GL_STATIC_DRAW);
// 设置顶点属性指针
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
[...绘制物体]
```

首先，firstTriangle对象中保存了三角形三个顶点具体位置信息（对应从硬盘中读取数据到内存中），然后通过VBO将数据加载到GPU中（这里由于简单示例，并没有对数据进行操作），最后就是绘制物体过程。

3.2 渲染状态设置

通过前面数据加载阶段，这时GPU已经获得信息包括网格和顶点数据（位置信息、法线信息、光照信息等），这些信息已经足够渲染一个物体了，但渲染结果要么不够细致、要么不够真实、不够正确。因此，GPU还需要其他额外信息，比方材质、纹理信息等。

因此，渲染状态就是**指示GPU如何去渲染一个物体**，例如指定一个三角形采用什么样着色器，是否开启半透明，使用哪种材质，使用哪个纹理等等。当这些设定都准备好之后，CPU就可以发送一个命令通知GPU使用指定的状态来渲染给定的数据。

我们以一段OpenGL代码示例，具体看看是如何操作的。

```
// 创建2种shader(每种shader中都包含一个顶点/片段着色器)
Shader shader1("v1.vs", "f1.fs");
Shader shader2("v1.vs", "f2.fs");
[....数据操作]
while (!glfwWindowShouldClose(window))
{
    [...开始渲染]
```

```
    // 第一个设置状态和绘制
    glActiveTexture(GL_TEXTURE0);// 采用texture1
    glBindTexture(GL_TEXTURE_2D, texture1);
    shader1.use();// 设置使用当前绘制采用第一种着色器
    glBindVertexArray(VAOs[0]);//使用第一个顶点数据
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);// 绘制

    // 第二个设置状态和绘制
    glActiveTexture(GL_TEXTURE1);// 采用texture2
    glBindTexture(GL_TEXTURE_2D, texture2);
    shader2.use();// 设置使用当前绘制采用第二种着色器
    glBindVertexArray(VAOs[1]);//使用第二个顶点数据
    glDrawElements(GL_TRIANGLES, 3, GL_UNSIGNED_INT, 0);

    // 开启双缓冲技术
    glfwSwapBuffers(window);
    glfwPollEvents();
}
```

通过这段代码，我们可以清楚看到设置渲染状态是如何工作的。同时有一点我们需要重点讲清楚，所谓设置渲染状态阶段，重点要理解**状态**。简单字面理解就是，图线渲染是由一系列状态机控制，如果将A状态设置为某一值，后续操作没有改变A状态操作，这后续渲染全部采用该值。这个很直观，比方我们一般在渲染初始就设定了是否要开启深度测试，而不会每次绘制指定一遍深度测试开启状态。

在具体图形API中，例如OpenGL是如何工作呢?

在OpenGL中，本身就是一个巨大的状态机(State Machine)：一系列的变量描述OpenGL此刻应当如何运行。这个状态通常被称为OpenGL上下文(Context)，即GL-Context。GL-Context 是 GPU和OpenGL的桥梁，在整个程序中，一般只有一个GL-Context 。在前面代码中VBO将顶点信息存放在显存中，GPU渲染时去显存中读取数据，二者的桥梁就是GL-Context。

在使用OpenGL时，经常会遇到一些状态设置函数(State-changing Function)，例如开启深度测试函数glEnable(GL_DEPTH_TEST)，这类函数就是改变GL-Context的值；以及状态使用函数(State-using Function)，这类函数会根据当前GL-Context执行一些操作。

同时GL-Context也负责在一个渲染流程中不同绘制命令进行状态切换。例如前面代码中2个glDrawElements绘制命令。

3.3 绘制

当一切数据和状态都设置完毕，CPU将发起绘制命令，通知GPU开始渲染图像，例如上面提到glDrawElements绘制命令。这个阶段也是CPU参与渲染的最后一个阶段，GPU参与渲染的入口。

需要注意的是，绘制命令仅仅指向一个需要被渲染的图元（primitive）列表，具体其他信息在上一个阶段已经完成了。例如：

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);// 绘制
```

glDrawElements是一个OpenGL绘制命令，原型如下：

```
void glDrawElements( GLenum mode,
    GLsizei count,
    GLenum type,
    const void * indices);

mode
```

```
    指定那种类型图元需要被渲染。图元包括GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP,
GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRIANGLES, GL_QUAD_STRIP,
GL_QUADS, and GL_POLYGON

count
    指定渲染元素数量。
type
    指定数据类型，类型包括 GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT, or GL_UNSIGNED_INT.

indices
    指定数据在偏移量。
```

在这一阶段，CPU和GPU之间并不是上下游串行工作方式，而是利用命令缓冲区（Command Buffer）技术实现并行工作方式。

命令缓冲区包含一个命令队列，有CPU像其中添加命令，GPU从中读取命令，添加和读取是完全分开相互独立的，从而实现CPU和GPU并行工作。当CPU需要渲染图像时，将渲染指令添加到命令队列中；而当GPU完成了上一次绘制任务之后，继续从命令队列中读取下一个渲染指令，完成接下来绘制任务。

在命令队列中，不仅仅包含绘制指令（draw call），还包含其他指令，比方改变渲染状态等。具体可以如图所示：

在这一过程中，绘制指令（draw call）属于比较耗时阶段。因为在CPU通知GPU绘制之前，需要先准备很多内容，包括数据，状态，命令等，其中重新设置材质/shader是一项非常耗时的操作。另一方面，对于GPU渲染能力非常强（可以看前面关于GPU介绍），渲染300个网格和3000个网格没有什么区别。

因此，实际上当渲绘制指令（draw call）过多时，会出现GPU一直在等待，CPU花费大量时间在提交Draw call 上，造成CPU的过载。从而导致整个渲染效率低下，这也是游戏引擎中，渲染优化过程中，draw call 是一个重要参数原因。

既然，提交Draw Call是导致渲染低下的一个原因，那么很显然一个优化想法是把很多小的Draw Call合并成一个大的Draw Call，这就是批处理（Batching）思想。

在游戏引擎中，一般都会提供batch接口，例如cocos2dx 中SpriteBatchNode，就是批处理运用。这里大概讲下原理：

```
// SpriteBatchNode
bool SpriteBatchNode::initWithTexture(Texture2D *tex, ssize_t capacity/* =
DEFAULT_CAPACITY*/)
{
    [...]
    _textureAtlas = new (std::nothrow) TextureAtlas();
    _textureAtlas->initWithTexture(tex, capacity);
    [..]
    return true;
}

// TextureAtlas
bool TextureAtlas::initWithTexture(Texture2D *texture, ssize_t capacity)
{
    [...]
    // 分配capacity空间
    _quads = (V3F_C4B_T2F_Quad*)malloc( _capacity * sizeof(V3F_C4B_T2F_Quad) );
    _indices = (GLushort *)malloc( _capacity * 6 * sizeof(GLushort) );
    memset( _quads, 0, _capacity * sizeof(V3F_C4B_T2F_Quad) );
```

```
    memset( _indices, 0, _capacity * 6 * sizeof(GLushort) );
    [...]
    setupVBOandVAO();
    [..]
    return true;
}

void TextureAtlas::setupVBOandVAO()
{
    // 创建、绑定VAO
    glGenVertexArrays(1, &_VAOname);
    GL::bindVAO(_VAOname);
#define kQuadSize sizeof(_quads[0].bl)
    // 创建 2个缓存对象，一个是VBO，一个EBO
    glGenBuffers(2, &_buffersVBO[0]);
    // VBO 用来存储数据（顶点数据、颜色、纹理）
    glBindBuffer(GL_ARRAY_BUFFER, _buffersVBO[0]);
    glBufferData(GL_ARRAY_BUFFER, sizeof(_quads[0]) * _capacity, _quads,
GL_DYNAMIC_DRAW);
    [...指向顶点数据、颜色、纹理]
    // EBO 用来存储顶点索引
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, _buffersVBO[1]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(_indices[0]) * _capacity * 6,
_indices, GL_STATIC_DRAW);
    [...]
}
```

通过这段代码，我们可以很清楚看到batch工作原理：将绘制多个sprite指令统一在一个指令中，即绑定在一个VAO内部。同样，我们可以看到batch的限制必须是同一个纹理。

以上通过引擎coco2dx对batch技术进行稍微讲解。在unity中，支持4种Batch操作，Static Batching(静态合批)，Dynamic Batching(动态合批)，GPU Instance， SRP Batch。具体原理，可以看相关内容。

同时，由于batch是在CPU中进行的，这个过程涉及到malloc、memset 内存分配等操作，存在一定耗时问题，因此batch程度需要一个折中考虑。同时，对于静态物体（即静态合批），一般我们只需要合批一次即可；但对于动态物体（即动态合批），需要在每一帧都重新合批，这对CPU是一定负担。

额外多提一些内容，这里讲的Draw Call，实际上指的是Direct Rendering，渲染数据全部来至于CPU，还有Indirect Rendering 和 Conditional Rendering 2种渲染方式。这两种方式的渲染数据有些是来自于GPU传过来的。

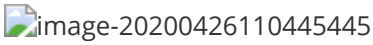
Indirect Rendering方式运行运行直接渲染GPU中的数据，省去GPU->CPU->GPU的拷贝过程。

Condition Rendering运行在某些条件下渲染，在某些条件不渲染，经典运用就是Occlusion Query。

这块内容，需要加深了解，可以参考openGL中的drawCall类型。

4. 几何阶段

从几何阶段开始，渲染的实现都在GPU中进行，通过一系列流水线化工作，大大加速渲染过程。在几何阶段中，我们将具体5个步骤（细分为顶点着色器、曲面细分着色器、几何着色器、裁剪、屏幕映射）看成一个整体，那么输入就是CPU提供的一系列顶点数据，输出是具体三角形顶点数据。如图所示：



具体来说，几何阶段主要负责变换三维顶点（模型变换、视图变换等）、光照计算（着色、纹理映射等）和最后屏幕相关裁剪、映射（投影变换等）工作。最后，输出是得到了经过变换和投影之后的顶点坐标、颜色、以及纹理坐标。

在几何阶段，我们一定要牢记:显示屏是二维的，GPU 所需要做的是将三维的数据，绘制到二维屏幕上，并到达"跃然纸面"的效果。

4.1 顶点着色器

顶点着色器的主要是工作是完成：顶点着色和坐标变换。具体来说是将一个单独的顶点作为输入，然后对其进行坐标变换，得到另外一种3维坐标，同时我们还可以对顶点属性进行一些基本处理（光照计算、纹理映射等）。

顶点着色器有以下几个特点：

1. 本身不会创建和销毁任意一个节点；
2. 顶点之间相互独立，无法得到2个顶点之间的关系；

由于这些特点，GPU可以并行化处理每一个顶点，所以顶点着色器阶段处理速度非常快。

下面给出一段OpenGL的简单顶点着色器代码：

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec2 aTexCoord;
out vec3 vertexColor;
out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // MVP变换
    gl_Position = projection * view * model* vec4(aPos, 1.0f);
    TexCoord = aTexCoord;   // 输出纹理坐标
};
```

示例中，顶点位置进行MVP变换之后，正确摆放到世界坐标中，同时输出纹理坐标。对应应用阶段部分代码如下：

```
    float vertices[] = {
        -0.5f, -0.5f, -0.5f, 0.0f, 0.0f,
        0.5f, -0.5f, -0.5f, 1.0f, 0.0f,
        ....
    };

    unsigned int VBO, VAO, EBO;
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);
```

```
    glBindVertexArray(VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    // position 属性，即aPos
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float),
(void*)0);
    glEnableVertexAttribArray(0);
    // texture coord 属性，即aTexCoord
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)(3
* sizeof(float)));
    glEnableVertexAttribArray(1);
    ...
```

4.2 图元装配

图元装配这一过程，是将顶点根据之前设定的图元（即连接关系），还原成对应图元形状（网格结构）。在OpenGL中，可以支持的图元包括：

1. 点：GL_POINTS；
2. 线段：独立线段 GL_LINES、线段间首尾相连但最终不闭合 GL_LINE_STRIP、线段间首尾相连最终封口闭合 GL_LINE_LOOP；
3. 多边形：按点的定义顺序依次连接 GL_POLYGON ；
4. 三角形：从第1个点开始，每三个点一组画一个三角形，三角形之间是独立的 GL_TRIANGLES、从第三个点开始，每点与前面的两个点组合画一个三角形，即线性连续三角形串 GL_TRIANGLE_STRIP、从第三个点开始，每点与前一个点和第一个点组合画一个三角形，即扇形连续三角形 GL_TRIANGLE_FAN；
5. 四边形：从第1个点开始，每四个点一组画一个四边形，四边形之间是独立的 GL_QUADS、每点与前面的三个点组合画一个四边形，即线性连续四边形串 GL_QUAD_STRIP

其中，最主要运用的是三角形，这是因为三角形具有非常好的性质：

1. 没有不存在凹凸问题；
2. 三个顶点必然在同一个平面；
3. 多边形中顶点数量最好；
4. 具体非常多的良好几何性质：比方重心坐标等
5. 。。。

4.3 曲面细分

这块内容，后续补上。

4.4 几何着色器

几何着色器是一个可选着色器，其输入是上一阶段的图元装配输出，即一个图元的一组顶点（完整的图元数据），例如三角形，就是其3个顶点。几何着色器通过产生新顶点构造出新图元，是原来图元发生改变，例如一个三角形可以变成两个三角形。

基本来说，几何着色器的主要功能如下：

1. 改变原有图元，生成新的图元；
2. 生成更多的顶点；

其中，几何着色器的一个重要应用是用来显示物体的法线，可以帮助我们调试光照效果。具体方法是我们首先在不使用几何着色器的情况下正常渲染一次场景；然后开启几何着色器第二次渲染场景，送到几何着色器的是三角形图元，我们为其每个顶点生成一个法线向量。然后绘制法线向量即可。例如下图所示：

关于几何着色器大概内容，就介绍到这里，后续如果有更好的思路继续更新补充（这块内容，我也不太熟悉 😥）

4.5 裁剪

裁剪（Clipping）的目的是为了减少渲染分支，提高效率。在实际项目中，我们场景很有可能会很大，超出屏幕空间（或者说摄像机的视野范围）。这时，会了提升渲染速度，我们可以将视野范围外面的物体不进行处理。

具体来说，一个物体和摄像机关系，只有三种关系：

1. 完全在摄像机视野范围内；
2. 部分在摄像机视野范围内；
3. 完全不在摄像机视野范围。

第1种情况，这个阶段直接跳过；第3种情况，属于剔除功能，这个阶段也不处理；第2种情况才是裁剪阶段需要处理的。对于裁剪和剔除有不懂，具体可以看之前应用阶段内容。

最原始的裁剪算法是：点的裁剪。原理非常简单，就是判断一个点是否在视野范围内：

$$
\begin{gathered}
\text{对于任意一点} P(x,y)\text{，若满足下列公式：} \\
\begin{cases} x_{left} \le x \le x_{right} \\ y_{bottom} \le x \le y_{top} \end{cases} \\
\text{则} P\text{点在窗口内部，否则在窗口外部。}
\end{gathered}
$$

点的裁剪算法要求计算场景中的所有顶点，效率很低，都不采用这个方式。后续改进裁剪算法有：

1. 直线裁剪算法
    1. Cohen-Sutherland算法；
    2. 中点分割法；
    3. Liang-Barsky算法
2. 多边形裁剪；
3. 文字裁剪

这些算法的具体内容，将会在后面裁剪章节重点阐述。

4.6 屏幕映射

到这个阶段，目前所有顶点坐标仍是三维坐标下坐标（范围在单位立方体内），这时需要完成三维坐标转换到二维屏幕上的工作，屏幕映射就是做一个过程。这个过程，需要进行一次视口变换，变换到屏幕坐标中。这块内容，可以查看第三章 变换章节。

这里有一点需要注意的是，在屏幕映射过程中，由于z并不会参与，因此对z坐标不做任何处理。最终生成的屏幕坐标和z坐标构成一个坐标系，我们称之为：窗口坐标系（Window Coordinates)。这些值会一起被传到下一阶段——光栅化阶段。

同时，由于不同屏幕分辨率不一样，实际呈现出来的画面跟分辨率有很大关系。具体到游戏中，就是我们常说的屏幕适配这一个问题。在常见的游戏引擎中，都会提到这个问题，一般大概有：宽高自适应拉伸、按高比例拉伸、按宽比例拉伸等好几种方面。具体采用哪些方案，根据游戏情况可以自行选择调整。

5. 光栅化阶段

在前面名词解释部分，就已经提到光栅化概念：将顶点数据转换为片元的过程。通俗来说，就是根据前面阶段的输出数据，即一系列顶点数据x、y（这里我们只讲坐标数据，并且由于是在二维屏幕上，因此不考虑z坐标），如何在屏幕上进行绘图。

总的来说，光栅化从根本上来说，就是要解决2个问题：

1. 如何光栅化（绘制）一条直线；
2. 如何光栅化（绘制）一个三角形；

当然最基础是如何光栅化一个点，这是一个非常简单的问题，这里没必要单独拿出来讨论。至于其他类型图元绘制问题，都可以从这2个问题进行延伸。同时，我们需要注意一点，这里提到的绘制一个三角形，实际是做了2个方面操作：a. 判断三角形覆盖了哪些像素；b. 这些像素需要如何着色（计算其颜色值）。

如何绘制一条直线，有三个常用的直线绘制算法：

1. 数值微分法（DDA）；
2. 中点画线法；
3. Bresenham算法；

这些内容，会在第四章 基础绘制 章节详细阐述。我们重点讨论关于三角形内容（在实际模型中，都是采用三角形网格进行处理）。

5.1 三角形设置

三角形设置是光栅化第一个阶段内容。接收的数据是从几何阶段最后一个部分内容屏幕映射传进来内容：屏幕坐标系下的顶点位置以及和它们相关的额外信息，如深度值（z坐标）、法线方向、视角方向等。这个阶段会计算光栅化一个三角网格所需的信息。

具体来说，上一个阶段输出的都是三角网格的顶点，即我们得到的是三角网格每条边的两个端点。但如果要得到整个三角网格对像素的覆盖情况，我们就必须计算每条边上的像素坐标。为了能够计算边界像素的坐标信息，我们就需要得到三角形边界的表示方式。这样一个计算三角网格表示数据的过程就叫做三角形设置。它的输出是为了给下一个阶段做准备。

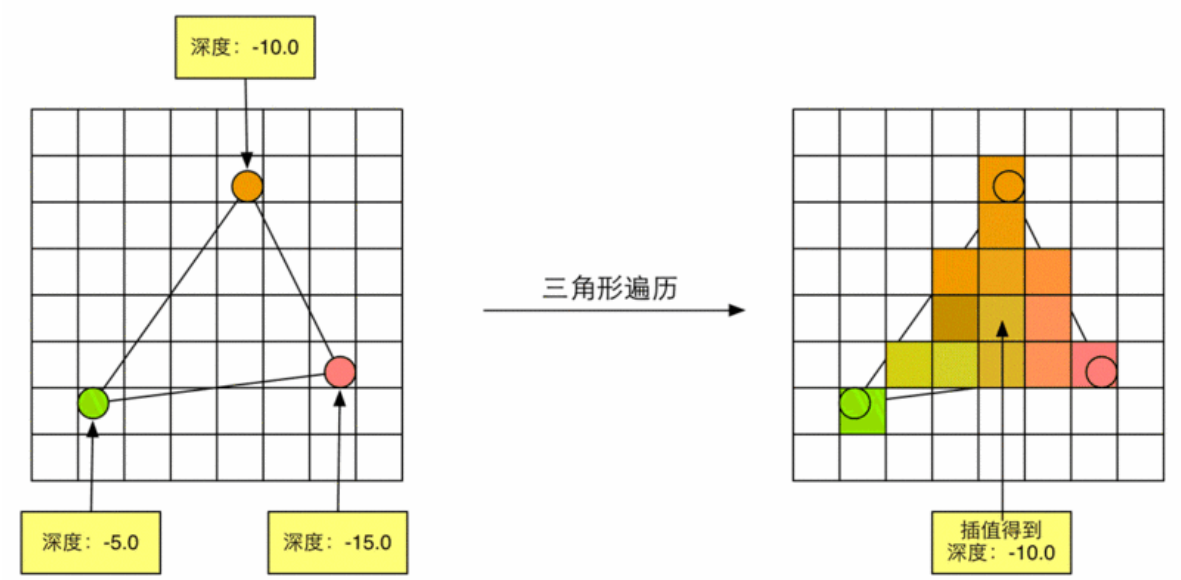
5.2 三角形遍历

在讲三角形遍历之前，我们先回顾之前提到如何绘制一条直线，介绍了常用的三个算法。那么绘制一个三角形呢？推广开来，就是如何绘制一个多边形。这里我们需要介绍什么是多边形扫描转换和一个转换算法——X扫描线算法。

通俗来说，在前面我们表示一个三角形（多边形）都是采用顶点形式。采用顶点形式，有直观、几何意义强、占内存少、易于进行几何变换等优化，但缺点是没有明确指出哪些像素在多边形内，这样对于绘制一个三角形（多边形）来说，完全没法处理。因此我们需要将顶点形式转变成点阵形式。这个也是光栅化的一个基本问题。从顶点形式转换成点阵形式，这这种转换称之为多边形扫描转换。

X扫描线算法基本思想是按扫描线顺序，计算扫描线与多边形的相交区域，再用要求的颜色显示这些区间的像素，即完成填充工作。其中区间的断点可以通过计算扫描线与多边形边界线的交点获得。当然，对于三角形来说，判断一个点是否在三角形内部，可以采用向量中点乘方式来实现。这些都是额外话题，具体可以自行深入了解。这块内容，也将会第四章 基础绘制 章节详细阐述。

到这里，我们就对三角形遍历有了一个直观认识了，所谓三角形遍历就是检查每一个像素是否被一个三角形网格覆盖，如果覆盖就生成一个片元。通常我们也称这个阶段为扫描变换。。



这一步的输出就是得到一个片元序列。需要注意的是一个片元并不是真正意义上的像素，而是包含了很多状态的集合，这些状态用于计算每个像素的最终颜色。这些状态包括了但不限于它的屏幕坐标，深度信息，以及其他从几何阶段输出的顶点信息，例如法线，纹理坐标等。

6. 像素处理阶段

到这一步，已经是图线渲染最后一步：像素处理，或者称之为着色阶段。经过前面一系列操作，我们已经得到一个三角形内的所有像素（片元）。接下来我们需要正确处理这些片元，使之得到正确的渲染。

6.1 像素着色

像素着色是像素处理第一个阶段，处理对象是每一个片元，使用的数据则是从顶点着色器中输出的数据插值得到的（常用的插值算法是三角形重心算法）。输出是一个或者多个颜色值。

这里需要注意一点的是，像素着色在不同图形API中叫法不大一样，比方在OpenGL叫做片元着色器（Fragmeent Shader），在DX中称为像素着色器（Pixel Shader）。

这一个阶段可以做很多渲染技术。其中我们经常谈到的纹理采样就是在这一个阶段处理。例如下面代码：

```
#version 330 core
out vec4 FragColor;
in vec3 vertexColor;
in vec2 TexCoord;

uniform sampler2D texture1;
uniform sampler2D texture2;

uniform float mixValue;

void main()
{
    FragColor = mix(texture(texture1, TexCoord), texture(texture2,
vec2(TexCoord.x, TexCoord.y)), mixValue);
};
```

这里采样2个纹理texture1和texture2，并对纹理进行简单混合处理。具体纹理采样内容，后续讨论。

6.2 像素合并

现在，我们终于到最后一步：合并阶段。这一个阶段在不同的图形API中，依然采用不同叫法，在OpenGL中我们称为逐片元操作（Per-Fragment Operations），在DX中则称为输出合并阶段（Output-Merger）。综合理解就是：

1. 操作内容：合并一系列数据；
2. 操作对象：对每一个片元进行操作；

对于每一个像素来说，最终都存储在颜色缓冲区（color buffer，保存rgb具体数值）。每一个片元都过上一个阶段（像素处理）之后，产生对应颜色值，则新的颜色值需要覆盖当前颜色缓冲区。

比如2个三角形互相覆盖，那么我们会先计算第一个三角形，并将所有像素颜色都存储在颜色缓冲区中；接下来计算第二个三角形（假设第二个三角形在第一个三角形前面，并且都是不透明的），第二个三角形的部分像素颜色将覆盖之前的颜色缓冲区（这里通过假设，我们已经默认覆盖合并操作）。

因此，总得来说，合并阶段，需要处理的问题包括合并哪些数据？进行哪些操作？具体任务有：

1. 决定每一个片元的可见性。包括深度测试、模板测试等。
2. 如果一个片元通过所有测试。则需要将新的片元颜色值与已经存储在颜色缓冲区的颜色进行合并。

需要注意的是，这里的合并不简简单单就是覆盖，还包括混合等操作（覆盖是混合的一种）。混合功能非常好用，在OpenGL中，常见的GL_ONE_MINUS_SRC_COLOR、GL_SRC_ALPHA等内容，可以实现已给黑板檫、刮刮乐等有趣功能。当然这只是小功能，更多内容可以自行了解。

在可见性判断中，需要经过一系列测试，只有通过测试，才能获得合并资格。不同的图形API最常见的测试有模板测试和深度测试。这里主要讲下深度测试原理。

类似颜色缓冲区（color buffer），深度测试同样采用深度缓冲区（depth buffer）技术。每一个片元进行深度测试时，都会与深度缓冲区对应的值进行比较，大于这个值时舍弃该片元（也可以设置小于时舍弃，这个用户可以自行定义）。如果该片元被保留下来，则会将新的片元深度值会覆盖原来深度值。当然我们设置决定是否开启或者关闭深度写入。这块内容，与透明效果关系非常密切。后续会继续讨论。

这个算法我们称之为Z-Buffer 算法，是有艾德.卡姆尔与1973年提出来的。

如果一个片元经过了一系列复杂操作之后，到最以后一步没能通过测试被舍弃了。那么前面的操作都是无效工作了。因此，提高GPU效率，我们可以尝试提前判断哪些片元会被抛弃。对于深度测试来说，如果我们提前在像素着色之前就做好深度测试，则会大大提高GPU效率，这一个技术成为Eary-Z。

不过，Eary-Z技术也有几个明显的问题，这块内容，我后续补上。在后续实践篇中，也希望能补上Eary-Z技术的实现。

（哈哈，理想很丰满，也许后面现实很骨感😢）

第二章 理论篇--基础数学

The cleaner the math, the cleaner the resulting code.

这一句话，摘自《Fundamentals of Computer Graphics》，本人特别喜欢，也将它送给你。这句话直面翻译就是：数学越简洁，生成的代码也就越简洁。当然也可以这样解释，我们对数学越了解，对代码编写也就越有把握。

在图形学中，会运用大量数学相关内容，在后续章节中，大家可以深刻体会到。不过，本教材用到的数学内容，相信大家在本科教育中都会接触到，属于一些基础数学内容。比方向量、矩阵知识，比方几何三角形等。同时，这里我们假定你对数学已经有了基本了解，因此我们不会介绍一些过于基础内容，比方三角函数，求解一元二次方程式解法和韦达定理，常用微积分求解等。同时，向量和矩阵内容也是点到为止，只会介绍我们要用到内容（还是时间不够原因，也许后面我就会补全，哈哈😁）

1. 向量

通常单纯一个数，我们称为标量，如果给这个数加一个方向，那么我们就叫做向量（vector）。向量通常用一个黑体或者箭头表示，例如**v**或者$\vec{v}$，向量长度我们用单竖线表示：$|a|$，单位向量用一个头上帽子表示：$\hat{a}$。向量有行向量和列向量区别，图形学中，通常采用列向量表示，例如：

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

接下来对向量的讨论均是在二维平面上，三维或者更高维推而广之即可，如果有不同的，会特殊提出。

1.1 向量坐标表示

对于任何一个向量，都可以用不同方向的2个向量表示出来（注意，这里我们讨论的是二维向量），例如：

$$\vec{c} = m\vec{a} + n\vec{b} \tag{1.1}$$

 image-20200507171007080

根据这个特性，我们可以将**a**和**b**特殊处理，使二者互相垂直，并且数值上特殊处理，即$\vec{a}=(1, 0)^T$、$\vec{b}=(0,1)^T$。这就是我们常说的的笛卡尔坐标系。这样，对于任何一个向量，可以如下表示：

$$\vec{c} = x_a\vec{x} + y_a\vec{y} \tag{1.2}$$

我们将这种表示，称之为向量的坐标表示方法。

1.2 点乘

2个向量互相相乘会得到一个标量（没有方向的数），我们称为点乘（注意与叉乘相区别）。点乘公式如下：

$$\vec{a} \cdot \vec{b} = |a||b|cos\theta \tag{1.3}$$

 image-20200507172706935

用坐标系表示的话，对应公式如下（可以自行推导）：

$$\vec{a} \cdot \vec{b} = x_a x_b + y_a y_b \tag{1.3}$$

采用点乘方式，我们可以很方便求一些值，比方用来求2个向量夹角、一个向量在向量上的投影大小。

$$\begin{aligned} cos\theta &= \frac{\vec{a} \cdot \vec{b}}{|a||b|} \\ b_{b->a} &= |b|cos\theta = \frac{\vec{a} \cdot \vec{b}}{|a|} \end{aligned} \tag{1.5}$$

同时点乘满足常见交换律和结合律：

$$\begin{aligned} \vec{a} \cdot \vec{b} &= \vec{b} \cdot \vec{a} \\ \vec{a} \cdot (\vec{b} + \vec{c}) &= \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c} \\ (k\vec{a}) \cdot \vec{b} &= \vec{a} \cdot (k\vec{b}) = k\vec{a} \cdot \vec{b} \end{aligned} \tag{1.5}$$

在图形学中，点乘具有很多很好性质，这里举一个应用：用来判断2个向量同向性，但点乘>0，表示同向；反之，则表示反向。但无法判断两个向量左右问题。而这个问题采用叉乘可以解决。

1.3 叉乘

跟点乘不同，2个向量叉乘的结果还是一个向量，通常用$\times$表示。叉乘的结果是一个3D向量，该向量垂直于前面2个向量所在的空间，同时，叉乘满足以下性质：

$$|\vec{a} \times \vec{b}| = |\vec{a}||\vec{b}|sin\theta \tag{1.6}$$

用坐标系表示，则对应公式如下：

$$\vec{a} \times \vec{b} = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b) \tag{1.7}$$

采用点乘方式，我们统一可以很方便求一些值，比方2个向量围城平行四边形面积，垂直于2个向量的第三个向量。

叉乘有有一些常见数学性质，跟点乘有个区别是不满足交换律，因此后续我们讲到基础变换时，变换的顺序很重要。常见数学性质，有这些：

$$\begin{aligned} \vec{a} \times (\vec{b} + \vec{c}) &= \vec{a} \times \vec{b} + \vec{a} \times \vec{c} \\ \vec{a} \times (k\vec{b}) &= k(\vec{a} \times \vec{b}) \\ \vec{a} \times \vec{b} &= -(\vec{b} \times \vec{a}) \end{aligned} \tag{1.7}$$

点乘可以快速判断2个向量同向性问题。叉乘则快速判断2个向量左右问题，这个在计算几何中是一个非常基础问题：ToLeft。这块内容，在附录：计算几何会重点讲到。

2. 矩阵

矩阵是向量的推广，一系列向量的有序集合，并且赋予一些确定的数学规则，我们称之为矩阵，例如

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

在图形学中，矩阵具有非常多的运用，下一章基础变换内容，基本上都是一些矩阵问题。在这里，我们需要做2个基本的假定：

1. 矩阵中的数字都是实数，不讨论复数情况；
2. 矩阵的行和列个数一样。

2.1 性质与计算

2个矩阵互相相乘，计算起来稍微比较复杂，我们这里直接给出公式，具体如果所示

$$p_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj} \tag{1.8}$$

 image-20200507200610817

这里，我们需要注意下，矩阵不满足交换律，即$AB\neq BA $，这个跟叉乘不满足交换律一样。同时如果存在$AB \neq AC$，并不一定说明$B=C$。但矩阵满足结合律和分配率：

$$\begin{aligned} (AB)C &= A(BC) \\ A(B+C) &= AB + AC \\ (A+B)C &= AC + BC \end{aligned} \tag{1.9}$$

矩阵内容其他内容，例如转置矩阵和反矩阵就不做阐述了。还有其他内容看后续有空再讲吧。

3. 插值计算

插值计算被广泛用到图形学中，是一个很重要的技术。后续提到反走样、纹理计算、运动等内容，均会看到插值计算。比方已知三角形三个顶点颜色，内部顶点颜色是多少，才能让效果看起来更加平滑、柔和，就涉及到插值计算。

3.1 线性插值

线性插值可能是图形学中用到最多的一种数学特性。假设有a、b2个顶点，那么ab中任意一个顶点，则可以表示如下：

$$p = (1-t)a + tb \qquad t \in [0,1] \tag{2.1}$$

这个就是线性插值公式，相信我们在很多地方会看到这个公式。

现假定a、b 2个顶点是采用坐标轴表示，则可以得到斜率$k=\frac{y_j-y_i}{x_j-x_i}$，对应插值公式如下：

$$f(x) = y_i + \frac{x - x_i}{x_j - x_i}(y_j - y_i) \tag{2.2}$$

这里，我们不考虑斜率不存在特殊情况。如果，x代表是坐标，y代表是rgb值，则根据公式我们可以计算线段ab之间每一点之间的颜色值，从而得到一条光滑的线段。这里具体实现，可以看第四章 基础绘制 的实践篇。

3.2 双线性插值

双线性插值是线性插值的二维延伸，用来计算二维网格中随机位置的值。如图所示：

首先，我们通过线性插值得到a和b的值，在通过a和b的线性插值得到c的值。值得注意的是，先计算横轴还是先计算数轴，对最终结果都没有影响。

当然，在三维方面的延伸，就是三线性插值（Trilinear），原理跟双线性插值没有任何区别，这里就不多做解释。大家可以自行查阅资料。这块内容的运用，会在今后第九章 纹理渲染章节具体提到运用。

3.3 三角形插值

三角形插值计算，基本原理是已知三角形三个顶点，计算三角形内部点的值。有常见邻近插值法、距离插值法和重心插值法，其中以重心插值应用最广泛。效果最好。

3.3.1 邻近插值法

邻近插值法是最简单一个方法，原理很简单：对于三角形内部一个点p，判断距离三个顶点距离，p的值（颜色、纹理等）采用距离最近的顶点值。具体效果如下：

该方法，计算量最简单，但效果比较差强。比较适合小三角形，一旦三角形过大，效果会很差。如果，模型网格密度很大，我觉得到时可以尝试一下。后续网格，会讲到网格细化内容，到时我们尝试一下效果。

3.3.2 距离插值法

距离插值法是邻近插值法的推广，不再简单考虑一个顶点的影响，而是综合三个顶点影响，具体原理是：根据内部点p距离三个顶点距离大小，进行分别权重判断，公式如下：

距离三个顶点分为为：P_1、P_2、P_3，权重分别为：$w_1 = 1/P_1 \ldots$

则p的值为：

$$p = \frac{w_1 a + w_2 b + w_3 c}{w_1 + w_2 + w_3} \tag{2.3}$$

距离插值法具体效果如下：

效果明显好于邻近插值法，完全达到可用步骤。但对于一些尖角三角形，效果并不是很理想，问题比较多，例如下图所示：

我们原本想要用V1和V3的颜色来表示点P的颜色，但是由于V2距离点P最近，所以点P 大部分的颜色值来自于点P，这是我们不想看到的，因此有时候距离插值法也被称为天真插值法。😥

3.3.3 重心插值法

重心插值法是利用了三角形一个性质：对于三角形内部任意一点，都可以用三个顶点表示出来

$$p = w_1 a + w_2 b + w_3 c, \text{其中} w_1 + w_2 + w_3 = 1;$$

因此，转换为坐标方式，我们都可以用以下公式表示：

$$\begin{gathered} x = w_1 x_a + w_2 x_b + w_3 x_3 \\ y = w_1 y_a + w_2 y_b + w_3 y_3 \\ w_1 + w_2 + w_3 = 1; \end{gathered}$$

简化之后，我们可以得到：

$$\begin{aligned} w_1 &= \frac{(y_b - y_c)(x - x_c) + (x_c - x_b)(y - y_c)}{(y_b - y_c)(x_a - x_c) + (x_c - x_b)(y_a - y_c)} \\ w_2 &= \frac{(y_c - y_a)(x - x_c) + (x_a - x_c)(y - y_c)}{(y_b - y_c)(x_a - x_c) + (x_c - x_b)(y_a - y_c)} \\ w_3 &= 1 - w_1 - w_2 \end{aligned} \tag{2.4}$$

这里需要注意的是，当点P在三角形外部时，w1，w2，w3中至少有一个值是负数。这也很方便判断某一点P是否在三角形内部，如果在外部，可以直接抛弃。

重心插值法，效果如下：

可以看到效果还是很不错的，对一些特色三角形也不会有距离插值法问题。因此，重心插值法的运用最为广泛。三角形插值计算方法还有其他一些方式，比方说面积法，这里不多做介绍。读者可以自行查阅一些资料。

4. 泰勒展开

在讲泰勒展开之前，先讲一个画画例子。艺术家在做一副画作时（这里，我们假定是要画出一个真实现实场景），首先都需要打个底稿，底稿通常就是一个简单的线条描绘和框框之类的，目的是将模型大概形状勾勒出来。然后艺术家会对线条进行丰富，对框框进入柔化等工作，目的则是细化模型，使之更像现实中的物体。最后艺术家就会给模型上色，结果就是看起来像一个真实物体。

这里，我知识粗糙讲一下画画例子，实际可能比这个复杂多。不够，我并不是跟大家将如何画一个化，而是告诉大家一个数学核心思想：仿造。

当我们想要仿造一个东西的时候，无形之中都会按照刚才提到的思路，即先保证大体上相似，再保证局部相似，再保证细节相似，再保证更细微的地方相似……通过不断的细化下去，将仿造品无限接近于真品。

这个思想太重要了，不管是数学、物理还是其他学科，很多工作都是依照这个思路来进行。

我们现在要讲的泰勒展开则是这一个思想的运用。

在实际中有很多函数，比方三角函数$y=sinx$，具有很多良好性质，比方说不断重复，0到1范围，可以无限求导等。但具体计算某一个值时，就相当的麻烦。比方计算$y=sin2$的值，相信很多人一下子计算不出来，即使采用计算机计算，如果没有采用合适公式也很难一下子求出结果。这时，我们就考虑能不能用其他函数来替代计算。

那么如何仿造函数$y=sinx$呢?

首先，我们先保证起点相同（加上起点在x=0处）；

再保证增减性形同，就是一阶导数一致；

然后保证凹凸性相同，就是二阶导数一致……等等，就这样无限仿造下去，最终结果应该就会无限接近。

所以我们认为：仿造一段曲线，同样方式，先保证起点相同，再保证在此处导数相同，继续保证在此处的导数的导数相同……。这样无限下去，我们可以仿造任何一条去下。现实中，我们最终不可能无限仿造下去，当我们仿造到一定程度是，只要这时误差在允许范围内，就可以停止仿造，认为当前的仿造曲线可以替代原先曲线了。

到此为止，我们给出具体的泰勒展开公式（具体推导比较简单，可以自行查阅）：

$$f(x)=f(x_0)+\frac{f^1(x_0)}{1}(x-x_0)+\frac{f^2(x_0)}{2!}(x-x_0)^2+\ldots+\frac{f^n(x_0)}{n!}(x-x_0)^n \tag{4.1}$$

4.1 皮亚诺余项

时间有限，后面再继续补充。

4.2 拉格朗日余项

时间有限，后面再继续补充。

5. 蒙特卡洛积分

在讲什么是蒙特卡洛积分之前，先看下面一张图：（图片摘自ttp://www.scratchapixel.com/）

 image-20200604151217940

要计算$F(x)=\int_{a}^{b}f(x)dx$的值，如果$f(x)$的积分比较好计算，使用积分公式可以将结果很好的计算出来。但现实生活中，很多模拟出来的公式比较复杂，单纯的运用数学积分计算则是一个几乎不可能的事情，例如后面讲到的渲染方程式。因此，我们同样可以借鉴仿造思想，我们使用其他方式来代替积分运算。

积分$F(x)=\int_{a}^{b}f(x)dx$可以看做是曲线$f(x)$包围在a到b之间的面积值，加入我们取a到b中将的一个值x，那么$F_1(x)=f(x)(a-b)$，显然$F_1与F(x)$的取值相差会很大。但是，如果我们多取一些a到b的值，明显可以将这个结果的误差不断优化下去。

蒙特卡洛积分就是采用这个思想。蒙特卡洛积分也称统计模拟方法，是一种计算机化的数学方法，通俗来说，就是在求积分时，如果找不到被积函数的原函数，我们是无法得到积分结果的。但采用蒙特卡洛积分，利用一个随机变量的被积函数进行采样，并将采样值进行一定处理。当采样数量很高时，得到的结果我们可以近似认为就是积分结果。

蒙特卡洛公式如下：

$$F(x) = \int_a^b f(x)dx \approx \frac{b-a}{N}\sum_{i=1}^{N} f(x_i) \tag{5.1}$$

这里，我们假定任意取值x在a到b之间的概率是同样的。

蒙特卡洛方法可以说是整个离线渲染最核心的基础，这是因为离线渲染的核心方程渲染方程是一个（理论上无限）高维积分，我们不可能通过分析的方法求解。

我们对蒙特卡洛积分的理解到这里就可以，按我的经验是足够应对后续的图形学内容了。

6. 傅里叶积分

傅里叶分析可能是仅次于蒙特卡洛方法的数学工具，它广泛运用于过滤，平滑，以及数据压缩，它也是小波分析和球谐函数的重要基础。如同一系列正交的坐标轴（如(0,0,1), (0,1,0), (1,0,0)）构成一个矢量空间，然后该空间中的任何矢量都是这组正交基矢量的线性组合一样，如果一系列的函数是相互正交的（即它们每两个函数的乘积的积分值均为0，除非两个函数相同，此时积分值为1），则这些函数构成一个函数空间，因此该函数空间的任意一个函数都可以表述为这些基函数的线性组合，傅里叶变换使用的正余弦函数正是这样的一组无限个数量的基函数，因此这就是为什么任意函数都可以进行傅里叶变换，傅里叶变换的本质就是将一个函数转换为所有这些无限个正余弦函数的线性组合，这些线性组合的系数是由一个称为投影的方法计算的，该方法的算法就是通过求原始信号函数与某个基函数乘积的积分值，理论上我们记住这无数个线性组合的系数就可以重合合成为原始的信号函数，这个过程称为重建。所有这些投影形成的系数形成一个连续的函数，它反映的是原始信号函数的频率分布，如果你去掉该频率分布中的某些区间，你就丢掉了原始信号对应的频率部分，通常对于一个方差比较大的采样结果，我们可以丢弃掉频率高的部分，它们通常都是导致方差较大的罪魁祸首，因此函数的高频部分需要更多的样本，而由于计算资源限制我们通常都无法满足这样的条件。你会发现一个在频率域对频率分布进行带限的函数，在原始的时间域或空间域变成了一个平滑函数，这也是通过对这个带限函数本身进行傅里叶逆变换得到的，这就是过滤的思想。你还会发现在频率域计算两个函数的乘积变成了时间域或空间域对一个数组（例如所有像素组成的一个图像）元素中的每个元素都执行一次过滤计算，这就是卷积的概念。

时间有限，后面再继续补充。

7. 卷积

时间有限，后面再继续补充。

7. 总结

图形学中用到的数学内容非常多，这里我们只不过是做一个非常粗浅的介绍，希望能够尽量的涵盖后续要讲到的数学内容。不过由于篇幅和本人懒的原因，对一些基础内容篇幅非常少，只是蜻蜓点水般带过，这个跟其他图形学书籍差别很大（哎，主要还是打字太累了，平时要上班没有那么多时间）。

第三章 理论篇--基础变换

变换是我们在学习图形学中首先要遇到的第一个问题，属于非常基础的内容，关系到后面很多问题的开展，因此对于变换内容我们很有必要理解它。

在学习变换之前，我们需要明白：为什么我们要学习变换？

假设，我们很容易画出一个位置在原点，长度为1的正方形，如果要画一个任意位置，长度为m、n的长方形，我们该如何比较方便处理呢？

第一个思路，就是在该位置重新画一个长度为m、n的长方形。这样处理，理解起来最浅显易懂，但实际相对比较麻烦，因为长方形的位置和长度都不一样，还可能有旋转等需要，如果发现变形，可能完全就无法采用长方形方式绘制。

第二个思路，利用最初正方形，我们经过一定操作，将正方形变成要求的长方形，步骤如下：

1. 将长和宽分别缩放到m、n倍；
2. 将位置挪到特定位置；

这里操作，我们统称为变换。

因此，我们为什么需要变换？因为变换可以做到：模块化，将复杂几何图形拆成简单已处理图形。

在本章内容中，我们首先从基本的变换讲起，慢慢过渡到复杂变换，最后详细MVP变换的数学公式推导。其中，对于一些简单变换矩阵，我们就直接给出公式了，就不做推导了（哈哈，还是时间原因😅）

1. 基础变换

根据矩阵基础知识，我们知道一个2-向量和2*2矩阵相乘，相对应公式如下：

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix} \tag{1.1}$$

可以很清楚看到，结果还是一个2-向量。在数学中，我们将这种2-向量和2*2矩阵相乘得到另一个2-向量，称为线性变换。根据矩阵的不同，可以得到各种基础变换矩阵。

1.1 缩放变换

缩放变换可以改变向量长度和方向。对应矩阵如下表示：

$$scale(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \tag{1.2}$$

缩放矩阵作用于一个向量，公式如下：

$$\begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} s_x x & 0 \\ 0 & s_y y \end{bmatrix}$$

当缩放矩阵 scale(0.5, 0.5)，缩放效果如下：

1.2 旋转变换

在处理旋转矩阵之前，我们需要先做一个假定：绕着坐标原点旋转。这是因为，绕着不同点旋转同一角度，结果完全不一样。这里我们做一个简单化处理。跟缩放变换不一样，旋转变换并不会向量长度，只改变向量的方向。

根据下图示例，我们可以得到以下2个等式：

$$\begin{gathered} x_a = rcos\alpha \quad y_a = rsin\alpha \\ x_b = rcos(\alpha + \phi) = rcos\alpha cos\phi - rsin\alpha sin\phi = x_a cos\phi - y_a sin\phi \\ y_b = rsin(\alpha + \phi) = rsin\alpha cos\phi + rcos\alpha sin\phi = y_a cos\phi + x_a sin\phi \end{gathered}$$

由此，我们可以得到旋转矩阵公式：

$$rotate(\phi) = \begin{bmatrix} cos\phi & -sin\phi \\ sin\phi & cos\phi \end{bmatrix} \tag{1.2}$$

1.3 错切变换

错切变换是一个比较特殊变换，只会在一边发生变换（二维是边，三维则是面）。这里，我们可以想象：给定一个矩形，我们在矩阵一边用力推，那么矩形将变换成平行四边形，这个变换就是错切变换。2个典型图形如下：

由于错切变换，有一边位置发生变换，另一边位置不动。因此，我们可以很容易给出错切变换公式：

$$\begin{aligned} shear - x(s) &= \begin{bmatrix} 1 & s \\ 0 & 1 \end{bmatrix} \\ shear - y(s) &= \begin{bmatrix} 1 & 0 \\ s & 1 \end{bmatrix} \end{aligned} \tag{1.3}$$

这里，s表示形变程度，当然我们也可以用角度表示，假设与x(y)轴夹角为ϕ，则公式如下：

$$\begin{aligned} shear - x(s) &= \begin{bmatrix} 1 & tan\phi \\ 0 & 1 \end{bmatrix} \\ shear - y(s) &= \begin{bmatrix} 1 & 0 \\ tan\phi & 1 \end{bmatrix} \end{aligned} \tag{1.4}$$

1.4 对称变换

对称问题，算是老生常谈话题了。我们在初中就经常会学到一些轴对称、x轴对称等内容。因此，在处理对称变换之前，我们也需要先做一个假定，对称轴是x轴或者y轴，而不是任意一条直线。不同假定，给出公式差别很大。这里，我们直接给出如下公式：

$$\begin{aligned} reflect - x &= \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ reflect - y &= \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \end{aligned} \tag{1.5}$$

对称示例如下：

1.5 平移变换

平移变换是基础变换中最特殊的一个变换了。我们无法按照之前方式给出一个平移变换矩阵。这是由于：

$$\begin{aligned} x^{'} &= x + x_t \\ y^{'} &= y + y_t \end{aligned}$$

原先矩阵无法满足以上公式。因此我们引入第三维数据，用3*3矩阵来表示平移矩阵。对应，平移矩阵公式如下：

$$translate(x_t, y_t) = \begin{bmatrix} 1 & 0 & x_t \\ 0 & 1 & y_t \\ 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

同时，矩阵需要改成3-矩阵，这样我们可以很方便表示点和方向。其中$[x \quad y \quad 1]^T 表示一个点，[x \quad y \quad 0]^T表示一个方向$。在实践篇中，相信你就会看到，我们表示一个Vector3f，用的是x、y、z、t四个分量。

通过平移变换，我们引出2个常用概念：

1. 仿射变换；
2. 齐次坐标系；

仿射变换，又称仿射映射，是指在几何中，对一个向量空间进行一次线性变换并接上一个平移，变换到另一个向量空间。

齐次坐标系，则是为了平移变换而增加额外维度，通俗来说就是将一个n维向量用n+1维向量表示。这里，我们可以用《计算机图形学(OpenGL版)（F.S. Hill Jr)中一句话概括：“齐次坐标表示是计算机图形学的重要手段之一，它既能够用来明确区分向量和点，同时也更易用于进行仿射（线性）几何变换。”

2. 三维变换

前面提到的都是二维变换内容，但在实际中，我们模型和场景都是采用三维表示。因此，需要做一次简单二维到三维拓展。

2.1 二维到三维拓展

前面我们提到的变换都是在2维，实际上我们在应用时采用的是三维内容。因此，需要做一个2维到3维拓展过程。同时为了方便，我们采用其次坐标系来表示。

- 缩放矩阵公式：

$$scale(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

- 旋转矩阵公式：

$$\overset{\text{绕 } z \text{轴 旋 转}}{rotate - z(\phi) = \begin{bmatrix} cos\phi & -sin\phi & 0 & 0 \\ sin\phi & cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}} \tag{2.2}$$

$$\overset{\text{绕 } x \text{轴 旋 转}}{rotate - x(\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\phi & -sin\phi & 0 \\ 0 & sin\phi & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

$$\overset{\text{绕 } y \text{轴 旋 转}}{rotate - y(\phi) = \begin{bmatrix} cos\phi & 0 & sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -sin\phi & 0 & cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}}$$

旋转矩阵稍微比较复杂，其中绕y轴旋转跟绕x、z轴旋转的格式不一致，主要是因为采用左手坐标系原因。如果采用右手坐标系，则应该是另一个公式不一样，这个可自行验证。

- 错切矩阵公式：

$$shear - x(d_y, d_z) = \begin{bmatrix} 1 & d_y & d_z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

- 对称矩阵公式:

$$
\begin{aligned}
&xy\text{平面对称} \\
reflect-xy &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&\text{绕}yz\text{轴旋转} \\
rotate-yz(\phi) &= \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
&\text{绕}zx\text{轴旋转} \\
rotate-zx(\phi) &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned}
\tag{2.4}
$$

- 平移矩阵 公式:

$$
translate(x_t, yt_t, z_t) = \begin{bmatrix} 1 & 0 & 0 & x_t \\ 0 & 1 & 0 & y_t \\ 0 & 0 & 1 & z_t \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}
$$

其中，对于旋转矩阵，这里由于实际需要，给出绕任意角度旋转矩阵公式：

2.2 法线变换

法线向量是一个很特殊向量，在后续内容中，基本都会提到法线向量。同时，由于基本基础变换之后，变换后法线向量不再保有原来性质了，需要做一个特殊变换，才能继续保有原来性质。因此，这里有必要对法线变换进行特殊讲解下。具体现象如下图所示：

 image-20200518195622128

这里，经过简单错切变换，法线就不再是变换后的法线了。这是为什么呢？下面给出严格数学判断。

首先定义，向量$\vec{t}$，其法向量为$\vec{n}$，变换矩阵为M，假设变换后的向量和法向量依然垂直，则有如下等式：

$$\vec{n}^T t = 0$$

$$(M\vec{n})^T(M\vec{t}) = 0 => \vec{n}^T M^T M\vec{t} = 0$$

要使第二个等式成立，则$M^t M = I$

这个向量不可能。因此从数学上进行判否。

那么，法线变换需要如何处理呢？这个其实和上述证明类似：

首先定义，向量$\vec{t}$，其法向量为$\vec{n}$，变换矩阵为M，法线变换矩阵为N，我们要计算N，使得：

$$(N\vec{n})^T(M\vec{t}) = 0 => \vec{n}^T N^T M\vec{t} = 0$$

由于，$\vec{n}^T\vec{t} = 0 => \vec{n}^T It = 0 => n^T M^{-1} Mt = 0$

结合2个等式，我们可以得到：

$$N^T = M^{-1} => N = (M^{-1})^T$$

因此，我们可以得到法线变换矩阵公式:

$$N = (M^{-1})^T \tag{2.5}$$

3. 矩阵分解与逆变换

3.1 矩阵分解

如果我们处理一些简单变换问题，采用前面介绍变换矩阵还可以勉强用下，但一些涉及复杂变换问题，我们就需要更高的数学工具了。

很幸运的是，矩阵支持结合律性质，我们可以通过简单矩阵组合成复杂矩阵。例如，我们为了得到下面图像：

可以通过先缩放后旋转方式得到（当然也可以先旋转后缩放方式）。

同样道理，当我们拿到一个复杂矩阵时，通过分析，完全有可能将这个矩阵分解成多个简单矩阵的组合。例如上图所示矩阵：

$$\begin{bmatrix} .707 & -.353 \\ .707 & .353 \end{bmatrix} = \begin{bmatrix} .1.0 & 0 \\ 0 & .5 \end{bmatrix} \begin{bmatrix} .707 & -.707 \\ .707 & .707 \end{bmatrix}$$

这里有一点需要特殊我们需要。如果复杂矩阵M，可以分解成几个简单矩阵R、S、T。如果数学上的表示如下：

$$M = RST$$

那么依次表示先经过T变换，再经过S变换，最后进行S变换。变换从右到左进行。如果将顺序调换，则是新的矩阵变换。这是因为矩阵不满足交换律性质。

在矩阵分解中，有2种分解比较特殊：1. 对称矩阵；2. 奇异矩阵。这里暂不做介绍，等后续高级在仔细介绍这些内容；

3.2 逆变换

在数学上，矩阵通常都是可以进行逆变换。严格数学定义如下：

$$\begin{gathered} 对于n阶方阵A，如果存在一个n阶方阵B，使得： \\ AB = BA = E \\ 则A可逆（非奇异），B是A的一个逆矩阵。同时逆矩阵记做A^{-1} \end{gathered}$$

这让我们还原一个矩阵变换变得实际可行。

例如，缩放矩阵$scale(s_x, s_y, s_y)$，其逆变换为$scale(1/s_x, 1/s_y, 1/s_z)$；旋转矩阵的逆矩阵这是反方向旋转同样角度；平移矩阵逆矩阵是反方向移动相同距离。

对于一个复杂矩阵M，如果M矩阵可以分解成多个矩阵组合，则有如下公式：

$$\begin{gathered} M = M_1 M_2 M_3 \dots M_n \\ => M^{-1} = M_n^{-1} \dots M_3^{-1} M_2^{-1} M_1^{-1} \end{gathered}$$

4. 投影矩阵

通过前面变换，我们已经能够随便将一个物体变换到特定位置和形状，现在我们需要解决如何呈现问题。由于屏幕是一个2D平面，而物体处于3D空间中，因此物体变换之后也将变成2D形式。我们将这种3D到2D的映射变换称为viewing 变换。

viewing变换包括。。。和投影变换。投影变换，我们可以简单理解成照相机照相效果，具体分为正交投影矩阵和透视投影矩阵。

4.1 正交投影矩阵

正交投影矩阵在工业处用处广泛，是图像呈现中倒是比较少用，主要还是由于正交投影出来的画面不符合人眼呈现规律。对于正交投影，我们可以理想化假设一系列平行光线（阳光）向物体照射过去，在垂直于光线的墙面形成的投影效果。如下图所示：

 image-20200519172426316

很明显，物体经过正交投影之后，舍弃其中一个坐标（设置为0）而保留其他2个坐标不动。通常，我们将设置z坐标为0，因此正交投影矩阵表示如下：

$$P_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.1}$$

对于任意向量$(x,y,z)^T$，等式变换如下：

$$P(x,y,z)^T = (x,y,0)^T; \text{可以看到} x, y\text{坐标保留}, z\text{坐标舍弃为} 0$$

同时，由于$|P_0|$=0，说明P是一个不可逆变换矩阵（行列式=0）。因此，物体经过正交投影变换（3D到2D过程）之后，我们没法将进行还原。简单理解，就是1维数据丢失了。

使用这个正交投影矩阵，我们将所有点（z大于、小于0）都会映射到投影平面上。然而，现实中我们需要将z值（或者x值，y值）限定在一个确定范围内，因此我们介绍另一种正交矩阵。

一种更通用正交投影可以用一个长方体表示，如下图所示：

长方体6个方向包围在(l,r,b,t,n,f)之中，分别代表左边、右边、底板、上边、近面与远面，很明显，我们可以通过简单平移和缩放将长方体变换成一个单位立方体，如下图所示：

在这里，我们假定采用右手坐标系法则（有些书籍可能会采用左手坐标系），同时假定：

$$l < r, b < t, n > r, \text{这里需要注意} z\text{轴朝向相机，因此离相机越近，值反而越大；}$$

转换得到的单位立方体我们称为：规范视域体（canonical view volume），坐标系称为：归一化设备坐标轴（NDC，normalized device coordinates。转换成规范化视域体之后，对后面裁剪非常有效，因此这一步骤很重要。

物体经过变换到规范视域体之后，不在立方体之内的点都可以舍弃，渲染时只需要考虑立方体内点即可。这个正交变换矩阵公式如下：

$$P_0 = S(s)T(t) = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

具体效果，我们暂时采用unity表示：

4.2 透视投影矩阵

相对正交投影，另一种投影：透视投影在图形学中运用的更加广泛，这是由于其成像效果更符合现实生活。到这里，之前平行光线不在平行了，而是发散射出去（这些光线源头可以认为都在一个点上）。

透视投影推导过程，我们可以用下图进行表示：

1. 将平截头体frustum压缩成一个长方体；
2. 长方体进行正交投影计算；

其中，近平面z = 0，远平面z = f;

4.2.1 frstum 到长方体变换

将平截头体frustum压缩成一个长方体时，我们需要规定3个原则：

1. 近平面所有点坐标不变；
2. 远平面所有点坐标z值不点都是f；
3. 远平面中心点左边不变(0,0,f)；

有相似三角形定律（下图所示）可以得出，对于frustum内任意一点(x,y,z,1)，压缩以后坐标应该为(nx/z,ny/z,unknow,1)。

 image-20200520154012192

其中z坐标未知。即我们可以得到下面变换矩阵：

$$P = M(x,y,z,1)^T = (nx/z, ny/z, ?, 1)\ (nx, ny, ?, z)$$

这里我们根据向量性质，做了一个简单变换。你那么其中矩阵M第一行，我们有如下等式：

$$Ax + By + Cz + D = nx => \text{明显可以得到} A = n, B = 0, C = 0, D = 0$$

同样方式，我们最终可以得到下列等式：

$$M = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

剩下第三行目前未知，因此需要带入前面提到的3个原则。

首先，由于近平面所有点左边不变原则，我们有如下等式：

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{bmatrix} (x,y,n,1)^T = (x,y,n,1) ==>$$
$$nx = x; ny = y; n = 1;$$

显然对这个等式，只有当n=1时，等式才成立，因此依然根据矩阵性质，将等式改成如下：

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ ? & ? & ? & ? \\ 0 & 0 & 1 & 0 \end{bmatrix} (x,y,n,1)^T = (x,y,n,1) - -(nx, ny, n^2, n) ==>$$
$$nx = nx; ny = ny; n = 1n; \text{完美成立，则第三行，我们可以得到：}$$
$$Ax + By + Cn + D = n^2 ==> A = 0, B = 0, Cn + D = n^2$$

另一方面，由于远平面中心点坐标值(0,0,f)不变，则有如下等式:

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & C & D \\ 0 & 0 & 1 & 0 \end{bmatrix} (0,0,f,1)^T = (0,0,f,1) ==>$$

$$Cf + D = f; \text{根据前面} Cn + D = n^2 ==>$$

$$C = n + f, D = -nf$$

最终，我们得到M_{p2o}等式如下：

$$M_{p20} = \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

4.2.2 长方体正交变换

通过以上推导，我们已经得出了frustum到长方体变换矩阵，接下来需要正交投影变换即可，则我们有以下等式：

$$M_p = M_o M_{p2o} = M_s M_t M_{p2o}$$

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & -\frac{l+r}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

4.2.3 一般应用

以上推导出来公式，在实际使用时，需要进行参数转换下。通常情况下，透视投影矩阵采用fov，aspect，far，near四个参数定义。

1. fov：视场角；
2. aspect：宽高比；
3. far：远平面；
4. near：近平面

则有等式：$n=near;f=far;$同时根据三角形公式，有：$t=near*tan(fov/2),b=-near*tan(fov/2)$，有宽高比aspect得到：$r=aspect*near*tan(fov/2),l=-aspect*near*tan(fov/2)$，最终，我们有以下公式：

$$M_p = \begin{bmatrix} \frac{1}{tan(fov/2)*aspect} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(fov/2)} & 0 & 0 \\ 0 & 0 & \frac{far+near}{far-near} & -\frac{2near*far}{far-near} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

5. 视口变换

经过透视变换之后，所有物体坐标都将规范到$[-1,1]^3$范围内。最终我们需要将物体绘制到屏幕上，因此最后还需要做一个屏幕变换操作。

对于屏幕，我们抽象定义成一个二维数组组成的数组，并且每个元素是一个像素点。二维数组维度我们用分辨率定义。例如分辨率1920*1080，表示二维数组大小。在实际中像素很复杂，但我们这里简单将 像素点定义成一个小方块，每个像素可以呈现黑到百各种颜色，可以用RGB表示。

屏幕空间，我们将屏幕左下角定义为原点(0,0)，右上角定义为(w,h)，因此视口变换可以简单先平移在缩放，具体公式如下：主要这里我们不考虑z坐标情况

$$M_{viewport} = M_s M_t = \begin{bmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

6. 总结

变换内容，我们讲了基本5中变换矩阵，同时还讲到如何进行法线变换、矩阵分解和逆矩阵内容。还有一些其他内容，由于在后续的内容中暂时不会用到，因此不多做过多阐述。

变换其他内容，包括顶点混合、形变等高级部分，将放在后续章节讲述。这是因为光栅化渲染和光线追踪内容，并不会涉及。等到讲到几何部分才会用到。

第三篇 实践篇--基础数学内容和基础变换

终于到实践环节了，作为本教程第一篇实践章节，有几点需要提前约束下：

- 语言：采用C++语言实现；
- IDE：采用vs2017；

采用C++语言，主要是因为运行效率比其他语言快很多。同时，由于教材特性，我们实现过程时，不会使用一些高级语法糖，尽量采用简单平滑描述方式。同时，采用v2017 ide，方便调试。我们最后代码都会放在GitHub上（https://github.com/DionysosLai/SoftwareRenderer），一般都是直接放源码，不会考虑cmake方式。

另一方面，随着实践篇深入，同一个文件同一段代码可能会有所区别。因此GitHub上的代码，已增量形式存放，你只需要对比就可以。

1. 向量实现

这一部分是第二章 基础数学 向量章节实现。

由于向量内容相对较少，同时会普遍运用到各种地方，因此我们直接定义成class，并且全员public属性。首先，我们创建一个vector.h 头文件。并添加如下代码：

```
#ifndef _VECTOR_H
#define _VECTOR_H

#include<string>

template<typename T>

class Vector3 {
public :
    T x;
    T y;
    T z;
    T w;

    // 三种不同构造方式
    Vector3() :x(0), y(0), z(0), w(1) {};
    Vector3(T val) :x(val), y(val), z(val), w(1) {};
    Vector3(T xVal, T yVal, T zVal) :x(xVal), y(yVal), z(zVal), w(1) {};
};

typedef Vector3<float> Vector3f;
typedef Vector3<int> Vector3i;

#endif
```

1.1 四则运算实现

1.2 归一化与齐次坐标化

1.3 测试

2. 矩阵实现

2.1 四则运算实现

2.2 转置与逆变换

2.3 测试

第四章 理论篇--基础绘制

经过前面步骤之后，我们最终可以到一个二维的场景图，下一步我们需要将得到的结果：空间中描述的一堆三角形网格信息，在屏幕上呈现出来，这个就光栅化过程。

对光栅化的内容，可以看第一章《渲染管线》内容。光栅化核心内容：

1. 如何光栅化（绘制）一条直线；
2. 如何光栅化（绘制）一个三角形；

至于后续的光照模型（着色）、网格简化、纹理映射等，我都认为核心内容的推广。因此，本章节主要阐述光栅化的最基础内容，

1. 线段绘制

绘制问题是我们要谈到的第一个光栅化绘制算法，相关内容相信很多人之前都会零零散散接触过。

在数学上，一条直线段可以有无穷多个点，但在屏幕呈现上的像素具有实际大小（可以理解为一个小正方形），因此直线段绘制时，需要做一些处理。如下图所示：

用有限点来逼近无限点效果。

1.1 基本思路

已知2个顶点P0和P1，我们根据数学直线定义：$y=kx+b$，可以求出直线公式：

$$y = \frac{y_1 - y_0}{x_1 - x_0}x + \frac{x_1 y_0 - x_0 y_1}{x_1 - x_0} \tag{1.1}$$

因此对于任意点$x\in[x_0,x_1]$，根据公式都可以求出。当然由于像素点都是整数范围，因此结果我们需要做四舍五入处理。

直线算法是最基本图形，一个动画或者真实图形往往需要调用成千上万次画线算法程序，因此直线算法的好坏与效率至关重要。当前$y=kx+b$算法存在一个最基本问题，就是如何将乘法优化掉。

1.2 DDA(数值微分)画线算法

DDA（Digital Differenttial Analyzer）算法，引入一个重要思想----增量思想。具体推导如下：

$$\begin{aligned} y_i &= k_i + b \\ y_{i+1} &= kx_{i+1} + b \\ &= k(x_i + 1) + b \\ &= kx_i + k + b \\ &= y_i + k \end{aligned} \tag{1.2}$$

于是，我们可以得到：当前步的y值等于前一步y值加上斜率k。这样就将一个乘法和加法变成了一个加法。

具体步骤如下图所示：

DDA算法将一个乘法和加法变成了一个加法，效率得到了提升；但好坏程度呢，通过分析，我们可以看到当|k|<1时，表现效果优良；但当|k|>1时，绘制时会变得非常系数，具体效果图如下所示：

可以看到当k不断增大时，线段会变得非常稀疏。

因此，当|K|>1时，将原先x为自增量变成y为自增量，具体公式如下：

$$\begin{aligned} y_i = kx_i + b; ==> x_i &= \frac{y_i - b}{k} \\ x_{i+1} &= \frac{y_{i+1} - b}{k} \\ &= \frac{y_i - b}{k} + \frac{1}{k} \\ &= x_i + \frac{1}{k} \end{aligned}$$

优化后算法效果如下：

具体实现，可以参考函数drawLineDDA1实现。

从上图可以看到，优化后的dda算法明显比之前好很多了。

1.3 中点画线算法

采用增量思想的DDA算法，直观明了，代码易实现。公式$y_{i+1}=y_i+k$，没计算一个像素坐标，只需计算一个加法即可。当然在实际代码实现中，存在一个浮点转整形的四舍五入过程，如下所示：

```
int x = (int)(xi + 0.5);
```

那么如何改进呢：

1. 改进效率，由于每计算一个像素坐标，计算量只有一个加法运算，基本不存在优化空间。但前面提到，我们需要做一个浮点到整形的四舍五入过程。因此，可以考虑**浮点运算变成整数运算**；
2. 直线公式变换。将原先斜截式方程采用两点式或者一般式等方程表示；

根据改进方案2，我们最终引申出直线画线法。

直线一般方程如下 所示：

$$
\begin{aligned}
&F(x,y)=0 \\
&Ax+By+C=0
\end{aligned}
\tag{1.3}
$$

对于任意点，我们有如下3中情况，如图所示：

1. F(x,y)=0==>点在直线上；
2. F(x,y)>0==>点在直线上方；
3. F(x,y)<0==>点在直线下方；

则对于任意一条直线，当$|k|<=1$时（大于1时，则跟DDA算法优化思路一致），假设当前像素点(x_i,y_i)在直线上（并不一定是数学意义上F(x,y)=0，而是点与线很接近，因此绘制时像素点(x_i,y_i)表示在直线上），则x+1时，y要么+1，要么不变2种情况。

如何确定呢？我们采用直线上点Q与中点$(x_i+1,y_i+0.5)$进行比较。当：

1. Q点在M点上方，则点Pu比Pd更加靠近直线，我们认为下一个像素点$(x*{i+1}, y*{i+1})$；
2. Q点在M点下方，则点Pd比Pu更加靠近直线，我们认为下一个像素点$(x*{i+1}, y*{i})$；

当然还有第三种情况，Q点在M点，则我们任意选一个点即可。在图形学中，我们会遇到很多边界情况，例如判断一个点是否在多边形内，点正好在多边形边上的情况时，我们可以认为点在多边形内或者点在多边形外部。这些对最终呈现不会有任何影响。

以上思路，就是中点画线法的基本思路。

接下来，需要对中点画线法进行效率分析，任意点(x_i,y_i)，则下一个中点$d(x*{i+i},y*{i+0.5})$带入直线方程：

$$
d=A(x_i+1)+B(y_i+0.5)+C
$$

该公式存在2个乘法4个加法，明显比之前差很多。那么我们是否也可以采用增量思想进行优化呢：

当d0<=0时，对于M0和M1个中点，存在以下关系：

$$
\begin{aligned}
d_0 &= A(x_i+1)+B(y_i+0.5)+C \\
d_1 &= A(x_i+2)+B(y_i+1.5)+C \\
&= A(x_i+1)+B(y_i+0.5)+C+A+B \\
&= d_0+A+B
\end{aligned}
$$

同样当d0>0时，推导如下：

$$
\begin{aligned}
d_0 &= A(x_i+1)+B(y_i+0.5)+C \\
d_1 &= A(x_i+2)+B(y_i+0.5)+C \\
&= A(x_i+1)+B(y_i+0.5)+C+A \\
&= d_0+A
\end{aligned}
$$

同时，对于直线起始点(x_0, y_0)，有：

$$
\begin{aligned}
d_0 &= A(x_0+1)+B(y_0+0.5)+C \\
&= Ax_0+By_0+C+A+0.5B
\end{aligned}
$$

最终，我们有如下公式：

$$
\begin{aligned}
&d_{new}=\begin{cases} d_{old}+A+B & d_{old}<0 \\ d_{old}+A & d_{old}>=0 \end{cases} \\
&\text{其中} d_0=A+0.5B;
\end{aligned}
\tag{1.4}
$$

我们可以看到，中点画线算法也只有一个加法运算，这块就和DDA算法一致；同时不在需要浮点到整形的四舍五入计算，因此整体效率高于DDA算法；

前面，我们讨论了$|k|<=1$情况，当$|k|>1$时，只只需要将x增量方式变成y增量方式即可。

1.4 Bresenham 算法

DDA算法将效率提高到每步只需要一个加法；中点算法进一步将效率调高到不需要四舍五入精度转换问题。那么有没有一个算法，效率更好，切应用更加广泛呢？

这里，我们就需要提到Bresenham算法。

2. 三角形绘制

作为光栅化的第二个基础内容：三角形绘制，比直线绘制稍微复杂一些。注意，这里输的三角形边绘制，并不简简单单指的是三条边绘制，而是包括了填充问题。其中对于三条边绘制，可以简单采用直线绘制算法。因此，我们重点需要关注三角形填充问题。

2.1 X扫描线算法

光栅图形的另外一个基本问题，是如何将用顶点表示的多边形转换为点阵表示。这个转换我们称之为**多边形扫描转换。**

现在问题是我们已知多边形顶点（在这里，我们简化为已知三角形三个顶点），如何找到多边形内部的点，然后将点进行着色，即实现多边形内部填充。

X扫描线算法基本思想是按照扫面线顺序，计算扫描线与多边形的相交区间，再用要求的颜色对这些区间内像素进行着色，即完成填充工作。

X-扫描线算法步骤如下：

1. 确定多边形所占有的最大扫描线数，得到多边形顶点的y轴最小和最大值（ymin和ymax），x轴的最小和最大值（xmin和xmax）；
2. 从y=ymin到y=ymax，每次用一条扫描线进行填充；
3. 对一条扫描线填充过程可分为：
    - 求交：计算扫描线与多边形各边的交点；
    - 排序：把所有交点按递增顺序进行排序；
    - 交点配对：第一个与第二个交点配对，第三个与第三个交点配对。。。等按照这个方式进行；
    - 区间填色：把这些相交区间内的像素点按照要求颜色进行着色；

以上算法中，对于交点配对问题中，我们默认了交点个数是**偶数。**但是，当扫描线与顶点相交时，交点的个数极有可能变成**奇数。**

因此，我们采用如下方案解决：

1. 若顶点的两条边分别落在扫描线的两边，交点个数算1个；
2. 若顶点的两条边落在扫描线的一边，交点个数算0个（有些地方这里是算2个，不影响）

这里，交点个数如下图所示：

到目前为止，算法的步骤和交点个数问题，均已解决。按照算法步骤，我们只需要将多边形的所有边存在一个数组中，扫描线依次与边求交，即可得到所有顶点。

但这样处理，效率比较低下。如下所示：

$$
\text{假设两条直线} y=k_1x+b_i、y=k_2x+b_2,\text{交点公式为：}
\begin{cases}
x=\frac{b_2-b_1}{k_1-k_2} \\
y=k_1x+b_1
\end{cases}
$$

在已知直线方程式情况下，求一个交点需要2个乘（除）法和3个加（减）法。因此，我们需要将算法改进。

2.2 X扫描线算法改进

如何改进X扫面线算法呢，最理想的方案还是不进行求交运算。同样方式，我们 采用增量思想。

当前情况，我们已知任意一条扫描线的y值，关键是求得x的值。

一种最最简单方式（纯粹是笔者自己构思方法）：

1. 求得所有边直线方程式，和多边形包围盒；
2. 从最低一条扫描线开始，从做左到右，依次计算每个像素点与所有边情况。这样我们得到所有交点情况。

这里的步骤2实现起来细节比较多。包括如下：

1. 点与线段相交情况前提是，点在线段围城包围盒中；
2. 点与某条线段相交，就不可能与其他线段有相交情况。（当点就是顶点是，例外讨论）。

本思路实现可以参考。。。。

那么，我们如何不进行求交呢？或者减少求交运算呢？这里，我们采用如下改进方案；

1. 在处理一条扫描线时，仅对与它相交的多边形的边（有效边）进行求交计算；即点与线段先交前提是，点在线段围城包围盒中；---减少求交计算；
2. 考虑扫描线连贯性，即当前扫描线与各边的交点顺序与下一条扫描线与各边交点顺序很有可能相同或者相似；除非交点遇到顶点情况，才会发生改变；同时，某条边与扫描线相交时，它很可能

2.3 ToLeft 算法

待续

3. 三角形网格

待续

3.1 网格定义

待续

3.2 三角形网格实现

待续

4. 总结

待续