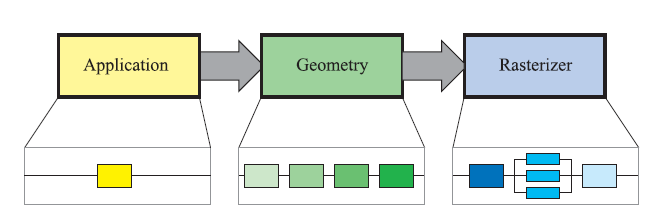
# 2渲染管线

渲染管线从大的结构上分可分为3个部分：应用程序阶段->几何阶段->光栅化阶段



## 2.1应用程序阶段

应用程序阶段：主要决定发送哪些图元给几何阶段。

在这个阶段完成的功能主要有：物体剔除（场景图结构），碰撞检测，纹理动画，动画的变换，层级视景体剔除。

## 2.2几何阶段

几何阶段主要被分为以下5个功能阶段：模型和视图变换->顶点着色->投影变换->裁剪->屏幕变换。



注意：几个功能阶段可能会被合并为一个流水线阶段。如OpenGL，将模型视图变换，顶点着色和投影合并为一个流水线阶段：顶点着色器。

### 2.2.1模型变换

### 2.2.2 顶点着色

（物理渲染是渲染理论下的一个分支。主要建模材质与光源之间的关系）

决定光源对材质影响的过程叫做：**着色**。着色阶段要计算着色方程

在顶点计算着色方程叫做：**顶点着色。**在光栅化以后的像素上着色叫：**像素着色**或者**片元着色**

**注意：**一般情况下认为顶点着色在世界空间下完成。所以法线变换一般变换到世界空间下。（也可以变换到其他方便的空间下）

### 2.2.3投影变换

（规范投影体又叫**规范化设备空间：NDC**）

### 2.2.4 裁剪

如果用顶点裁剪，裁剪过后会使顶点数=索引数。因为裁剪是按三角形来裁的。它无法确定被先后放入裁剪程序的两个三角形，在裁剪过后生成的新顶点是否有共点。

如果要判断共点的话，需要遍历整个生成后的顶点集合。这将是一个n^2复杂度的算法，不合适。就让它多占一些内存没有关系。

如果用索引裁剪也同样会出现在无法判断两个三角形中新增加的点是否共点的问题。

但是这样没关系，因为后续的执行都是按三角形来的，即按照索引来执行，顶点数的增加并会增加后续算法的时间，只是多占一些内存而已。

### 2.2.5 屏幕映射

在屏幕映射中从float的坐标映射到Int型的坐标只是简单截断即可。因为认为像素0的中心是0.5。采取最近邻采样原则。即[0,1] （不包括1）之间的数到0.5最近。因此映射到像素0上去。

在屏幕变换的过程中，如果是OpenGL，它会把深度值从（-1,1）映射到（0,1）。

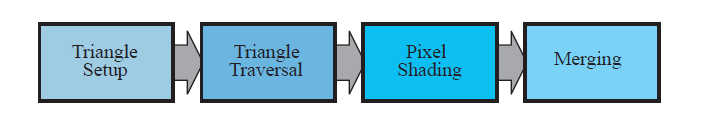
## 2.3光栅化阶段

光栅化的主要功能是：决定被物体覆盖的像素的颜色

具象化就是，如何在屏幕上绘制三角形，以及这些三角形在屏幕上覆盖的像素的颜色。

光栅化被分为4个功能阶段：三角形设置->三角形遍历->像素着色->合并。

光栅化又叫扫描转换。



**光栅化算法原理**

光栅化：是一个过程。判断一个三角形覆盖了二维平面上哪些栅格点。

光栅化的算法有多种，从渲染模式上来讲，有线框模式和填充模式。

线框模式：主要就是在画线，画线算法有Bresenham算法

填充模式：针对凸多边形有比较通用的扫描线算法，对于三角形则有特殊的重心法。

扫描线算法：对于一个凸多边形，可以用多条扫描线去覆盖它。而每条扫描线与凸多边形最多有两个交点。如果有两个交点，这两个交点之间的像素被绘制，如果只有一个交点，那这个交点所在的像素被绘制，如果没有交点，则不绘制。基于扫描线算法的绘制方法有比较经典的上下平底三角形，基于边缓冲区的三角形绘制。所谓边缓冲区就是用一个缓冲区记录每条扫描线与三角形的交点，这个方法不需要将三角形分成平底和平顶。

**重心法**：重心法仅能绘制三角形。不在一条线上的三个点可以确定一个平面，且该平面上的点可以表示为v=a\*v0+b\*v1+c\*v2.如果a>=0,b>=0,c>=则该点在三角形内部。可以根据这个式子在判断一个栅格点的采样点是否被三角形覆盖。

如果一个栅格点的采样点恰好在三角形的一条边上，则使用**左上规则**来决定这个采样点是否被三角形覆盖。如过一个点恰好在三角形的左边或者上边，则认为这个点被三角形覆盖。

**左上规则：上边，**一条完全水平的边，且另外两条边均在他的下面。**左边，**沿着边的方向是向下走的（三角形式逆时针方向），即代表这条边的向量y坐标是<0的。

**边方程：**一种用来判断一个像素是否被三角形覆盖的方法。让三角形三条边为顺时针，或者逆时针，将一个点的坐标代入三角形三条边的边方程中，根据所得值的符号，判断这个点是否在三角形中。

### 2.3.1 三角形设置

**暂时不清楚这个阶段在做什么。**

### 2.3.2三角形遍历

判断哪个像素被三角形所覆盖。

**片段（fragment）**指被三角形覆盖的像素

进行插值计算

**三角形重心坐标**

对于平面上不共线的三点a,b,c，则平面上任意一点可以表示为



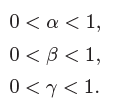








若一个点在三角形abc内则有



### 2.3.3 像素着色

逐像素着色

### 2.3.4 合并

合并阶段需要实现的功能有：alpha混合， Z-Buffer， stencil buffer，frame buffer

**Alpha Test**：一种粗略的实现透明效果的方式。若当前的alpha值不满足某一条件（大于或者小于某一值），则当前片元被丢弃。

**Alpha 混合**：一种实现透明效果的方式。Color=Src\*Src\_Alpha+Dst\*Dst\_Alpha;

**深度缓存**：若深度小于当前深度缓冲区中的值，则当前片元可见。否则丢弃。

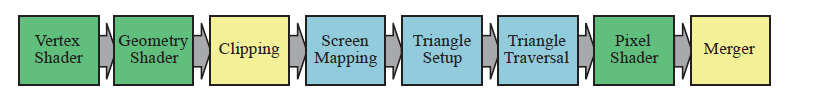
后续应该多注意一下alpha值应该如何处理

# 3GPU

在渲染管线中，应用程序阶段由CPU执行，几何阶段和光栅化阶段由GPU执行

## 3.1GPU管线

GPU流水线主要分为：顶点着色->几何着色->裁剪->屏幕映射->三角形设置->三角形遍历->像素着色->合并



**注意**：GPU管线与渲染管线的区别在于：**渲染管线是按照功能来划分的，GPU管线是按照实现来划分的**。功能阶段和实现阶段是一个多对对的关系。

**顶点着色器**主要完成的功能有：模型视图变换，顶点着色，投影变换

**几何着色器**：在GPU运行中动态的创造新的图元或者摧毁一些图元

## 3.2 可编程着色器阶段

这一节主要介绍Shader编程模型

GPU一般为4-SIMD（4路单指令流多数据流）类型

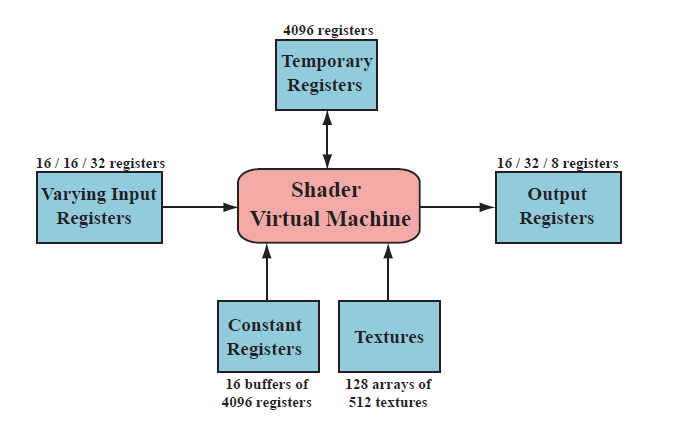
**一个DrawCall指绘制一组图元**

每个着色器阶段由两种输入：**uniform**和 **varying**

Uniform：在一个DrawCall中保持为常量，为只读状态

Varying：变量

Shader虚拟机模型如下



Shader支持的控制流分为两种：**静态控制流**和**动态控制流**

静态控制流：需要来做判断的变量为Uniform类型

动态控制流： Varying类型

## 3.3可编程着色的历史

## 3.4顶点着色器

顶点着色器可以完成的功能：棱镜特效，物体形变

## 3.6像素着色器

OpenGL把像素着色器叫做片段着色器的原因是对于像素着色器的输出需要经过深度测试，alpha测试，混合等操作再输出到像素上。所以叫片段着色器以示区别。

最多有16个向量可以被从顶点着色器传递像素着色器。（SM4.0）

像素着色器可以访问导数信息。

在像素着色器中实现multiple render targets（MRT）多重渲染目标技术。（仔细看下什么叫多重渲染技术）

## 3.8特效

一个Pass指一次渲染，包含一个顶点着色器和一个像素着色器

（本章节完结）

# 4 变换

仿射变换：线性变换+平移变换，包括：平移，旋转，缩放，反射，错切

仿射变换的特点：**放射变换前后平行线依旧平行**

## 4.1 基本的变换

### 4.1.1 平移变换

### 4.1.4错切变换

错切变换：错切变换把一个物体看做是有许多内部夹层组成，然后每一个夹层可以单独平移。

例如在x方向上错切

以后应该稍微多注意一下错切变换

### 4.1.7法线变换

对于一个模型做变换T，则此模型的切线也做同样的变换。

对于一个模型做变换T，则此模型的法线变换为

因为：

所以：

构造新的N’’

可以看出N’’一定与N’同向。因此N’’也是模型的法线.

**注意**：使用伴随矩阵有如下有趣的性质。不管向量N，V之前的关系如何，若变换T不可逆，则总与TV垂直。

因此使用伴随矩阵来求V变换后的法线总是可行的。

因为平移不改变法线，并且总是在投影之前去变换法线。因此可以只用变换矩阵的左上3\*3的矩阵来计算法线。也即，计算法线的时候只用一个3维向量。

下面来证明为什么在投影之前做法线变换可以只用左上角的3\*3矩阵来做。假设对顶点的变换为T，则对向量的变换为

由于在投影之前,的齐次项均为1，因此向量的齐次项为0。所以T的第4列对计算结果没有影响。又因为我们现在想重新用三维坐标来表示向量。因此T的第4列可以完全丢弃。因此有

又因为，投影变换之前的变换不会改变齐次项。因此T的第4行为。因此的第4行也全部为0.因此可以丢弃。

所以在投影之前，若对一个模型做变换T,则对这个空间里的向量做的变换是用T的左上3\*3矩阵来做的变换。

再对比之前求法线的算法，不难得出结论，若要用左上3\*3的矩阵来对法线转换，必须在投影之前转换。

### 4.1.8 视图变换矩阵的推导

在三维坐标系下，世界空间的点与相机空间的点的坐标有如下关系

其中d,u,r是相机空间下的坐标.现在需要解出d,u,r。两边同时点乘 D,U,R.得

现在讲相机空间和世界空间的坐标用齐次坐标表示，分离一个第4项=1出来。

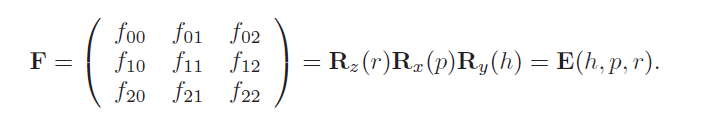
因此视图变换矩阵为

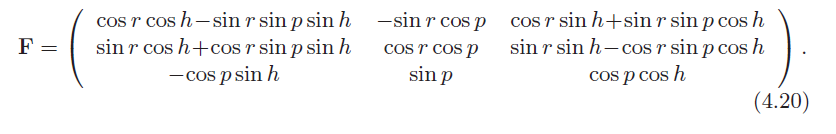
## 4.2几种具体矩阵变换

### 4.2.1 欧拉变换

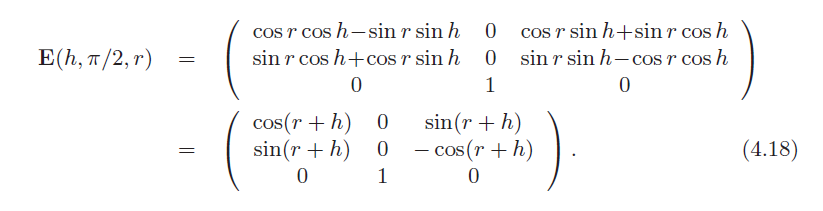
一个万向锁的例子：先绕x轴旋转r,,在绕y轴旋转90度，此时旋转后新的旋转轴z’与最初的旋转轴x同向。因此丢失了一个轴的旋转自由度。

证明：





将p=90度带入上式。可得



新旋转矩阵实际只用一个自由度 r+h.

注意：**万向锁是不可避免的**。

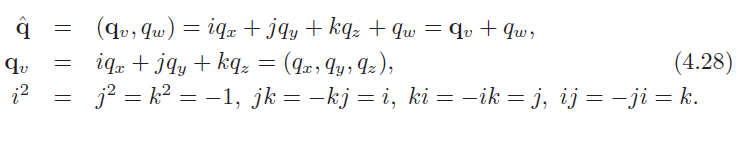
### 4.2.4 绕任意轴旋转

方法很简单，以这个轴为一个基，构造一组正交基，将物体变换到这组正交基坐标下，然后就可以绕坐标轴旋转了。

## 4.3 四元数

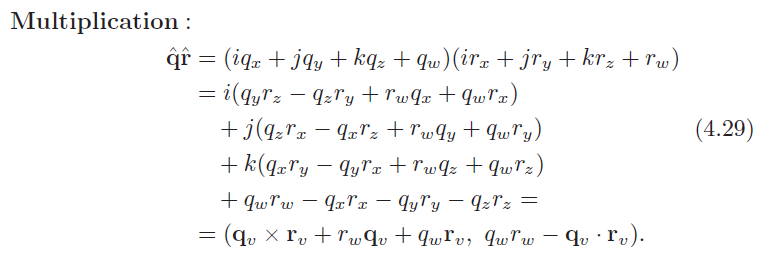
### 4.3.1 基本的四元数运算

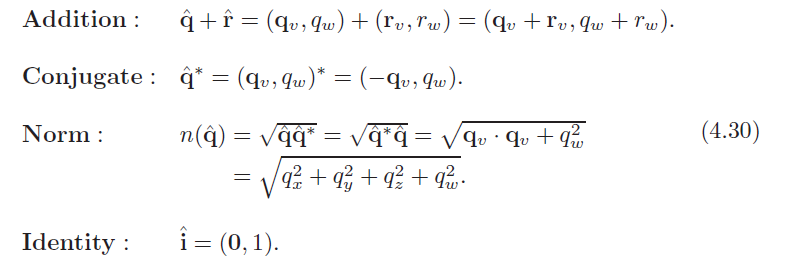
**定义：**一个四元数可以定义成如下形式



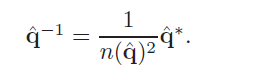
其中，qw称为实部，qv称为虚部，ijk称为虚元。

其中qv可以简单的看作一个向量，其拥有向量的所有运算。

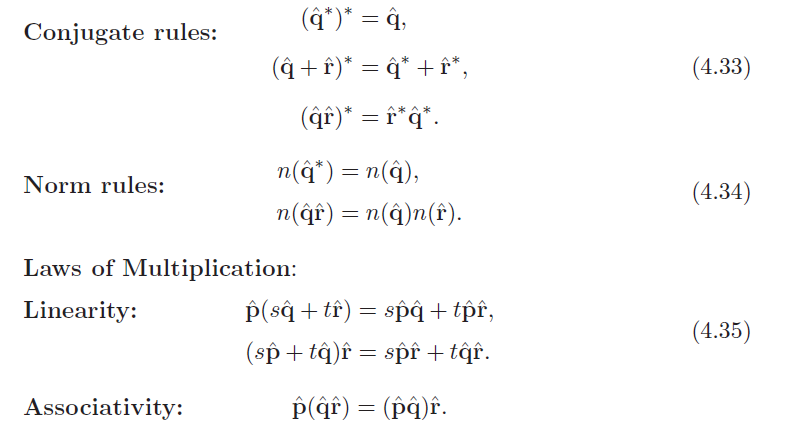






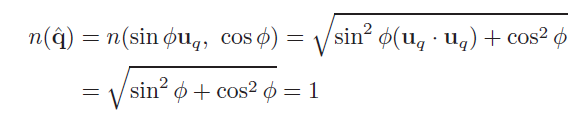






一个单位四元数可以写成如下形式





其中，



### 4.3.2 四元数变换

将一个点p直接看作一个四元数**，**有一个单位四元数*,*如下运算表示一个旋转，

其中，**u**表示旋转轴，旋转角度为2，方向为逆时针。

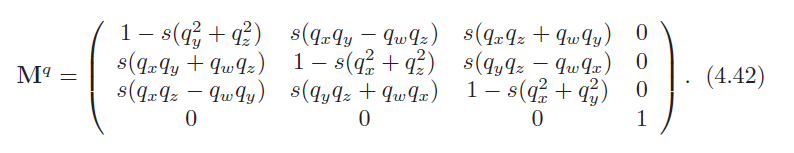
因为

其中和-表示相同的旋转。这也就意味着从一个旋转矩阵转换为四元数，可以得到两个四元数，和-。

级联旋转如下

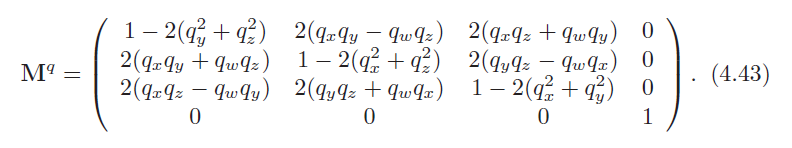


**一个四元数变换对应的矩阵变换如下**



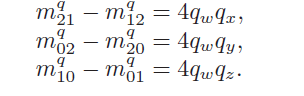
其中s=2/n(q)

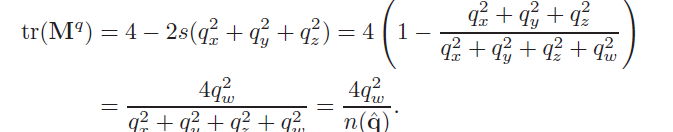
因为q是一个单位四元数，因此有



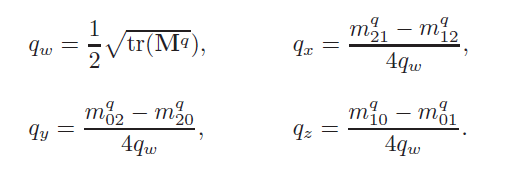
**从一个矩阵求对应的四元数。**

因为





根据这个迹可以求出qw。然后再代入上面3个方程，可以得出qx,qy,qz。解出的解如下。

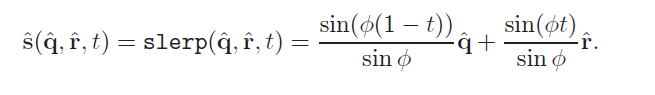


**球面线性插值**

给定两个四元数，q，r，和一个参数t，求一个插值四元数s

从上面这个式子可以看出，当t=0时，s=q,t=1时，s=r.

插值还可以使用如下的方式进行。



其中。

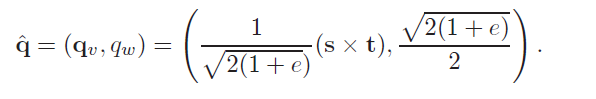
**从一个向量到另一个向量的旋转**

已知两个向量，**s**和**t，**如何求从s到t的最短路径的旋转？

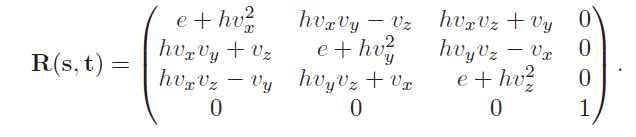
1. 首先必须归一化s,t
2. 
3. 

在上式中需要根据 和cos，去计算sin和cos。为了计算方便，可以做如下化简。将u代入q中

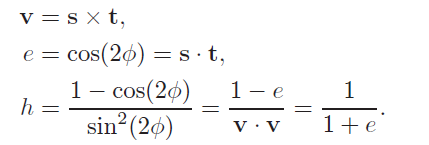




其对应的旋转矩阵可以表示为



其中



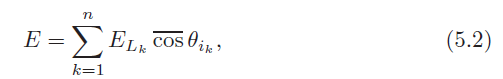
## 4.4顶点混合

# 5视觉外观

## 5.1视觉现象

## 5.2 光源

光向量的方向与实际光线的方向相反。这是为了后续处理的方便，因为我们需要的总是从物体到光源的光的方向，而不是从光源到物体的方向。



**Irradiance**：辐射照度，用RGB表示。

方向光建模：光源方向，辐射照度。

对于方向光而言，由于所有的方向都相同，所以辐射照度=辐射率

## 5.3 材质

在着色方程中有镜面光(specular)和扩散光(diffuse)

**Specular**:指那些入射光直接从表面反射，没有能量被吸收。

**Diffuse**：指入射光射入物理内部，经过吸收后有重新从表面发射出去。

**Exitance**：出射照度

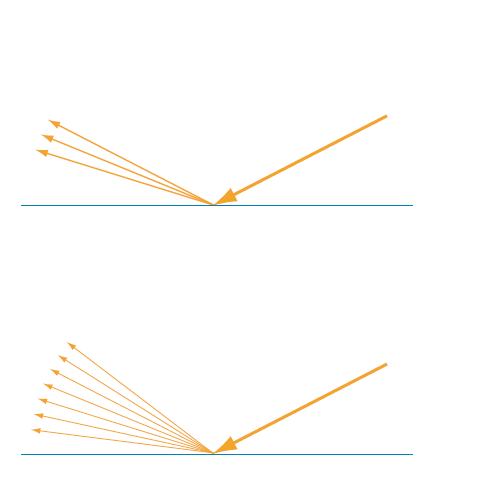
**Radiance**: 辐射率。只单一方向上的辐射照度。

对于同一个物体来讲，出射照度除以入射照度是一个常数。

同一个物体，对于不同颜色辐射系数是不一样的，因此，c是一个向量

这个c通常被叫作表面颜色c。

**注意**：对于同一个物体来讲，镜面光和扩散光的辐射系数是不一样的，因此，对于同一个物体来讲，讲会有两个辐射系数，



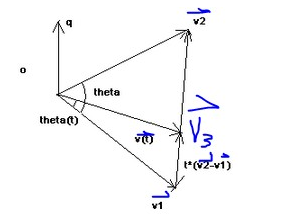
对于越光滑的物体，镜面反射光的方向越集中

## 5.4 传感器

## 5.5 着色

为了方便理解，所有的着色过程均在世界坐标系下计算。

两个向量的线性插值结果如下图



插值公式为

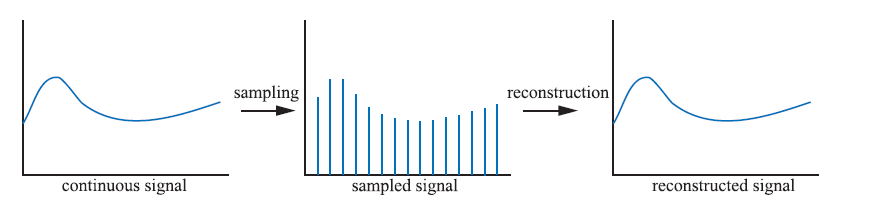
从这个公式看，对法线做线性插值，可以得到正确的角度。

## 5.6 走样与反走样

### 5.6.1 采样和滤波理论

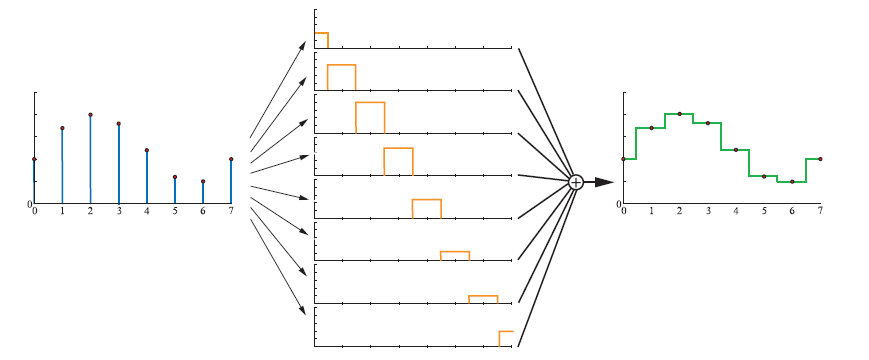
**采样：**对原始连续信号等距取一些点上的函数值出来。采样又叫**离散化**

**滤波：**对采样出来的点进行操作，恢复原始的信号。

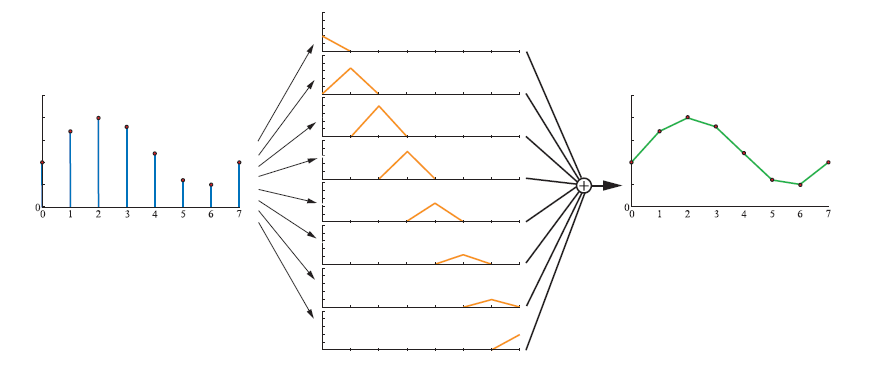


**重建**

箱式滤波器（最近邻）过程如下图

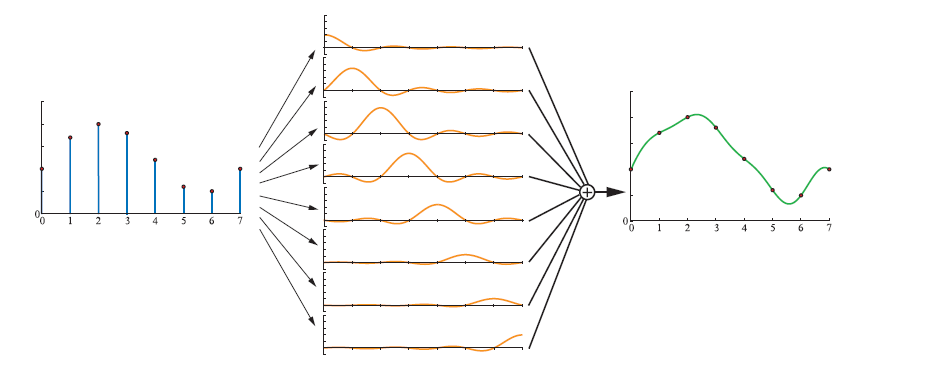


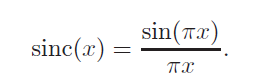
三角滤波器（线性插值）过程如下图。



低通滤波器（sinc滤波）重建过程如下。

所谓低通，指的是低频信号通过，高频信号丢弃。Sinc是其中一种低通滤波器





更好的滤波器是



**重采样**

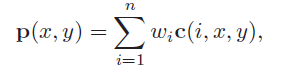
见附录

**奈奎斯特-香农 采样理论**

见附录

### 5.6.2 屏幕空间的反走样

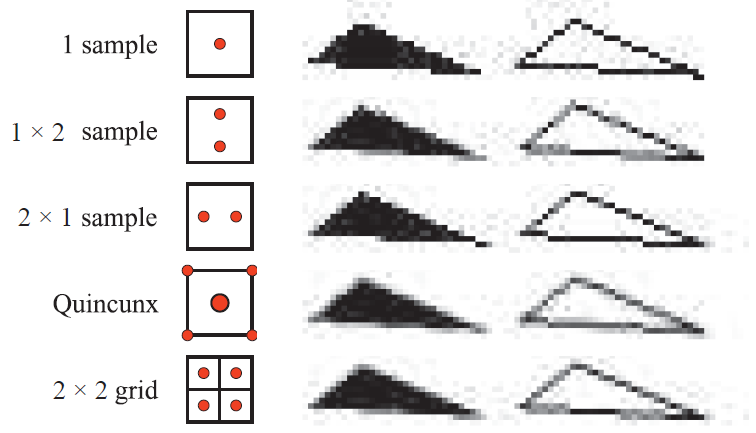
屏幕空间的反走样的思路就是对于一个屏幕格点使用多个采样点像素值的加权和来作为其像素值。

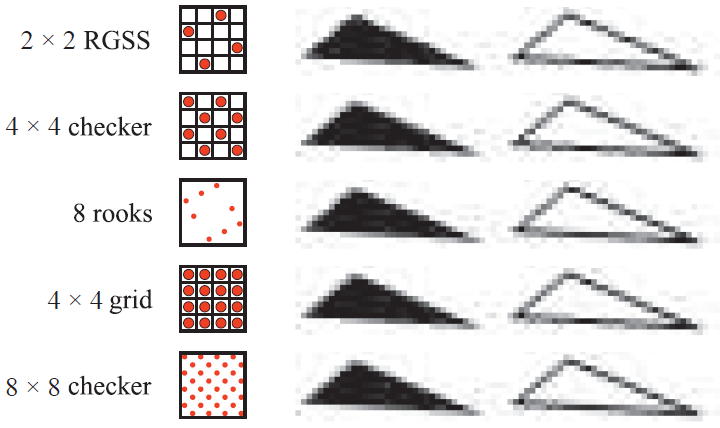


其中p(x,y)是格点(x,y)处的像素值，c(i,x,y)是对这个格点第i个采样的颜色。

c(i,x,y)的计算分成两步。第一步计算先根据(i,x,y)计算一个精确的采样位置，第二步根据这个位置检索一个颜色值出来。另外权重和。大部分实时渲染系统中，

这种方法叫做**超采样**。其一个具体的例子是FSAA（Full Screen AntiAlias）。它的的思路是如果我们要渲染一张M\*N 大小的图像，我们先渲染一张2M\*2N大小的图像，然后再4变1.缩回到M\*N的大小，生成新的图像。FSAA的另一种实现是Accumulation Buffer（累加缓冲区）使用累积缓冲区可执行不同的采样样式，如下图，







另一种更快速的超采样方法是MSAA（MultiSample AntiAlias）。在MSAA中对一个屏幕格点有多个采样点（几何采样点），比如说在4\*RGSS，对于一个屏幕格点有这4个采样点。然后对于每一个屏幕格点，其中心就是它的像素采样点，因此对于每个屏幕格点像素着色器执行一次。然后那4个几何采样点的颜色可以这么确定：如果这个点被三角形覆盖，它的颜色就是像素采样点的颜色，如果几何采样点没有被三角形覆盖，它的颜色就是0.最后这个屏幕格点的颜色就是这4个几何采样点颜色的平均值。因为MSAA对每个屏幕格点只有一个像素采样点，因此MSAA只能处理边的反走样。因为三角形内部的点一定是完全覆盖所有的几何采样点。另外像素采样点不一定非要是屏幕格点的中心，也可以根据三角形对几何采样点的覆盖情况来选择像素采样点。

## 5.7 透明 alpha 与合成

根据渲染算法的不同，透明物体的绘制可以分为**View-Based**和**Light-Based**

Alpha 混合是View-Based方法。

在透明物体的绘制过程中，会遇到Z-Buffer 的限制。因为Z-Buffer中每一个像素只存储一个深度值，但是如果一个像素上有多个透明物体，Z-Buffer 的存储会出现问题。使用A-Buffer可以解决这个问题。但是目前市场上的显卡均是基于Z-Buffer。所以透明物体的渲染应该以Z-Buffer为基础。OpenGL实现了软件方式模拟A-Buffer。

一种实现透明效果的方法被称作screen-door transparency。这种方式的做法是对于一个透明物体使用棋盘格样式渲染，即，每隔一个像素渲染，（渲染，不渲染，渲染，不渲染…）这样那些不被透明物体覆盖的屏幕格点就可以用来渲染透明物体后面的物体。这种方法要求渲染的分辨率足够高，不然，这种棋盘格的渲染样式很容易被观察出来。这种方法额另外一个缺点是在一个屏幕区域只能渲染一个透明物体。因为如果有两个透明物体交叠在一起，那么渲染时就不知道渲染那个物体的颜色。

另外一种更通用的渲染透明物体的方法是alpha混合。



透明物体的渲染顺序为：

1. 渲染不透明物体。
2. 将透明物体按照离投影平面距离的远近排序。由远到近渲染。（如果希望可以不按顺序渲染透明物体，则需要另外一个眼神缓存，将透明物体渲染至该缓存上，然后再将两个颜色缓存混合）。

Alpha 测试：如果alpha不满足一定条件，则丢弃片元。（如alpha<某一值，或alpha>某一值）

另外对物体按照其重心排序并不能保证正确的渲染。对透明物体正确的渲染要求对片段正确的排序。

如果不方便实现排序，在渲染透明物体时可以使用Z-Test 代替Z-Buffer。

另外还有一种混合方式，称作additive blending（增量混合）



增量混合不需要对透明物体排序，因为透明物体无论是否排序对颜色的贡献都是ascs，即贡献只与自己的颜色和alpha值有关。这种混合方式有一个弱点，当有多个透明物体叠加时，会导致颜色过亮而不太真实。

Alpha值一般从纹理中读出来。

其他部分没怎么看明白

## 5.8 伽马矫正

**光栅设备**：一种显示设备，有一组矩形网格格点，每个格点可以被单独设置颜色。

**光栅图像**：由光栅设备生成的图像。是一个二维颜色数组。

**矢量图像**：使用线，面来描述形状（Mesh）。

显示设备分类：**发射显示器（LED）**和**透射显示器（LCD 液晶显示器）**

**发射显示器：**根据像素值发射不同数量的光子，产生亮度。

**透射显示器：**根据像素值透过不过数量的光子，形成不同的亮度。

**图像的本质：**二维光能量分布。从这个定义可以看出图像是一个连续函数。

一般认为一个像素的采样点为这个格点的中心。

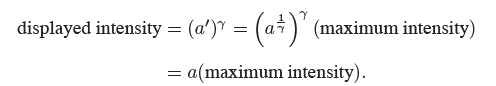
对于一个显示器，若输入的颜色是C，输出的颜色是Co，则有

指数就是伽马。这种指数形式导致了颜色空间的非线性化。例如我们有两个颜色，。我们希望显示两个颜色的中间颜色,但是，我们求均值后得到的颜色却是。二者并不相等。

为了使颜色空间线性化，我们对像素着色器中计算出来的颜色先求一个伽马分之1的幂。



这样处理后，显示器显示的颜色就是



这就是伽马矫正。

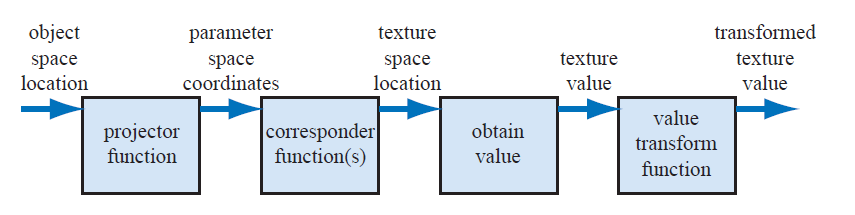
还是上面的例子，如果现在我们有两个颜色，，做完伽马矫正以后，显示器显示的颜色依旧是，然后我们求两个颜色的均值，在做伽马矫正，之后显示器显示的颜色仍然是。这样我们就得到了两个颜色中间的颜色。也即将颜色空间线性化。

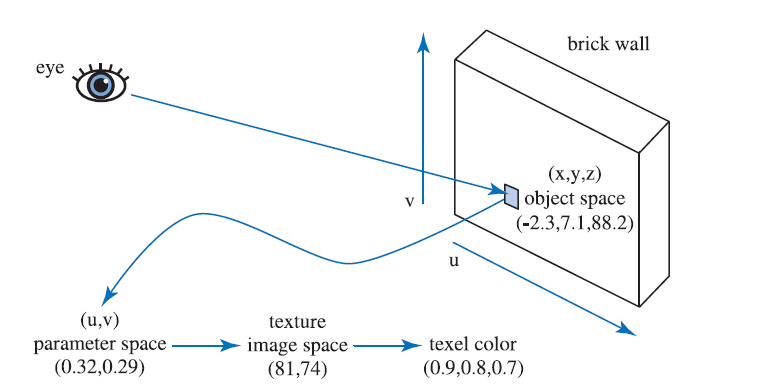
当我们有对图像做滤波操作时，就需要做伽马矫正。

# 6纹理

纹理映射是一个很大的范围。在实时渲染中**使用UV坐标去获取纹素值的过程叫做纹理采样**

## 6.1 纹理流水线





**纹理流水线的步骤**：

1. 拿到模型空间的坐标。
2. 经过投影器函数生成UV坐标。（又叫参数化空间坐标）
3. UV坐标经过一个对应性函数找到它在纹理图像上的纹理坐标。实际上就是拿到图像数组的索引值。并返回纹理值。
4. 拿到纹理值以后再做一个值变换（比如全部乘以一个系数），返回变换后的值。

### 6.1.1 投影器函数

投影器函数：**将一个三维空间的点的坐标转换成uv坐标**

### 6.1.2 对应性函数

对应性函数：**将一个uv坐标转换成纹理空间的索引值**

uv坐标的范围：，包括左端点0，不包括右端点1.

坐标范围为什么要使用这种左闭右开的形式？

假设两个端点都是闭合。那么在考虑[0,1]时，1表示最后一个纹素，在考虑[1,2]时，1又成了第一个纹素，逻辑出现问题。因此为来了逻辑的完整性，必须是左闭右开。

在OpenGL中对应性函数又被称为**环绕模式**（wrapping mode）

### 6.1.3 纹理值

一个纹理中的纹素通常是一个unsigned char 类型，在读取以后需要转换到0.0—1.0之间的float值。

## 6.2 图像纹理

在GPU中一张纹理图像的大小被设置为

纹理放大：**将一张纹理图像显示到更大的屏幕区域上**

纹理缩小：**将一张纹理图像显示到更小的屏幕区域上**

本来为了防止走样，应该对最终像素颜色（即屏幕上的颜色）做滤波，然而只要着色方程的输入和输入是线性相关的，就可以通过对纹素的滤波来反走样。

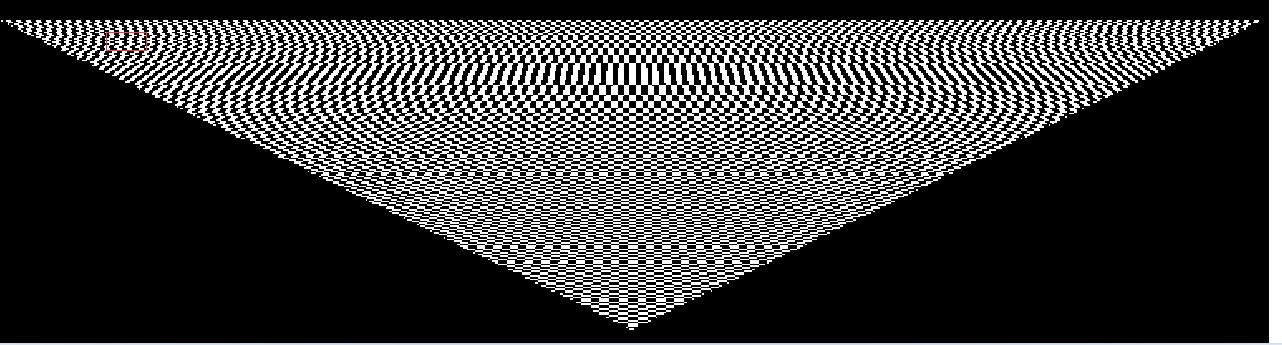
Lo=Ks\*Cdiff.类似这种形式，就叫做着色方程的输入和输出线性相关。其中Lo是输出，Cdiff是输入。

### 6.2.1 纹理放大

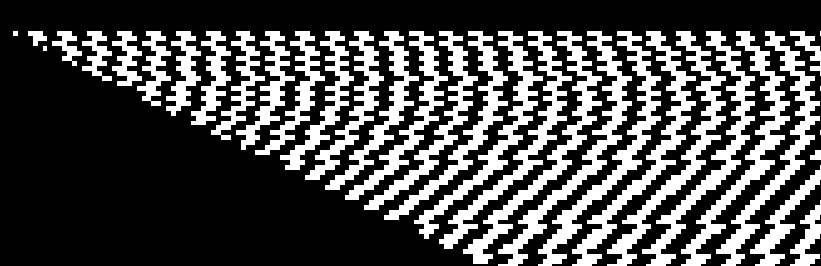
### 6.2.2 纹理缩小

所有的纹理缩小都会出现走样问题。走样问题的解决思路主要有两种：**增加采样频率**（增加**双线性插值的采样点**）和**减少纹理频率**（**MipMap**）

当纹理中携带高频信息时（如棋盘格纹理），纹理缩小走样会比较明显。在图中会出现非棋盘格的样式。如图，这个形式的走样又称为空间混叠。



放大走样的细节如下，这种混叠叫做莫尔条文。



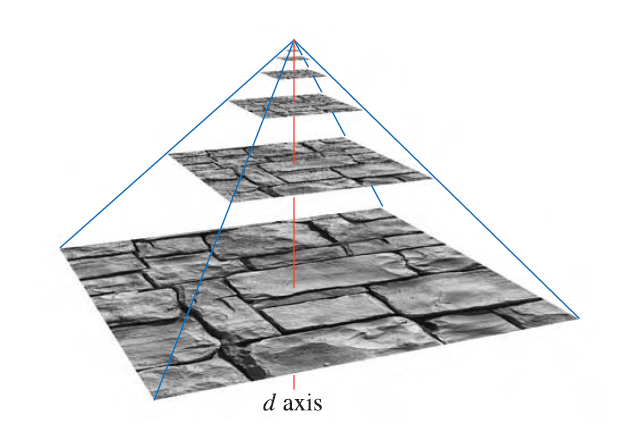
另外，图像中的细节一般携带高频信息。当物体相对相机远近移动时，细节的位置会出现闪烁的现象（时间混叠）。

**注意：**一般对一幅图像做上采样，不会增加图像的频率，但是对图像做下采样会减少图像的频率。

#### MipMaping

**MipMap的生成。**

1. 首先将纹理图像转换为一系列更小的图像。其中，原始纹理图像称作Mipmap Level 0（0级图像）。
2. Level 0降采样至原来的1/4 ，长宽各为原始的1/2生成level 1级纹理。也可以下采样至其它比率的大小如1/9等等。这个取决于我们的需要将纹理的频率下降多少。下采样也未必一定要使用三角滤波（线性插值），对于像棋盘格这种类似分段函数的图像应该使用盒滤波器，（最近邻）。
3. 递归执行下去，直到其中一维或者两维都降为一个纹素。（就是一直4变1，变下去。变到纹素数少于4，变不下去了为止）。所有生成的这些纹理叫做MipMap纹理链。



**注意：为了生成高质量的MipMap纹理链，必须使用高质量的滤波方式和做伽马矫正。**

通常情况下使用的滤波方式：对2\*2的像素做平均生成新的像素值。

上面说的这种使用平均值的方式叫做三角滤波器。更好的滤波器有：Gaussian, Lanczos, Kaiser, or similar filter。

做伽马矫正的目的是将颜色从非线性空间转换的到线性空间，然后做滤波，之后再转换回非线性空间。这样会得到比较高质量的MipMap纹理链。

**MipMap的使用**

使用LOD（Level of Detail）去决定如何使用MipMap纹理

使用（u,v,d）3个坐标去对MipMap采样。这里的d就是LOD。其中这个d不一定非要是整数。这样的话，纹理链中与d相邻的两个层级的纹理都被采样。之后使用线性插值综合两个纹理采样出来的值。这个过程又被称做**三线性插值。**MipMap的查询也未必非要三线性插值，像棋盘格纹理这种图像，应该使用最近邻。

d的计算方式：使用（u，v）对（x,y）的4个偏导中的最大值。

d 的计算方式主要有两种思路：

1. **利用偏导数来求d，**这种方式导致每个像素的d都不同。这种方式适合连续函数图像。
2. **利用图元在屏幕上的宽高和纹理的宽高比来计算d，**这种方式导致整个图元使用的同一个d。这个方式适合棋盘格这种分段函数图像。如果棋盘格图像在一个图元上使用不同层级的纹理，会导致纹理有割裂感。

MipMap 的一个问题是会造成过模糊。因为当u方向覆盖很多的纹素而在v方向上覆盖很少的纹素的时候。MipMap会倾向于使用更大的d来计算纹理值。（当我们平视一个MipMap纹理表面时）

对于MipMap上述问题的一个拓展的解决方案是**RipMap,即各项异性的MipMap**。RipMap如何实现，目前还不清楚。

### 6.2.3 体纹理

### 6.2.4 立方体映射

### 6.2.5 纹理Caching

### 6.2.6纹理压缩

## 6.3程序纹理（过程式纹理）

**噪声的定义：**噪声由以下几个条件共同定义，

1. 噪声是一个连续且可导的函数。
2. 噪声不应该携带任何明显的规则，即具备随机性
3. 对于相同的输入，噪声函数必须有相同的输出。即伪随机性。

噪声可以分为**值噪声**和**梯度噪声**

**值噪声**

如何构造一个值噪声函数（一维情况）：

1. 对于x=i 的整数位置上随机生成函数值，得到点（i,random）
2. 对（i,random）之间的地方进行平滑插值。

**插值技术**

一个线性插值可以理解为

其中插值系数。上述的线性插值形式又称为一阶牛顿插值法。

由于线性插值容易导致在端点处不可导，因此希望构建f(x)与x的高次多项式关系。在柏林噪声的插值计算中通过以下方式计算。

因为当t执行高次幂的时候x也变成了高次幂。因此通过上式的计算，f(x)与x变成的5次函数的关系。

柏林噪声中的多项式插值并不是标准的n+1个数据点产生一个n次多项式的插值。这个方法只是构造了一个通过给定两点的高次多项式。

**为什么插值函数比更好？**

因为（1）式的二阶导f’’(0)=f’’(1)而（2）式中没有这个性质。而这个性质在计算法线位移时特别有用。

**梯度噪声**

梯度噪声是用梯度去构造一个值，来代替值噪声中随机指定的值。

**柏林噪声**

设格点为V，格点处的梯度为G，要求柏林噪声的位置为P。则柏林噪声设（P-V）\*G作为V处的函数值，然后再做插值。

**使用柏林噪声生成地形**

**HeightMap 高度图**

高度图中记载了每个顶点的位移（不仅仅是高度）

## 6.4纹理动画

1. 用来作纹理的不仅仅可以是单张图像，也可以是视频源（多帧图像）。

2. 纹理坐标也是可以变化的。

## 6.5 材质映射

## 6.6 alpha 映射

## 6.7 Bump映射

**切线空间：**

首先，切线空间是一个坐标空间，有着他自己的XYZ坐标轴。

其次，切线空间是相对于一个平面来说的。选取这个平面上的法线作为切线空间的Z轴，然后再在这个平面上选取相互垂直的两个轴作为XY轴即可构成切线空间。在切线空间中，一般将X轴称为切线轴（T），将Y轴称为副切线轴（B）。但是在法线映射的应用中，为了使平面上任意一点的在法线贴图上的采样坐标与普通的纹理贴图的采样坐标一致，必须使切线空间的两个坐标轴与普通纹理贴图中的UV轴同向。一般习惯上使切线轴与U轴同向，V轴与副切线同向。

根据以上要求，可以根据平面中3点求出该平面的法线，再结合这3点的UV坐标可以求出切线和副切线。

设V0V1V2是平面上逆时针的三点。

将上述两式改写成矩阵形式

根据上式可以解出T，B。

对于一个顶点的切线空间，即这个顶点所在的三角形面片的切线空间。

矩阵的变换关系类推导似于视图变换。可以推出最后的变换矩阵形式为

如果，V T B N使用其他坐标系的坐标，该矩阵式也可以将其他空间的点转换到切线空间。

### 6.7.1 Blin的方法

### 6.7.2 法线映射

**为什么不直接存储对象空间的法线值？**

1. 存储在对象空间不够灵活。假设有一个片元从对象空间取一个法线是n。然后这个面被旋转了，这个时候我们从对象空间中去取法线得到的依然还是n（因为uv坐标没有变），然而由于旋转过后该片元的法线应该也是要跟着旋转的，所以其法线不应该还是n。为了得到正确的结果我们需要把读出来的法线也跟着旋转。所以说将法线存储在对象空间不够灵活。
2. 如果物体发送了非刚性形变，则需要重新更新法线贴图。而存储在切线空间则不用。

**注意：**虽然切线空间是以一个顶点，和他的切线，副切线，法线来共同定义的，但是一个三角形面片共一个切线空间。

**在切线空间计算法线映射还是世界空间计算法线映射？**

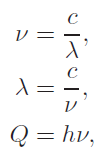
在切线空间计算法线映射只需要将视线和光线变换到切线空间即可，而在世界空间计算则需要将每个法线值转换到世界空间。一把情况下光源数都要比三角形覆盖的片元数要少很多。因此在世界空间计算法线映射计算量要比切线空间计算大很多。但是在世界空间计算有更大的通用性，因为有很多其他的效果是需要在世界空间计算的，比如说，反射。

### 6.7.3 视差贴图

# 7高级着色

## 7.1 辐射度理论

光子有波长，频率,能量Q。其关系如下



C是光速，h是普朗克常量

**辐射通量**=**辐射功率** 光源单位时间内发射的能量。符号

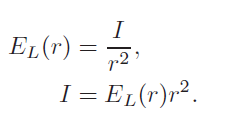
**辐射照度**：单位面积上的辐射功率。

一般情况下将辐射照度（irradiance）看作是射入一个表面的，将辐射度（radiosity）看作是射出表面的。

**辐射通量强度**（radiant flux density）同时指代上述两者。

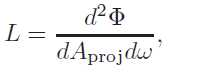
**辐射强度**（radiant intensity）单位立体角上的辐射功率。

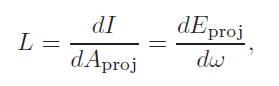
因为对于点光源来说，辐射照度正比于。而比例系数就是辐射强度。



**立体角：**二维角度在三维空间上的扩展。大小等于该交与一个单位球相交所得的面积。立体角的符号是sr

**辐射率**（radiance）：单位面积单位立体单位立体角上的辐射功率。辐射率一般用来考察一个表面接收到的辐射





## 7.2 光子理论

## 7.3 色彩理论

## 7.4 光源类型

### 7.4.1 Omni 光源

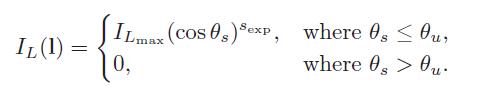
在各个方向上的辐射强度都相同的点光源，称为Ommi光源。

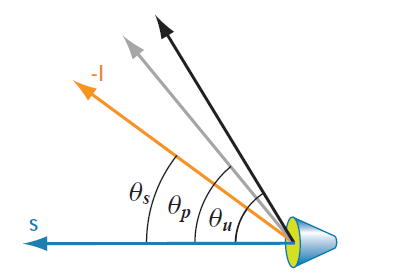
**距离衰减函数**（distance falloff functions）：用来描述辐射照度如何随着距离衰减



在物理上衰减函数为。

### 7.4.2 聚光灯





越靠近聚光灯s的方向上的辐射率越高。

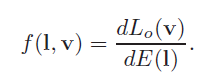
### 7.4.3 纹理光源

### 7.4.4 其他光源

## 7.5 BRDF理论

### 7.5.1BRDF

BRDF定义为出射辐射率与入射辐射照度之比



BRDF可以在RGB3个分量上各不相同。因此BRDF是一个3维向量。BRDF的单位是

各项同性的BRDF，当l，v绕着法线旋转时，BRDF结果不变。

次表面散射：射入物体内部的光经过吸收，散射，再次射出物体表面。

**注意**：Phong和Blin-Phong是传统的经验着色模型。BRDF是PBR范畴的东西。

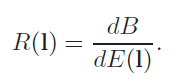
**BRDF**的特征

任何一个BRDF必须满足一下两个条件

但是，即使BRDF不满足这个条件，也并会引起什么问题。

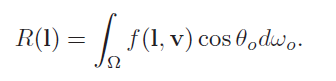
第二个约束是能量守恒。能量守恒指出射能量不可以大于入射能量。可以小于，因为表面可以吸收能量。

**有向半球反射**（directional-hemispherical reflectance）：用来度量一个BRDF能量守恒的程度。对于给定一个方向l,其有向半球反射定义如下



B是出射辐射照度。

有向半球反射与BRDF的关系



O表示出射，out

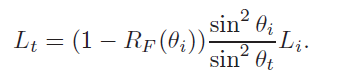
### 7.5.2 表面和主体反射

**表面反射**（surface reflectance）：使用BRDF仿真表面现象

**主体反射**（body reflectance）：使用BRDF仿真内部现象

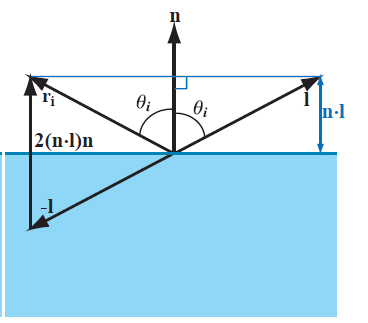
### 7.5.3 菲涅尔反射

入射辐射率和折射辐射率之间的关系如下



反射方向ri可以通过如下方式计算





折射角的计算遵循斯涅耳定律



n1 n2是折射率

**外部反射**

菲涅尔

# 8面光源和环境光照

# 9全局光照

## 9.1 阴影

### 9.1.1 平面阴影

# 10基于图像的特效

## 10.12镜头耀斑（Flare）和泛光（Bloom）

# 11 非真实感渲染

非真实感渲染（NPR）又称为风格化渲染。

## 11.1 Toon Shading

NPR渲染主要要两种方式 cel shading 和Tone Shading（基于色调的渲染方法）。

基于色调的着色是这样的。卡通人物身上的颜色并不是连续变化，而是呈现块状的。因此我们在渲染的时候必须模拟这种不连续性。但是diffuse项它是由法线和光线的点乘构成，这一项的值是连续的，我们必须把它映射到一个连续函数上。方法就是弄一个一维调色板。这个调色板的0-1区间上被划分多多个色块。比如说0-0.1全部映射成一个颜色，0.1-0.2全部映射成一个颜色。然后你在根据法线和光线的点积到这个调色板上去取颜色。这样做diffuse项就被离散化了。就产生了色块的效果。

高光项也是同理。但是高光项一般不做太细致的划分，只做一个阈值分割。就是当法线与半角向量的点积小于一个阈值就返回1，否则就返回0.

## 11.2 Silhouette Edge Rendering

轮廓线的绘制方法大体上可以分为以下几类：基于表面角度的，基于程序化几何体的，基于图像处理的，基于向量边检测的和混合以上方法的。

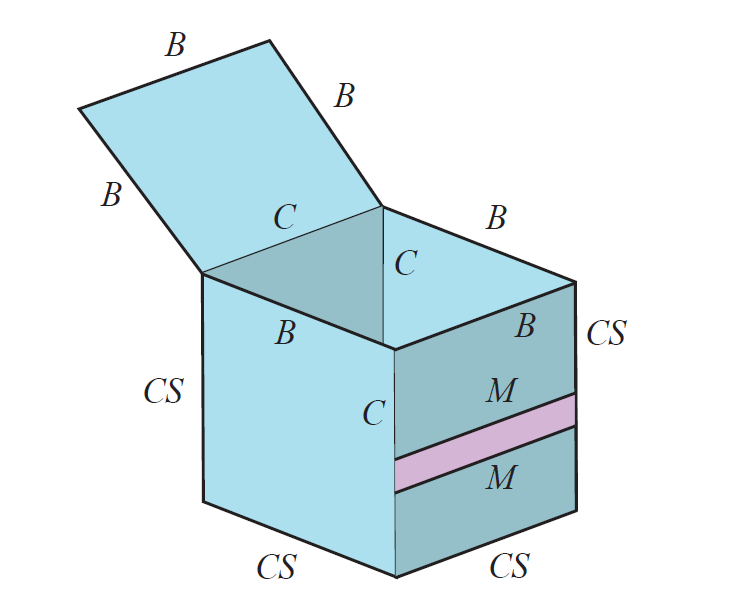
不同的绘制方法绘制出来的轮廓线也不相同，主要分为以下几类：

border edge：该边不会被两个多边形共享。

Hard edge：被两个多边形共享，且两个多边形之间的二面角大于预设值。换句话说构成硬边的两个多边形的法线是不同。硬边可以进一步细分为 ridge edges和valley edges。

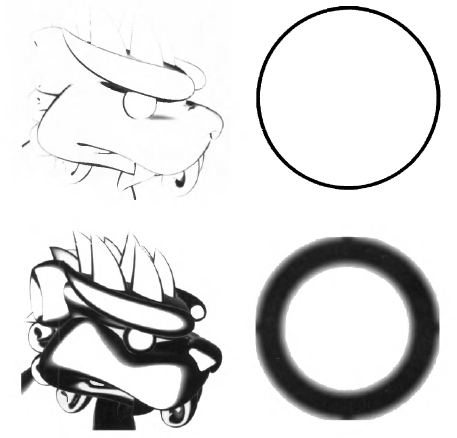
material edge：两个不同材质的三角形的公共边。

silhouette edge：构成该边的三角形面向视线的两侧。通俗点说就是构成该边的三角形一个正面朝向相机，一个背面朝向相机。



### 11.2.1 基于表面角度的轮廓绘制方法

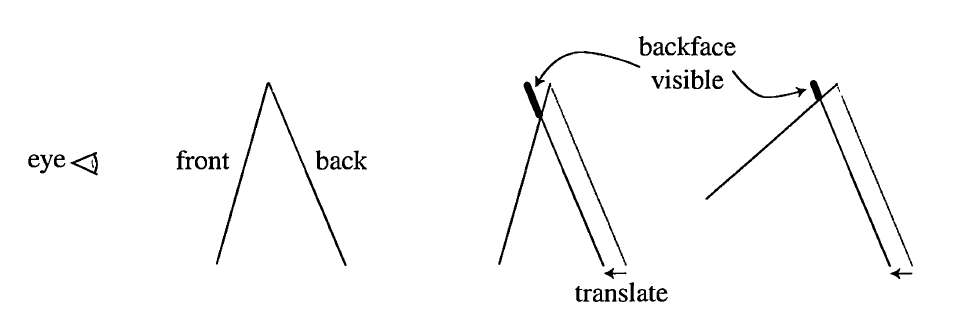
该方法的思路是：如果平面上一个像素的法线和视线的夹角越接近π/2，那么这个像素就越可能是轮廓线上的像素。该方法的特点是轮廓线的宽度受模型表面曲率的影响非常大。具体效果如下图。



### 11.2.2 基于程序化几何体的轮廓线绘制方法。

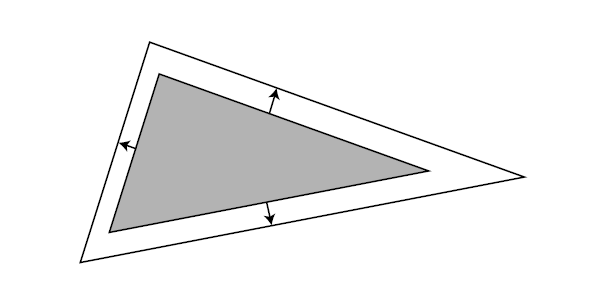
基本思路是第一次正常绘制正面，第二次渲染背面，并以某种方式让轮廓线可见。

其中一种方法是，渲染背面的时候只渲染边，不渲染面。该方案有着很强的性能优势，但是劣势也很明显，该方案绘制的轮廓线只有一个像素的宽度。



第二种方式如上图。在渲染正面以后，将背面向前移动（即，在视图变换后修改Z值），使得背面有一部分在正面的前面，这一部分便是轮廓线。这个方案也有一个明显的特点，它绘制的轮廓线没有统一的宽度，因为这个宽度除了与背面前移的距离有关，还与正面与背面的夹角有关。

一种解决方案是将背面三角形从三条边的方向向外扩展，扩展出来的部分便是轮廓线。在这种情况下轮廓线的宽度只与背面三角形三条边向外扩展的距离有关，去背面三角形和正面三角形的二面角无关。这个方案的一个问题是，当三角形比较小，或者三边向外扩展的距离比较大的时候，三角形三边会出现明显的不相交，这会导致一些地方的轮廓线不连续，出现留白的情况。



另一种解决方案是不把边向外扩展，而是把三角形的三个顶点在平面内向外扩展。

第三种绘制方式是将背面顶点沿着顶点法线向外扩展。这种绘制方式绘制出来的背面像一个壳一样盖在原物体上，因此这种扩展方式又叫做Shell Mapping或者基于Shell的方法。该方案具有实现简单，GPU友好，鲁棒性强和表现稳定等特点。

这个方案也会产生一个问题，当每个顶点使用单独的法线时（比如说正方体本来只有八个顶点，但是我们为了每个顶点使用单独的法线，而使得每个正方体使用24个顶点，这样可以使得顶点法线等于平面法线），这种方法会使得每个三角形均沿着自己的平面法线偏移。这样相邻的面会产生分离。一个解决方法是使用对于顶点法线使用平均法线，而不是单独法线。

### 11.2.3 基于图像处理的轮廓线绘制方法

这个方案的主要思路有：

1. 寻找z值不连续的点来确定轮廓线（举个例子，一个轮廓线的外部是背景，显然背景的z值是远大与轮廓线的，此处轮廓线的z值连续性很差）。
2. 相邻表面的法线变化很大，也是轮廓线的一个标志。
3. 另外直接用环境光来渲染场景可以检测到上述两种方案检测不到的轮廓线。

基于图像的轮廓线绘制方法共有的一个特点是，可以使用图像处理方法如膨胀，腐蚀来调整轮廓线的宽度。

一种具体做法是：首先，在顶点着色阶段把世界空间的法线，和Z值写入到一张纹理贴图中。之后在这张纹理贴图中分别对法线和Z值执行Sobel边缘检测算法去查找轮廓线。最后将查找出来的两组轮廓线合成。这个方案同时考虑了上述思路1,2。无论是基于法线还是基于深度的边缘检测，都有可能会出现对轮廓线检测的不准确，即有可能漏掉或者错误检测出轮廓线。

### 11.2.4 轮廓边检测

轮廓边的定义是：邻接的两个面，其中一个面向观察者，而另一面背向观察者。

数学定义是：

后面没看懂

# 13 曲线和曲面

# 14 加速算法

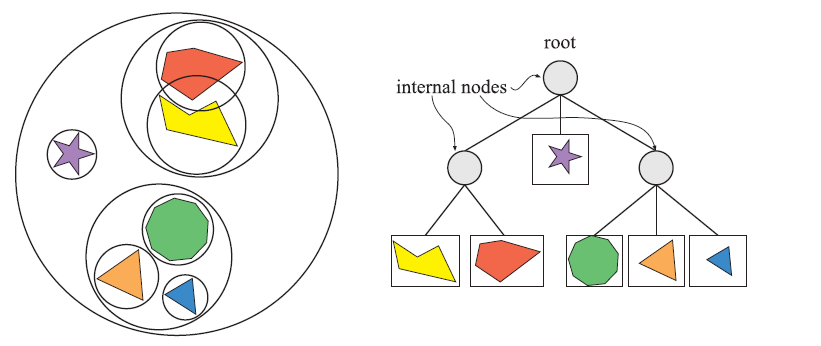
## 14.1 空间数据结构

通用的空间数据结构有：bounding volume hierarchies(BVHs)**层级包围体**，binary space partitioning (BSP) trees**二分空间分割树，**quadtrees**四叉树**, and octrees**八叉树**.

### 14.1.1 BVH层次包围体

**包围体的种类**：**球**，axis-aligned bounding boxes (AABBs)轴对齐包围盒（**AABB包围盒**），oriented bounding boxes (OBBs)方向包围盒（**OBB包围盒**），k-DOPs。

BVH的形象展示



在BVH中每一个结点都有一个包围体，这个包围体包含他所有子树中的所有几何体。叶子结点拥有一个需要被渲染的几何体。

使用BVH可以快速的找到光线与哪些物体相交。先一个结点的包围体做相交测试，如果不相交，则光线与其子节点均不相交；然后光线和下一个结点的包围体做相交测试，如果相交，再和其子结点做相交测试直到叶子结点，可以找出快速找出光线与哪些物体相交。

然而事实上我们并不关心光线与哪些物体相交，我们只关心与光线相交的距离相机最近的物体。我们在求得与光线相交的包围体后，可以求出包围体到相机的距离（包围体与光线的交点到相机的距离，也可以用包围体中心到相机的距离），并在遍历BVH树之后得到距离最小的包围体。

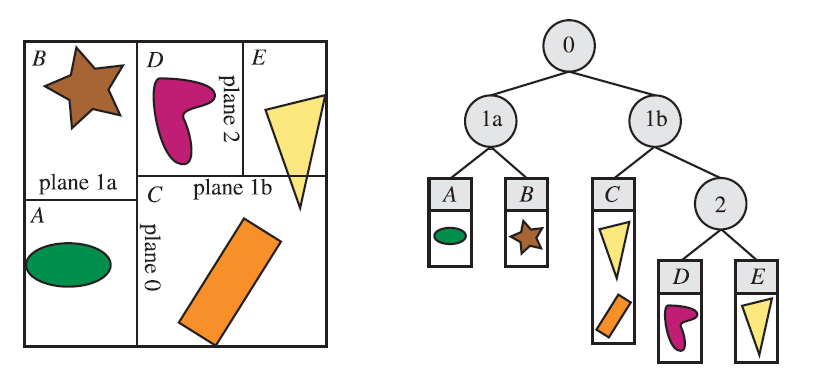
### 14.1.2BSP树 二分空间分割树

BSP树通常有两种类型：**轴对齐的BSP树**（axis-aligned）和**多边形对齐的BSP树**（polygon-aligned）

**轴对齐的BSP树**

构造方法：首先构造一个AABB包围盒包围场景中的所有物体。然后用一个垂直于包围盒的某一个轴的平面将这个包围盒分割成两个部分，一直递归下去，直到递归结束条件满足，停止递归。

轴对齐的BSP树结构如图



如果一个物体和分割平面相交，则它可以放在树的当前节点，或者当前结点的子结点。

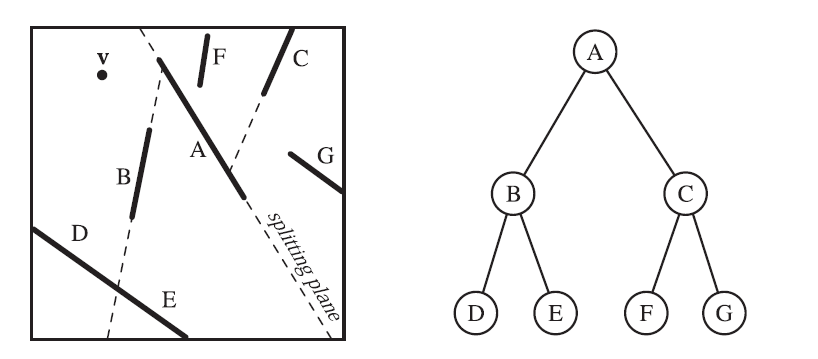
分割平面的选择也是比较讲究的。一个策略是让分割平面依次绕着最大包围盒的x,y,z轴旋转。如分割一个结点时，分割平面垂直x轴，则分割其子结点时，分割平面垂直于其y轴。使用这个方法构造的BSP树又叫K-D树。另一种选取分割平面的方法是使分割平面总是平行于当前包围盒面积最大的那个面。

轴对齐的BSP树可以给出场景中物体一个粗糙的排序。一个平面将包围体分割成两部分，假设相机所在的部分是空间A，另一个部分是空间B。则A中的物体比B中的物体离相机更近（并不那么精确）。

**多边形对齐的BSP树**

从一个多边形的众多三角形所在的平面中选择一个平面将空间分割成两个部分，多边形所在的空间和不在的空间。其他与分割平面相交的物体则分割成两个物体。

一个多边形对齐的BSP树结构如下



BSP树给出来的物体排序并不是严格意义上的物体到相机距离从远到近，而是遮挡关系上的从远到近。比如说，上图的空间结构A左侧的物体更可能挡住右侧的物体。

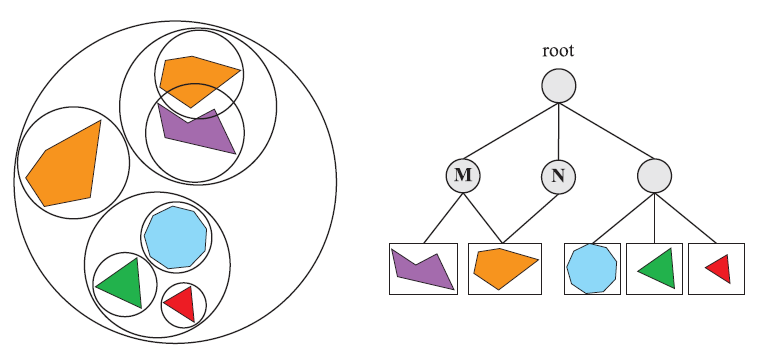
### 14.1.3 Octrees 八叉树

在包围盒的中心沿三个轴方向将包围盒划分成八个新的空间。四叉树是分割二维空间的例子。

与BSP树不同的是，几何体总是存储在八叉树的叶子结点上。

### 14.1.4 Cache相关和Cache无关的表示方法

### 14.1.5场景图



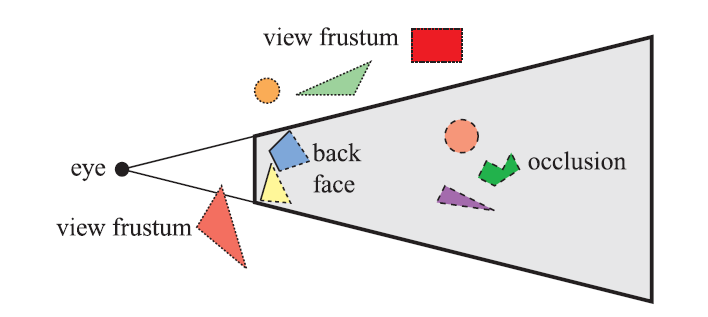
场景图是一个图结构。结点中包含一个变换，叶子结点的变换是从根遍历到叶子所经历的结点的所有变换的综合。只有叶子结点包含几何体。

场景图中每个结点也可以有一个包围体，该包围体包围他所有的子结点。此时场景图构造成了一个BVH结构。

## 14.2 剔除技术

主要的剔除技术又：**背面剔除**，**视景体剔除**和**遮挡剔除**。

三种剔除类型的图示



理想的剔除算法仅仅发送exact visible set (EVS)图元给渲染管线。所谓EVS图元是指完全或部分可见的图元。

实现理想剔除算法所需要的数据结构是aspect graph，但是构造这种数据结构并不划算，因为他最坏时间复杂度是O（n^9）

实际的剔除算法总是给出潜在的可见性集合potentially visible set (PVS)。

如果一个潜在可见集包含精确可见集，则成为保守PVS，否则称为近似PVS。

### 14.2.1 背面剔除

1如果一个三角形在世界空间是逆时针，但是投影到成像平面后变成了顺时针，则该三角形式背面。

2在三角形上任选一点，与相机构成一个向量，求向量与平面法线的夹角，如果夹角大于180，则该面是背面

方案1和2等价。

背面剔除是渲染器的一个功能，集成在OpenGL和DirectX中。

**剔除背面簇**：由CPU完成背面剔除，所有的背面都不发送给渲染管线。因为一次会剔除所有的背面，因此叫剔除背面簇。

## 14.3 层级视景体剔除

视景体剔除的主要工作：剔除视景体外的物体

**基于BVH的视景体剔除**

遍历BVH，如果一个结点的BV完全在视景体之外，则该结点极其子树被剪枝；如果一个结点的BV完全在视景体内，则其子树中包含的物体全部提交渲染管线，并且不再对其子树做BV相交测试；如果一个结点的BV与视景体相交，则其子结点仍然做相交测试，如果这是一个叶子结点，则该物体提交给渲染管线。

**基于BSP树 八叉树等空间分割技术的视景体剔除**

遍历八叉树，如果一个盒子完全在视景体内部，则其中的物体全部提交渲染；如果完全在外部，则其子树全部被剪枝；如果相交，则继续分割下去，如果是叶子结点，则其中的物体也全部提交渲染。

## 14.4 Portal Culling

Portal剔除是遮挡剔除中的一种。

在室内链接各个房间的门窗称为portal。

（具体怎么做的没有看懂。以后细看）

## 14.6 遮挡剔除

背面剔除也是遮挡剔除的一种形式。

遮挡剔除算法有两种方式：**基于点**的和**基于Cell**的

遮挡剔除算法的另一种分类方式：**图像空间**，**对象空间**，**光线空间** 的遮挡剔除

### 14.6.1 硬件遮挡查询

（没看懂）

### 14.6.2 层级Z-Buffer

# 15流水线优化

## 15.1 Profiling Tools

## 15.2 定位瓶颈

没看

## 15.3 表现度量

# 18 图形硬件

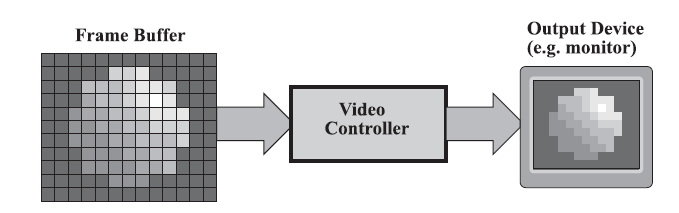
## 18.1 缓冲区和缓冲

所有的Buffer合在一起组成Frame Buffer

帧缓存可以位于内存中，或者有专门的帧缓存硬件，又或者直接放在显存中。

颜色缓存是帧缓存的一种。

颜色缓存通过视频控制器与显示器相连。



### 18.1.1 颜色缓存

颜色缓冲区类型:

**高彩色**：2字节每像素

**真彩色**：3字节每像素或4字节每像素（多出来的一个字节用于内存对齐）

### 18.1.2 深度缓冲

深度缓冲中保存的值：**片元到投影面的深度**。（0.0--1.0）

计算方法为**先对齐次空间的Z值做透视插值，得到齐次空间的Z值，然后用Z/W获取屏幕空间的深度值**。

不是对1/z插值。如果是对1/z插值那么有z属于0-1,则1/z为1-正无穷。而事实是深度缓冲中的值为0-1。

3d游戏编程大师技巧中使用1/z缓存也是为了做透视插值

**Z-Fighting**

离投影面近的点在深度中占用较高的精度，而远的点占用较低的精度。

解决方案：

1. 增加近裁剪面与相机的距离。
2. Polygon Offset（暂时不知道是什么方法）

### 18.1.3 Accumulation Buffer

A-Buffer 对于每一个采样值乘以一个权重，累加到颜色缓冲区中去。

### 18.1.4 Stencil Buffer

如果我们只想渲染屏幕的某些区域，可以通过设置Stencil Buffer来完成。Stencil Buffer的作用就是通过人工来设置Stencil值，来确定一个片元是保留还是丢弃。Stencil Buffer存在的原因就是人们希望可以人工的干预一个片元是否要被写入到颜色缓冲中。

### 18.1.5 剪切区域

剪切区域是一个矩形区域。区域内的渲染结果会被写入到颜色缓冲区，区域外的渲染结果会直接被丢弃。

## 18.2 透视矫正插值

因为光栅化操作的缘故，我们在屏幕空间做的插值应该是线性插值，即

其中，是屏幕空间的点。

而我们希望得到的变换其实是齐次空间（也是世界空间）的线性插值,即

其中是齐次空间的点。由于透视投影的关系，。（这个很好理解，比如在世界空间里一条直线上等距的3个点，投影到投影面上后，3个点不一定等距。）

因此我们需要做的是找到与的关系。

我们先将齐次空间的点除以w，转入屏幕空间下。

上式左边有于是，将右边也转换到屏幕空间的形式。

将上式右边化为的形式。因此有

所以

因此可以解出，

假设我们需要对颜色C插值。因为想要在齐次空间做线性插值，所以现在有

但是我们实际上能拿到的值是，因此把代入上式。

实际上到这里已经被计算出来了。但是为了硬件实现的方便，在此对上式做一定的变形。先通分，再分子分母同时除以。可将上式化为

上式表明，要利用，求,可以先在和之间做插值，然后再除以和的插值。因为线性插值可以通过做增量的方式去计算，因此下式会比上式计算快速的多。

## 18.3 架构

### 18.3.1 通用架构

**流水线和并行**

# 其他的注意事项

RGB颜色空间是一个非线性的空间，严禁对RGB颜色插值。

# 逐顶点着色和逐像素着色的区别

1. 因为逐顶点着色以后会对颜色做线性插值，而RGB颜色是非线性变换，所以会得到不正确的结果。
2. 在顶点着色器中不应该将非线性变换的值输出用来插值。比如说，两个向量的点积。

# 欧拉公式三角形式的推导

首先设有一个复数z

z=cosθ+isinθ

等式两边取微分，

dz=(−sinθ+icosθ)dθ

右边提一个i出来

dz=i(cosθ+isinθ)dθ

而右边括号内的数正好是z

dz=izdθ

L两边同时积分

lnz=iθ

# 傅里叶变换

傅里叶变换首先要从傅里叶级数谈起。傅里叶级数是说，对于任意一个周期函数f(x)，我们都可以用一组正交基函数去表示它，即

现在的想法是这种方法扩展到非周期函数上去。因为一个非周期函数都可以看做是一个周期无穷大的函数。因此对于一个非周期函数f(x)其傅里叶级数形式有相同的形式。

因为非周期函数f(x)定义域为（）。因此有如下形式

令，则可以改写为

现在定义一个傅里叶函数

不难看出

现在将傅里叶级数中的系数用傅里叶函数来代替

把L用来代替。

现在令L->。f(x)可以改写成积分形式

上式告诉我们，我们可以根据傅里叶函数重建出原函数f(x)。

现在总结一下，f(x)和他的傅里叶函数F(x)有如下关系

现在再分析一下什么是傅里叶变换。

我们把一个函数f(x)分解成无穷个基函数（）的和，而傅里叶函数的值是其对应的基函数（）的系数。

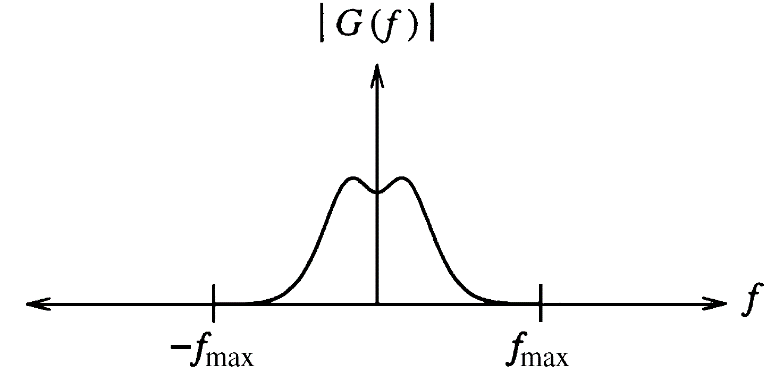
# 采样定理的证明

对于一个信号x(t)，采样过程可以描述为x(t)乘以一个脉冲序列函数, 脉冲序列函数如下。是一个Dirac delta函数，也叫冲击响应函数，该函数在t=0时，值为无穷,t!=0时，函数值为0。

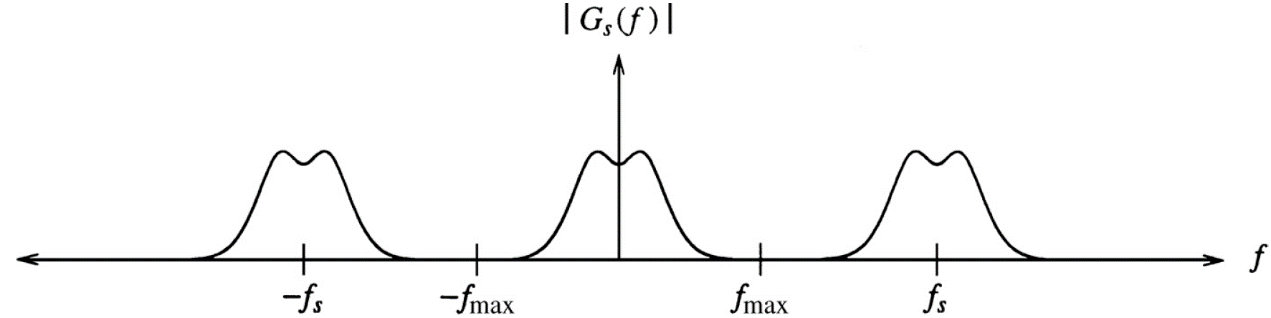
把采样结果表示为,

x(t)的傅里叶函数为X().采样结果的傅里叶函数为S()。可以计算出S()与X()的关系。

其中.**注意，**X()只在,之间有值，所以可以直接看作是n个X()的和。若X()的函数图像如下，



则，的函数图像如下。



当采样频率fs>2fmax时，中的各个X()分量互相不黏连。因此可以从分离出X()。当fs>2fmax时，要分离出X()，只需要做一个乘法操作。

其中

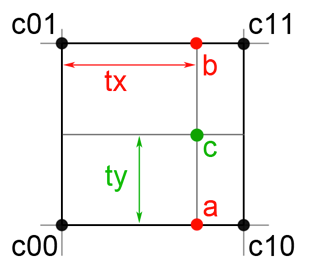
其中称为理想低通滤波器，低频信号通过，高频丢弃。根据傅里叶变换的性质，上式在时间域可以表示为。

其中，\*为卷积符号，h(t)为对应的时间域的函数。从上式已经根据采样信号s(t)，使用一个低通滤波器h(t)复原出了原始信号x(t)。根据傅里叶变换，可以计算出h(t)=sinc(t)。

另外：带限函数的定义域必定是（-无穷，无穷）。

# 插值技术

双线性插值



为了使用4个格点双线性插值出c,首先利用c00 c10线性插值出a，利用c01 c11线性插值出b,再利用a b线性插值出c.

**注意：**双线性插值并不是线性的，而是二次的。双线性插值展开，存在x\*y项，如下图

