图形学理论笔记

郑华

2019 年 5 月 13 日

目录

第一章　背景知识

1.1　绪论

计算机图形学是利用计算机研究图形的表示、生成、处理、显示的学科。

1.1.1　应用与意义

- 电影：科幻大片的特效
- 游戏：图形地理、人物控制
- 计算机仿真
- CAD/CAM ：设计、制造
- 建筑
- 可视化（科学）：化学结构、生物

1.1.2　研究内容

- 图形硬件、图形标准、图形交互技术
- 光栅图形生成算法
- 曲线曲面造型、实体造型
- 真实感图形绘制、科学计算可视化、计算机动画、自然景物仿真
- 虚拟现实

1.2 涉及概念

图形与图象

图像 计算机内以位图（Bitmap）形式存在的灰度信息。

图形 含有几何属性、更强调场景的几何表示，是由场景的几何模型和景物的物理属性共同组成的。

图形主要分为两种：

- 基于线条信息表示
- 明暗图（**Shading**）

图形学与 CAD

图形学与模式识别

图形学与视觉

1.3 历史

- 细分曲面
- 光栅化图形：区域填充、裁剪、消隐等基本图形概念（70 年代）
    1. 光反射模型 70 年（真实感图形学）
    2. 漫反射模型 + 插值 71 年（明暗处理）
    3. 简单光照模型 75 年（Phong 模型）
    4. 光投射模型 80 年，并给出光线跟踪算法范例（Whitted 模型）
    5. 辐射度方法 84 年

1.4 杂志与会议

会议

- Siggraph
- Eurograph
- Pacific Graphics
- Computer Graphics International.

杂志

- ACM Transaction on Graphics
- IEEE Computer Graphics and Application
- IEEE Visualization and Computer Graphics

1.5 真实感绘制及重要概念

目的是模拟真实物体的物理属性，包括物体的形状，光学性质，表面的纹理和粗燥程度，以及物体间的相对位置，遮挡关系等。

图形学理论
- 光照模型
  - 简单光照模型
  - 局部光照模型
  - 整体光照模型
- 绘制方法
  - 光线追踪
  - 辐射度
- 加速算法
  - 包围盒树、自适应八叉树等
  - 阴影算法、纹理合成

第二章　参考学习路线

2.1　OS 基础

《Programming Windows 5th》只需要看前面几章，理解 Win32 消息循环即可。

《Multithreading Applications in Win32》，理解多线程是怎么回事；

2.2　GUI

WPF、MFC

2.3　3D 图形学

二选一，我推荐 handmadehero，因为有很详细的视频教程。

`https://handmadehero.org/`

`http://frustum.org/3d/`从最开始的 demo 开始，用 DX 全部写一遍。

2.4　基础篇

- 上篇，各种概念与技术。《Fundamentals of Computer Graphics 3rd》or《Computer Graphics - Principles and Practice》
- 中篇，各种算法。《Real-Time Rendering 3rd》
- 下篇，高阶追求。《Physically Based Rendering - From Theory To Implementation 2nd》
- 杂篇 1，只谈数学，《3D Math Primer for Graphics and Game Development 2nd》
- 杂篇 2，同上，《Mathematics for 3D Game Programming and Computer Graphics 3rd》

2.5 API 之术（DX9/11 or OpenGL）

- 《3D 绘图程序设计》，dx9/10/ogl 实现各种 3D 游戏需要的效果
- 《More OpenGL Game Programming》，如何用 gl 实现各种中高级效果
- 《Introduction to 3D Game Programming with DirectX 9.0c - A Shader Approach》，龙书
- 《Introduction to 3D Game Programming with DirectX 11》，龙书 DX11 版
- 《Practical Rendering and Computation with Direct3D 11》，DX11 宝典
- 《OpenGL Programming Guide》，OpenGL 宝典
- 《OpenGL Shading Language》，glsl 宝典

2.6 引擎设计

David Eberly 的《3D Game Engine Design》，WildMagic，这有一个完整的引擎实现

Jason Gregory 的《Game Engine Architecture》

《Character Animation with Direct3D》，专注讲 model、animation 的。

2.7 各种技巧

ShaderX 系列《Real Time Collision Detection》《Real Time Cameras》《Real-Time Shadows》

第三章　渲染管线

在给定虚拟相机、三维物体、光源、照明模式，以及纹理等诸多条件的情况下，生成或绘制出一幅 2 维图像的过程。

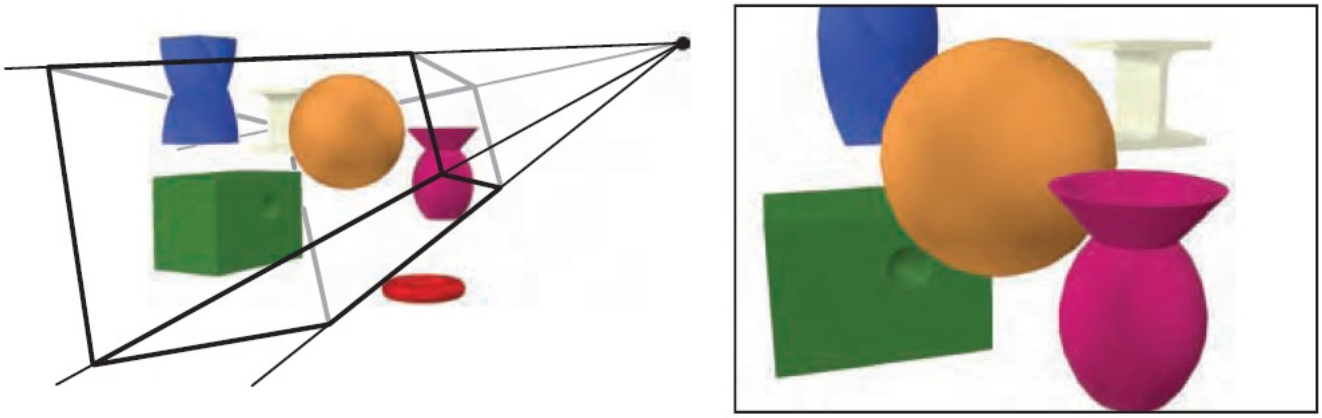

Fig 3.1: 渲染概述-三维到二维

而大体可以分为三部分：

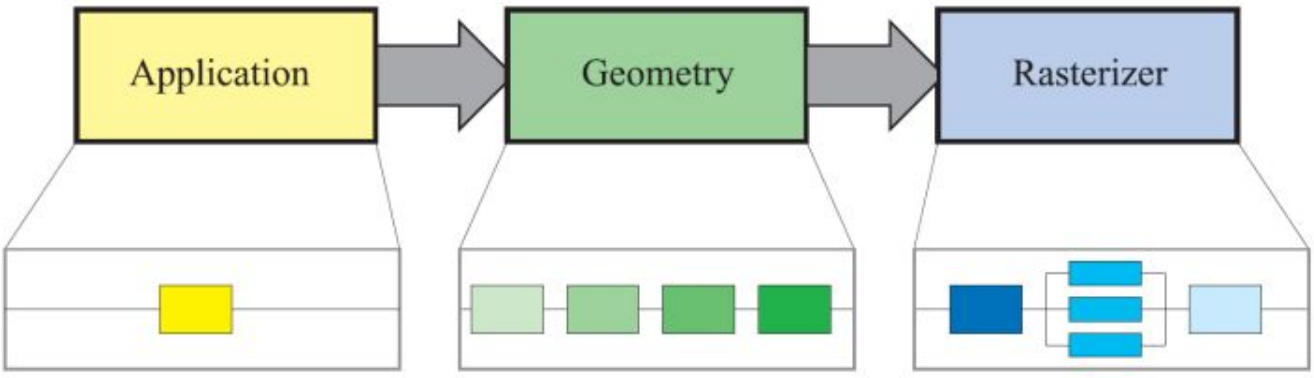

Fig 3.2: 渲染结构

3.1 应用阶段 - The Application Stage

应用程序阶段一般是图形渲染管线概念上的第一个阶段。应用程序阶段是通过软件方式来实现的阶段，开发者能够对该阶段发生的情况进行完全控制，可以通过改变实现方法来改变实际性能。其他阶段，他们全部或者部分建立在硬件基础上，因此要改变实现过程会非常困难。

正因应用程序阶段是软件方式实现，因此不能像几何和光栅化阶段那样继续分为若干个子阶段。但为了提高性能，该阶段还是可以在几个并行处理器上同时执行。在 CPU 设计上，称这种形式为超标量体系（superscalar）结构，因为它可以在同一阶段同一时间做不同的几件事情。

应用程序阶段通常实现的方法有**碰撞检测、加速算法、输入检测，动画，力反馈**以及**纹理动画，变换仿真、几何变形**，以及一些不在其他阶段执行的计算，如**层次视锥裁剪等加速算法**就可以在这里实现。

应用程序阶段的主要任务：在应用程序阶段的末端，将需要在屏幕上（具体形式取决于具体输入设备）显示出来绘制的几何体（也就是绘制图元，rendering primitives，如点、线、矩形等）输入到绘制管线的下一个阶段。

对于被渲染的每一帧，应用程序阶段将摄像机位置，光照和模型的图元输出到管线的下一个主要阶段——几何阶段。

3.2 几何阶段 - The Geometry Stage

负责大部分多边形操作和顶点操作。

Fig 3.3: 几何阶段演示

3.2.1 模型视图变换阶段 - Model and View Transform

模型变换的目的是将模型变换到适合渲染的空间当中，而视图变换的目的是**将摄像机位置放置于坐标原点**，方便后续步骤的操作。

在屏幕上的显示过程中，模型通常需要变换到若干不同的空间或坐标系中。模型变换的变换对象一般是模型的顶点和法线。物体的坐标称为模型坐标。世界空间是唯一的，所有的模型经过

变换后都位于同一个空间中。

不难理解，应该仅对相机（或者视点）可以看到的模型进行绘制。而相机在世界空间中有一个位置方向，用来放置和校准相机。

为了便于投影和裁剪，必须对相机和所有的模型进行视点变换。变换的目的就是要把相机放在原点，然后进行视点校准，使其朝向 Z 轴负方向，y 轴指向上方,x 轴指向右边。在视点变换后，实际位置和方向就依赖于当前的 API。我们称上述空间为**相机空间或者观察空间**。

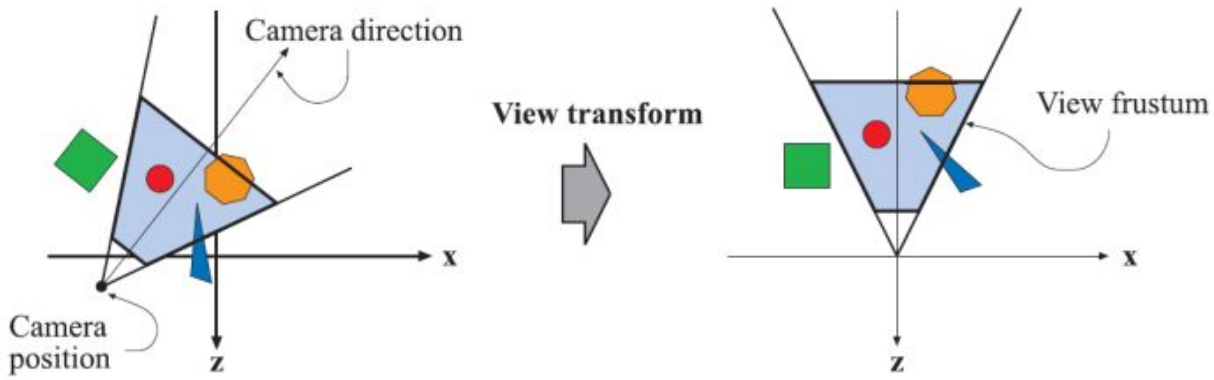

Fig 3.4: 视图变换

在左图中，摄像机根据用户指定的位置进行放置和定位。在右图中，视点变换从原点沿着 Z 轴负方向对相机重新定位，这样可以使裁剪和投影操作更简单、更快速。可视范围是一个平截椎体，因此可以认为它是透视模式。

总结 模型和视图变换阶段分为模型变换和视图变换。**模型变换**的目的是将模型变换到适合渲染的空间当中，而**视图变换**的目的是将摄像机放置于坐标原点，方便后续步骤的操作。

3.2.2 顶点着色阶段 - Vertex Shading

顶点着色的目的在于**确定**模型上的**顶点处**材质的**光照效果**及**颜色**值。

确定材质上的光照效果的这种操作被称为着色（shading），着色过程涉及在对象上的各个点处计算**着色方程**（shading equation）。

通常，这些计算中的一些在几何阶段期间在模型的顶点上执行（vertex shading），而其他计算可以在每像素光栅化（per-pixel rasterization）期间执行。**可以在每个顶点处存储各种材料数据，诸如点的位置，法线，颜色或计算着色方程所需的任何其它数字信息。顶点着色的结果**（其可以是颜色，向量，纹理坐标或任何其他种类的阴着色数据）计算完成后，**会被发送到光栅化阶段以进行插值操作。**

通常，着色计算通常认为是在世界空间中进行的。在实践中，有时需要将相关实体（诸如相机和光源）转换到一些其它空间（诸如模型或观察空间）并在那里执行计算，也可以得到正确的结果。这是因为如果着色过程中所有的实体变换到了相同的空间，**着色计算中需要的诸如光源，相机和模型之间的相对关系是不会变的**。

3.2.3 投影阶段 - Projection

就是将模型从三维空间投射到二维空间中的一个过程。

在光照处理之后，渲染系统就开始进行投影操作，即将视体变换到一个对角顶点分别是 (-1,-1,-1) 和 (1,1,1) 单位立方体（unit cube）内，这个单位立方体通常也被称为规范立方体（Canonical View Volume，CVV）。目前，主要有两种投影方法，即：

- 正交投影：（orthographic projection，或称 parallel projection）
- 透视投影：（perspective projection）。

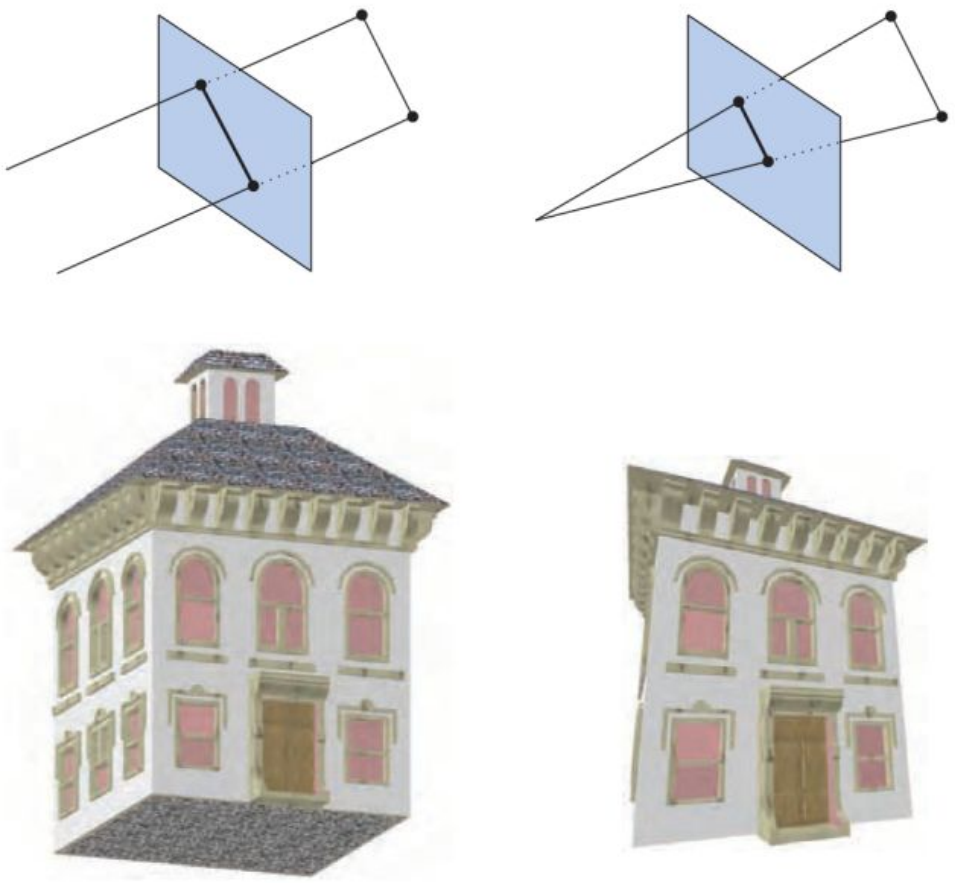

Fig 3.5: 投影演示

左边为正交投影，右边为透视投影。**正交投影**的可视体通常是一个矩形，正交投影可以把这

个视体变换为单位立方体。正交投影的主要特性是平行线在变换之后彼此之间仍然保持平行，这种变换是平移与缩放的组合。**透视投影**比正交投影复杂一些。在这种投影中，越远离摄像机的物体，它在投影后看起来越小。更进一步来说，平行线将在地平线处会聚。透视投影的变换其实就是模拟人类感知物体的方式。

正交投影和透视投影都可以通过 4 x 4 的矩阵来实现，在任何一种变换之后，都可以认为**模型位于归一化处理之后的设备坐标系中**。

虽然这些矩阵变换是从一个可视体变换到另一个，但它们仍被称为投影，因为在完成显示后，**Z 坐标将不会再保存于的得到的投影图片中**。通过这样的投影方法，**就将模型从三维空间投影到了二维的空间中**。

总结　投影阶段就是将模型从三维空间投射到了二维的空间中的过程。

3.2.4　裁剪阶段 - Clipping

就是对部分位于视体内部的图元进行裁剪操作。

只有当图元完全或部分存在于视体（也就是上文的规范立方体，CVV）内部的时候，才需要将其发送到光栅化阶段，这个阶段可以把这些图元在屏幕上绘制出来。

不难理解，一个图元相对视体内部的位置，分为三种情况：完全位于内部、完全位于外部、部分位于内部。所以就要分情况进行处理：

- 当图元完全位于视体内部，那么它可以直接进行下一个阶段。
- 当图元完全位于视体外部，不会进入下一个阶段，可直接丢弃，因为它们无需进行渲染。
- 当图元部分位于视体内部，则需要对那些部分位于视体内的图元进行裁剪处理。

对部分位于视体内部的图元进行裁剪操作，这就是裁剪过程存在的意义。裁剪过程见下图。

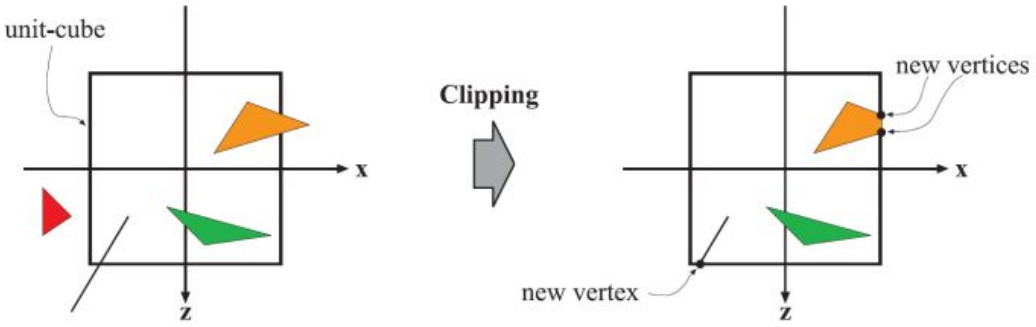

Fig 3.6: 裁剪演示

投影变换后，**只对单位立方体内的图元**（相应的是视锥内可见图元）**继续进行处理**，因此，**将单位立方体之外的图元剔除掉，保留单位立方体内部的图元，同时沿着单位立方体将与单位立方体相交的图元裁剪掉**，因此，就会产生新的图元，同时舍弃旧的图元。

3.2.5 屏幕映射阶段 - Screen Mapping

就是将之前步骤得到的坐标映射到对应的屏幕坐标系上。

只有在视体内部经过裁剪的图元，以及之前完全位于视体内部的图元，才可以进入到屏幕映射阶段。进入到这个阶段时，**坐标仍然是三维的**（**但显示状态在经过投影阶段后已经成了二维**），每个图元的 x 和 y 坐标变换到了屏幕坐标系中，屏幕坐标系连同 z 坐标一起称为窗口坐标系。

假定在一个窗口里对场景进行绘制，窗口的最小坐标为（x1，y1），最大坐标为（x2，y2），其中 x1<x2，y1<y2。屏幕映射首先进行平移，随后进行缩放，在映射过程中 z 坐标不受影响。新的 x 和 y 坐标称为屏幕坐标系，与 z 坐标一起（-1 z 1）进入光栅化阶段。如下图：

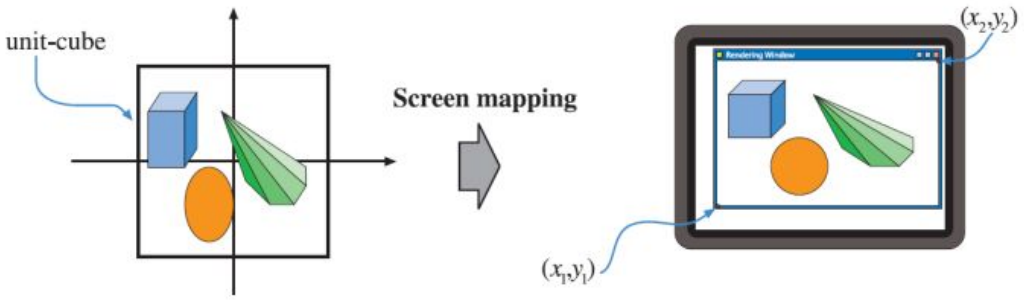

Fig 3.7: 屏幕映射演示

经过投影变换，图元全部位于单位立方体之内，而屏幕映射主要目的就是找到屏幕上对应的坐标

屏幕映射阶段的一个常见困惑是整型和浮点型的点值如何与像素坐标（或纹理坐标）进行关联。可以使用 Heckbert[书后参考文献第 520 篇] 的策略，用一个转换公式进行解决。

总结 屏幕映射阶段的主要目的，就是将之前步骤得到的坐标映射到对应的屏幕坐标系上。

3.3 光栅化阶段 - The Rasterizer Stage

给定经过变换和投影之后的顶点，颜色以及纹理坐标，给每个像素正确配色，以便正确绘制整幅图像。

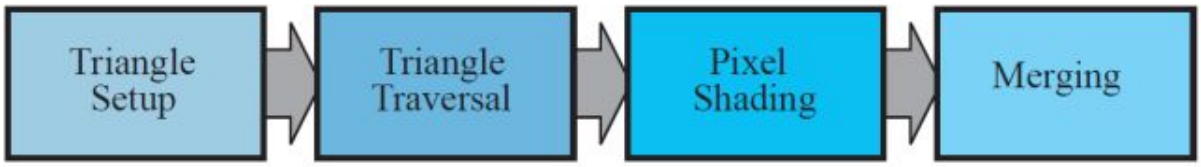

Fig 3.8: 光栅化演示

3.3.1 三角形设定阶段 - Triangle Setup

三角形设定阶段主要用来计算三角形表面的差异和三角形表面的其他相关数据。

该数据主要用于扫描转换（scan conversion），以及由几何阶段处理的各种着色数据的插值操作所用。该过程在专门为其设计的硬件上执行。

3.3.2 三角形遍历阶段 - Triangle Traversal

在三角形遍历阶段将进行**逐像素检查操作，检查该像素处的像素中心是否由三角形覆盖，而对于有三角形部分重合的像素，将在其重合部分生成片段（fragment）**。

找到哪些采样点或像素在三角形中的过程通常叫三角形遍历（TriangleTraversal）或扫描转换（scan conversion）。**每个三角形片段的属性均由三个三角形顶点的数据插值而生成**。这些属性包括片段的深度，以及来自几何阶段的着色数据。

3.3.3 像素着色阶段 - Pixel Shading

所有逐像素的着色计算都在像素着色阶段进行，使用插值得来的着色数据作为输入，输出结果为一种或多种将被传送到下一阶段的颜色信息。纹理贴图操作就是在这阶段进行的。

像素着色阶段是在可编程 GPU 内执行的，在这一阶段有大量的技术可以使用，其中最常见，最重要的技术之一就是纹理贴图（Texturing）。纹理贴图在书的第六章会详细讲到。简单来说，纹理贴图就是将指定图片“贴”到指定物体上的过程。而指定的图片可以是一维，二维，或者三维的，其中，自然是二维图片最为常见。如下图所示：

Fig 3.9: 片段着色器演示

左上角为一没有纹理贴图的飞龙模型。左下角为一贴上图像纹理的飞龙。右图为所用的纹理贴图。

3.3.4 融合 - Merging

每个像素的信息都储存在**颜色缓冲器**中，而颜色缓冲器是一个颜色的矩阵列（每种颜色包含红、绿、蓝三个分量）。融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。不像其它着色阶段，通常运行该阶段的 GPU 子单元并非完全可编程的，但其高度可配置，可支持多种特效。

此外，这个阶段还负责可见性问题的处理。这意味着当绘制完整场景的时候，颜色缓冲器中应该还包含从相机视点处可以观察到的场景图元。对于大多数图形硬件来说，这个过程是通过 Z 缓冲（也称**深度缓冲器**）算法来实现的。Z 缓冲算法非常简单，具有 O(n) 复杂度（n 是需要绘制的像素数量），只要对每个图元计算出相应的像素 z 值，就可以使用这种方法，大概内容是：

Z 缓冲器器和颜色缓冲器形状大小一样，每个像素都存储着一个 z 值，这个 z 值是从相机到最近图元之间的距离。每次将一个图元绘制为相应像素时，需要计算像素位置处图元的 z 值，并与同一像素处的 z 缓冲器内容进行比较。如果新计算出的 z 值，远远小于 z 缓冲器中的 z 值，那么说明即将绘制的图元与相机的距离比原来距离相机最近的图元还要近。这样，像素的 z 值和颜色就由当前图元对应的值和颜色进行更新。反之，若计算出的 z 值远远大于 z 缓冲器中的 z 值，那么 z 缓冲器和颜色缓冲器中的值就无需改变。

上面刚说到，颜色缓冲器用来存储颜色，z 缓冲器用来存储每个像素的 z 值，还有其他缓冲

器可以用来过滤和捕获片段信息。

- 比如 **alpha 通道**（alpha channel）和**颜色缓冲器**联系在一起可以存储一个与每个像素相关的不透明值。可选的 alpha 测试可在深度测试执行前在传入片段上运行。片段的 alpha 值与参考值作某些特定的测试（如等于，大于等），如果片断未能通过测试，它将不再进行进一步的处理。alpha 测试经常用于不影响深度缓存的全透明片段（见 6.6 节）的处理。

- **模板缓冲器**（stencil buffer）是用于记录所呈现图元位置的离屏缓存。每个像素通常与占用 8 个位。图元可使用各种方法渲染到模板缓冲器中，而缓冲器中的内容可以控制颜色缓存和 Z 缓存的渲染。举个例子，假设在模版缓冲器中绘制出了一个实心圆形，那么可以使用一系列操作符来将后续的图元仅在圆形所出现的像素处绘制，类似一个 mask 的操作。模板缓冲器是制作特效的强大工具。而在管线末端的所有这些功能都叫做光栅操作（raster operations，ROP）或混合操作（blend operations）。

- **帧缓冲器**（frame buffer）通常包含一个系统所具有的所有缓冲器，但有时也可以认为是颜色缓冲器和 z 缓冲器的组合。

- **累计缓冲器**（accumulation buffer），是 1990 年，Haeberli 和 Akeley 提出的一种缓冲器，是对帧缓冲器的补充。这个缓冲器可以用一组操作符对图像进行累积。例如，为了产生运动模糊（motion blur.，可以对一系列物体运动的图像进行累积和平均。此外，其他的一些可产生的效果包括景深（e depth of field），反走样（antialiasing）和软阴影（soft shadows）等。

而当图元通过光栅化阶段之后，从相机视点处看到的东西就可以在荧幕上显示出来。为了避免观察者体验到对图元进行处理并发送到屏幕的过程，图形系统一般使用了双缓冲（double buffering）机制，这意味着屏幕绘制是在一个后置缓冲器（backbuffer）中以离屏的方式进行的。一旦屏幕已在后置缓冲器中绘制，后置缓冲器中的内容就不断与已经在屏幕上显示过的前置缓冲器中的内容进行交换。注意，只有当不影响显示的时候，才进行交换。

总结 融合阶段的主要任务是合成当前储存于缓冲器中的由之前的像素着色阶段产生的片段颜色。此外，融合阶段还负责可见性问题（Z 缓冲相关）的处理。

3.4 可编程部分演示

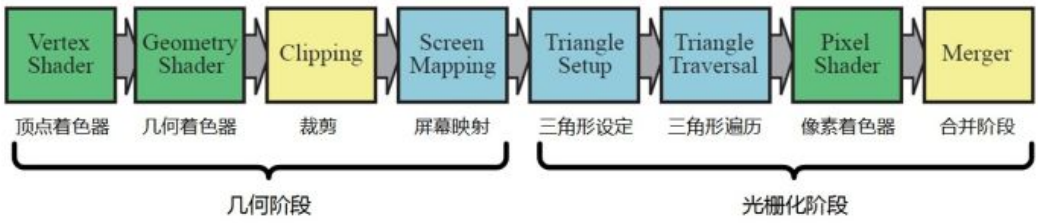

Fig 3.10: 可编程阶段演示

其中，绿色的阶段表示完全可编程的。黄色表示可配置，但不可编程。蓝色的阶段完全固定。

3.5 三大测试

参考 U3D Shader -> Shader 着色器一章。

https://blog.csdn.net/wangdingqiaoit/article/category/2107037

3.6 混合

3.7 背面剔除

3.8 渲染队列

https://blog.csdn.net/puppet_master/article/details/53900568

第四章　变换

见 OpenGL_4.5 笔记

第五章　视觉外观

5.1　光照与材质

5.1.1　光照现象-散射与吸收

5.1.2　表面粗糙度

5.2　着色原理

5.2.1　着色与着色方程

5.2.2　三种着色处理方法

5.3　抗锯齿与常见抗锯齿类型

抗锯齿（英语：Anti-Aliasing，简称 AA），也译为边缘柔化、消除混叠、抗图像折叠有损，反走样等。它是一种消除显示器输出的画面中图物边缘出现凹凸锯齿的技术，那些凹凸的锯齿通常因为高分辨率的信号以低分辨率表示或无法准确运算出 3D 图形坐标定位时所导致的图形混叠（aliasing）而产生的，抗锯齿技术能有效地解决这些问题。

下面将常见的几种抗锯齿类型进行总结介绍，也包括 RTR3 中没有讲到的，最近几年新提出的常见抗锯齿类型。

5.3.1 超级采样抗锯齿（SSAA）

5.3.2 多重采样抗锯齿（MSAA）

5.3.3 覆盖采样抗锯齿（CSAA）

5.3.4 高分辨率抗锯齿（HRAA）

5.3.5 可编程过滤抗锯齿（CFAA）

5.3.6 形态抗锯齿（MLAA）

5.3.7 快速近似抗锯齿（FXAA）

5.3.8 时间性抗锯齿（TXAA）

5.3.9 多帧采样抗锯齿（MFAA）

5.4 透明渲染与透明排序

5.4.1 透明渲染

5.4.2 透明排序

深度缓存（Z-Buffer）

画家算法（Painter's Algorithm）

加权平均值算法（Weighted Average）

深度剥离算法（Depth Peeling）

5.5 伽马矫正

5.6 参考文献

参考于：https://zhuanlan.zhihu.com/p/27234482

第六章　纹理贴图

6.1　纹理管线

纹理（Texturing）是一种针对物体表面属性进行“建模”的高效技术。图像纹理中的像素通常被称为纹素（Texels），区别于屏幕上的像素。根据 Kershaw 的术语，通过将投影方程（*projector function*）运用于空间中的点，从而得到一组称为参数空间值（*parameter-spacevalues*）的关于纹理的数值。这个过程就称为贴图（Mapping，也称映射），也就是**纹理贴图**（Texture Mapping，也称纹理映射）这个词的由来。

纹理贴图可以用一个通用的纹理管线来进行描述。纹理贴图过程的**初始点**是空间中的一个位置。这个位置**可以基于世界空间**，但是**更常见的是基于模型空间**。因为若此位置是基于模型空间的，当模型移动时，其纹理才会随之移动。

如下图为一个纹理管线（The Texturing Pipeline），也就是单个纹理应用纹理贴图的详细过程，而此管线有点复杂的原因是每一步均为用户提供了有效的控制。

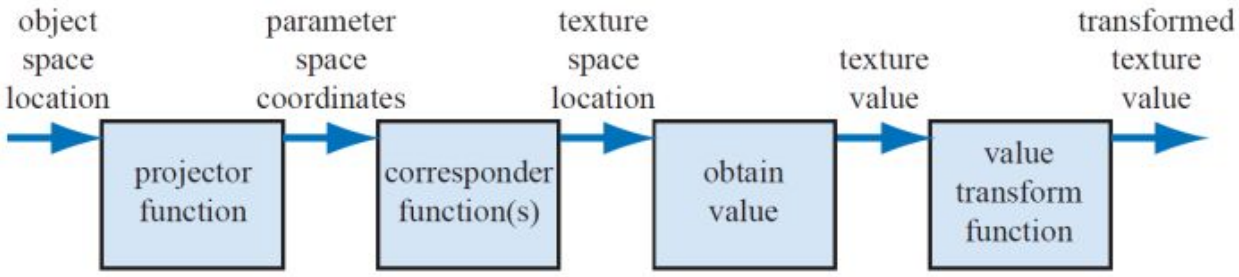

Fig 6.1: 单个纹理的通用纹理管线

下面是对上图中描述的纹理管线的分步概述：

1. 通过将投影方程（projector function）运用于空间中的点，从而得到一组称为参数空间值（parameter-space values）的**关于纹理的数值**。

2. 在使用这些新值访问纹理之前，可以使用一个或者多个映射函数（corresponder func-

tion）将**参数空间值**（parameter-space values ）转换到**纹理空间**。

3. 使用这些纹理空间值（texture-space locations）**从纹理中获取相应的值**（obtain value）。例如，可以使用图像纹理的数组索引来检索像素值。

4. 再使用值变换函数（value transform function）对检索结果进行值变换，最后使用得到的新值来改变表面属性，如材质或者着色法线等等。

而如下这个例子应该对理解纹理管线有所帮助。下例将描述出使用纹理管线，一个多边形在给定一张砖块纹理时在其表面上生成样本时（如下图）发生了哪些过程。

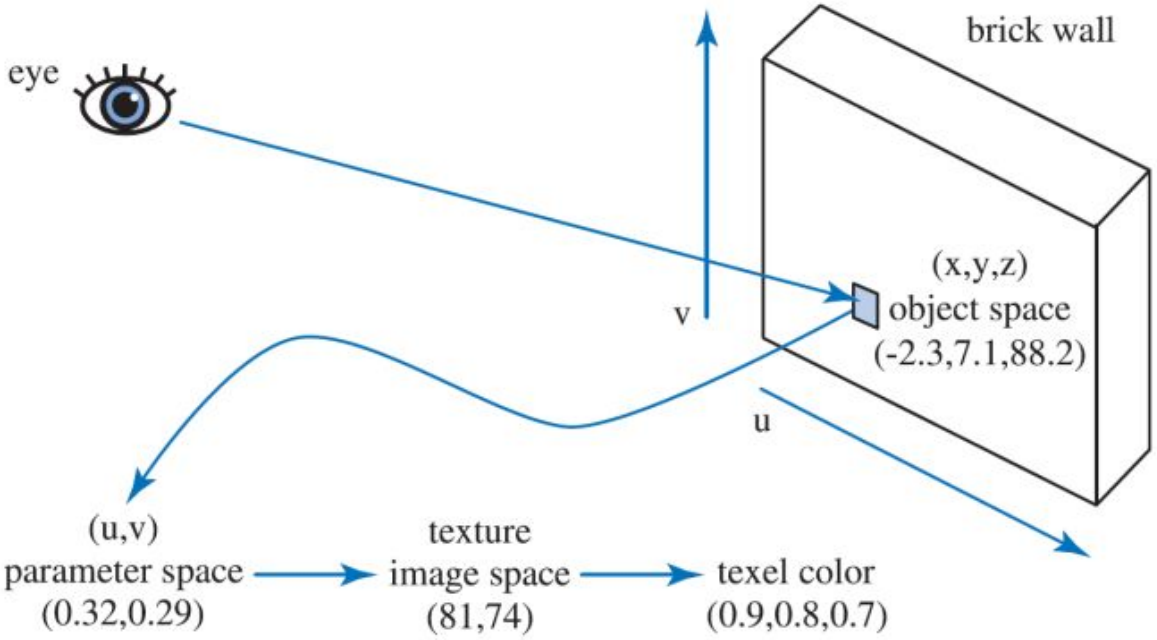

Fig 6.2: 一个砖墙的纹理管线过程

在具体的参考帧画面中找到**物体空间中的位置 (x,y,z)**，如图中点（-2.3,7.1,88.2），然后**对该位置运用投影函数**。**这个投影函数通常将向量 (x,y,z) 转换为一个二元向量 (u,v)**。在此示例中使用的投影函数是一个正交投影，类似一个投影仪，将具有光泽的砖墙图像投影到多边形表面上。再考虑砖墙这边，其实这个投影过程就是将砖墙平面上的点变换为值域为 0 到 1 之间的一对 (u,v) 值，如图，(0.32,0.29) 就是这个我们通过投影函数得到的 uv 值。**而我们图像 (纹理) 的分辨率是 256 x 256，所以，将 256 分别乘以 (0.32,0.29)，去掉小数点，得到纹理坐标 (81, 74)**。通过这个纹理坐标，可以**在纹理贴图上查找到坐标对应的颜色值**，所以，我们接着找到砖块图像上像素位置为 (81,74) 处的点，得到颜色 (0.9,0.8,0.7)。而由于原始砖墙的颜色太暗，因此可以使用一个值变换函数，给每个向量乘以 1.1，就可以得到我们纹理管线过程的结果——颜色值 (0.99,0.88,0.77)。

随后，我们就可以将此值用于着色方程，作为物体的漫反射颜色值，替换掉之前的漫反射颜色。

下面是纹理管线中主要的两个组成，投影函数（The Projector Function）和映射函数（The Corresponder Function）。

6.2 投影函数

作为**纹理管线的第一步**，投影函数的功能就是**将空间中的三维点转化为纹理坐标**，也就是**获取表面的位置并将其投影到参数空间中**。

在常规情况下，投影函数通常在美术建模阶段使用，并将投影结果存储于顶点数据中。也就是说，在软件开发过程中，我们一般不会去用投影函数去计算得到投影结果，**而是直接使用在美术建模过程中，已经存储在模型顶点数据中的投影结果**。

6.3 映射函数

映射函数（The Corresponder Function）的作用是将参数空间坐标（parameter-space coordinates）转换为纹理空间位置（texture space locations）。

我们知道图像会出现在物体表面的 (u,v) 位置上，且 uv 值的正常范围在 [0,1) 范围内。超出这个值域的纹理，其显示方式便可以由映射函数（The Corresponder Function）来决定。

在 OpenGL 中，这类映射函数称为“封装模式（Warapping Mode）”，在 Direct3D 中，这类函数叫做“寻址模式（Texture Addressing Mode）”。最常见的映射函数有以下几种：

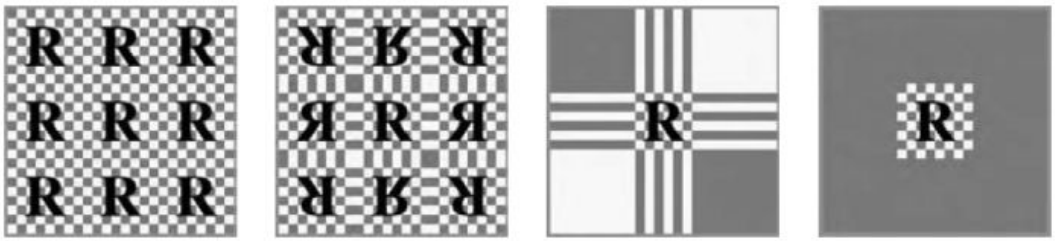

Fig 6.3: 映射方式

- **重复寻址模式**，`wrap` (DirectX), `repeat` (OpenGL)。图像在表面上重复出现。
- **镜像寻址模式**，`mirror`。图像在物体表面上不断重复，但每次重复时对图像进行镜像或者反转。

- **夹取寻址模式，**`clamp` (DirectX) ,clamp to edge (OpenGL)。夹取纹理寻址模式将纹理坐标夹取在 [0.0，1.0] 之间，也就是说，在 [0.0，1.0] 之间就是把纹理复制一遍，然后对于 [0.0，1.0] 之外的内容，将边缘的内容沿着 u 轴和 v 轴进行延伸。

- **边框颜色寻址模式，**`border` (DirectX) ,clamp to border (OpenGL)。边框颜色寻址模式就是在 [0.0，1.0] 之间绘制纹理，然后 [0.0，1.0] 之外的内容就用边框颜色填充。

另外，每个纹理轴可以使用不同的映射函数。例如在 u 轴使用重复寻址模式，在 v 轴使用夹取寻址模式。

6.4 体纹理

6.5 立方体贴图

立方体纹理（cube texture）或立方体贴图（cube map）是一种特殊的纹理技术，它用 6 幅二维纹理图像构成一个以原点为中心的纹理立方体，这每个 2D 纹理是一个立方体（cube）的一个面。对于每个片段，纹理坐标 (s, t, r) 被当作方向向量看待，每个纹素 (texel) 都表示从原点所看到的纹理立方体上的图像。

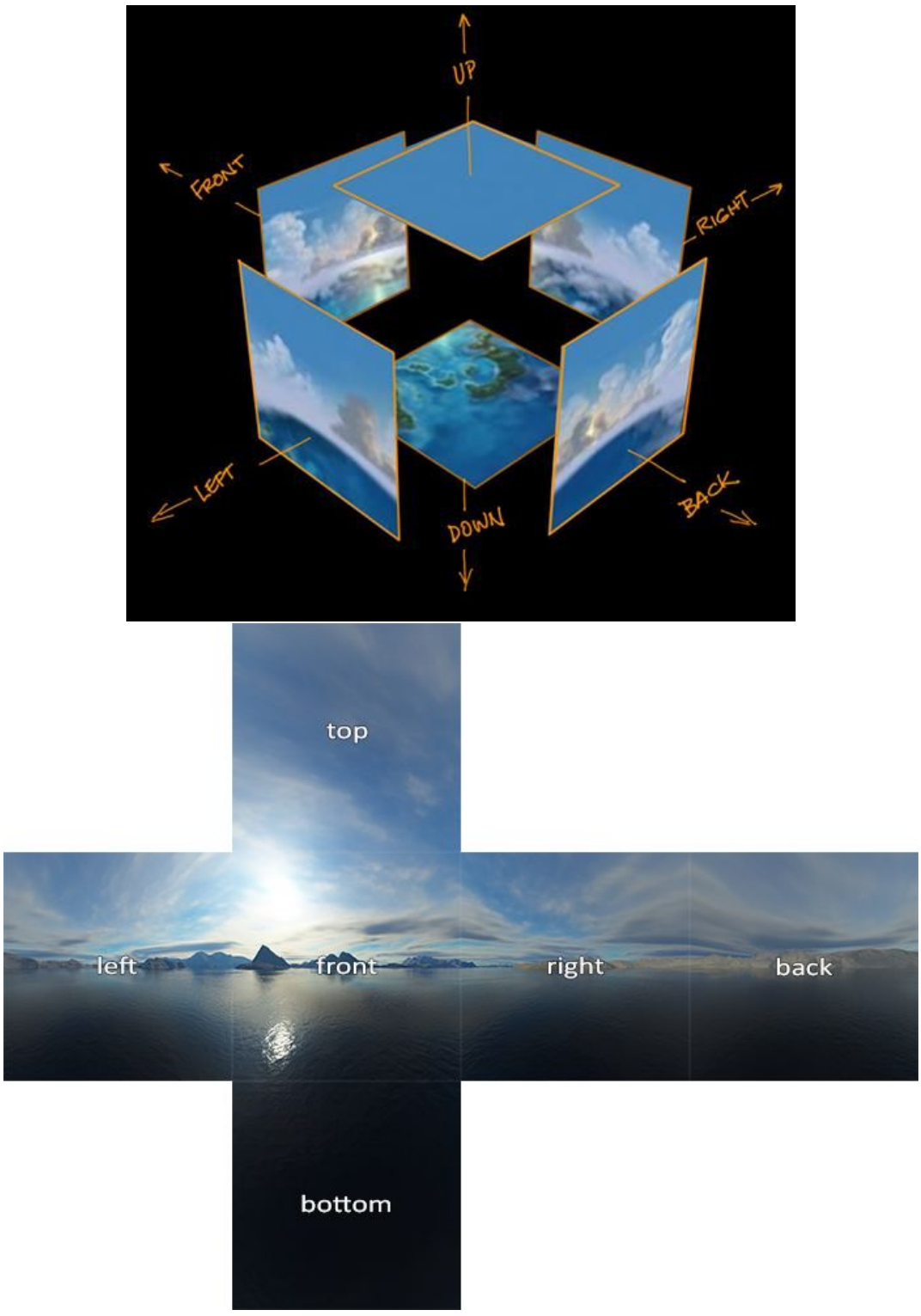

Fig 6.4: 立方体贴图

可以使用三分量纹理坐标向量来访问立方体贴图中的数据，该矢量指定了从立方体中心向外指向的光线的方向。选择具有最大绝对值的纹理坐标对应的相应的面。（例如：对于给定的矢量 $(-3.2, 5.1, -8.4)$，就选择-Z 面），而对剩下的两个坐标除以最大绝对值坐标的绝对值，即 8.4。那么就将剩下的两个坐标的范围转换到了-1 到 1，然后重映射到 [0，1] 范围，以方便纹理坐标的计算。例如，坐标（-3.2,5.1）映射到（(-3.2 / 8.4 + 1）/ 2，（5.1/ 8.4 + 1）/ 2） （0.31,0.80）。

立方体贴图支持**双线性滤波**以及 **mip mapping**，但问题可以可能会在贴图接缝处出现。有一些处理立方体贴图专业的工具在滤波时考虑到了可能的各种因素，如 ATI 公司的 CubeMapGen，采用来自其他面的相邻样本创建 **mipmap 链**，并考虑每个纹素的角度范围，可以得到比较不错的效果。

6.6 纹理缓存

6.7 纹理压缩

6.8 程序贴图纹理

6.9 凹凸贴图

凹凸贴图（Bump Mapping）思想最早是由图形学届大牛中的大牛 Jim Blinn 提出，后来的 Normal Mapping，Parallax Mapping，Parallax Occulision Mapping，Relief Mapping 等等，均是基于同样的思想，只是考虑得越来越全面，效果也越来越逼真。

贴图方式	思想概述	提出年代
Bump mapping 凹凸贴图	计算 vertex 的光强时，不是直接使用该 vertex 的原始法向量，而是在原始法向量上加上一个扰动得到修改法向量，经过光强计算，能够产生凹凸不平的表面效果。No self-occlusion, No self-shadow, No silhouette。	1978
Displacement Mapping 移位贴图	直接作用于 vertex，根据 displacement map 中相对应 vertex 的像素值，使 vertex 沿法向移动，产生真正的凹凸表面。	1984
Normal Mapping 法线贴图	normal map 需要法向量的信息，而法向量信息可由 height map 得到，且 texture 的 RGB 可以表示法向量的 XYZ，利用此信息计算光强，产生凹凸阴影的效果。No self-occlusion, No self-shadow, No silhouette。	1996
Parallax Mapping (Virtual Displacement Mapping) 视差贴图	没有修改 vertex 的位置，以视线和 height map 计算较陡峭的视角给 vertex 较多的位移，较平缓的视角给 vertex 较少的位移，透过视差获得更强的立体感，即利用 HeightMap 进行了近似的 Texture Offset。No self-occlusion, No self-shadow。	2001
Relief Mapping (Steep Parallax Mapping) 浮雕贴图	更精确地找出观察者的视线与高度的交点，对应的 texture 坐标则是位移的距离，所以能更正确地模拟立体的效果。Relief Mapping 实现了精确的 Texture Offset。Relief Mapping 可以产生 self-occlusion, self-shadowing, view-motion parallax, and silhouettes。	2005

Fig 6.5: 凹凸贴图对比

除了 Displacement Mapping 方法以外，其他的几种改进一般都是通过修改每像素着色方程来实现，关键思想是访问纹理来修改表面的法线，而不是改变光照方程中的颜色分量。物体表面的几何法线保持不变，我们修改的只是照明方程中使用的法线值。他们比单独的纹理有更好的三维感官，但是显然还是比不上实际的三维几何体。

6.9.1 移位贴图

6.9.2 法线贴图

高度图或灰度图 一张二维纹理有两个维度 u 和 v，但其实，高度 (h) 可以算第三个维度。有了高度，一张二维纹理就可以想象成一个三维的物体了。

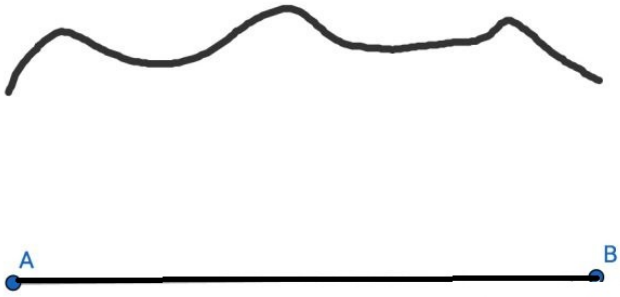

Fig 6.6: UV 例子

先来考虑只有 u 方向的情况，如图所示，A 和 B 是纹理中的两个点，uv 坐标分别是（$0, 0$）和（$1, 0$），上方黑线表示点对应的高度，那么显然，只要求出 u 方向上的高度函数在某一点的切线，就能求出垂直于他的法线了。同理，v 方向也是如此。也就是说，如果有纹理的高度信息，那么就能计算出纹理中每一个像素的法线了。

所以计算法线需要一张高度图，它表示纹理中每一个点对应的高度。

但其实并不需要求出每个纹理像素上 uv 方向各自的法线，只需要求出 uv 方向上高度函数的切线，再做一个叉积，即可计算出对应的法线了。

如果没有高度图，也可以用灰度图代替，灰度图就是把 rgb 三个颜色分量做一个加权平均，有很多种算法提取灰度值，这里用一个比较常用的基于人眼感知的灰度值提取公式。

$$color.r * 0.2126 + color.g * 0.7152 + color.b * 0.0722$$

这个公式是由人眼对不同颜色敏感度不同得来的，这里无需过多计较，直接把提取出来的灰度值作为高度值即可。

计算方法

实例

6.9.3 视差贴图

6.9.4 浮雕贴图

6.10 参考

参考于：https://zhuanlan.zhihu.com/p/27551369

法线贴图：https://www.zhihu.com/people/zheng-hua-31-46/following

第七章　光照

第八章　基于图像的效果

第九章　数学基础

参考:https://blog.csdn.net/wangdingqiaoit/article/details/51383052

9.1　向量

9.2　矩阵

9.3　GLM 库