目录

GLSL 常用语句总结
GLSL built-in 参数总结
GLM 常用函数总结

Getting Started

2. Introduction of OpenGL

2.1 概念集锦

immediate mode (fixed function pipeline): 用户无法自行更改 OpenGL 的操作过程

core-profile mode：用户参与编辑 OpenGL 运作机制

•OpenGL3.3 版本以后成为 modern OpenGL，运行机制与从前有变化，之后变化不大

•OpenGL 的版本与计算机使用的显卡相关，越新的显卡才能支持越新的 OpenGL 版本

•OpenGL 是由一个 specification 和一个 library 构成的

extensions：扩展；一般在显卡驱动程序中附带的 OpenGL 的新功能或算法，随着显卡或其驱动更新而发布

state machine：状态机，OpenGL 及其函数库按照状态机机制工作，即通过设定一些变量的值，改变 OpenGL 的工作方式（context）；比如设定其绘制直线或三角形

object：对象；即 C++中类和对象的定义；OpenGL 中会使用很多对象，如缓存 buffer，使用方法是绑定 bind，绑定一个对象后，该对象处于激活状态，后续相关语句可以隐式调用该对象对其操作，直到解绑定为止；有时同类对象只能绑定一个，绑定其他对象前要解绑；这种使用方式的好处是，可以将一些设置 configuration 保存在对象中，需要使用时绑定即可，不必每次改变设置都要进行大量语句操作

API: Application program interface (API) is a set of routines, protocols, and tools for building software applications. An API specifies how software components should interact. Additionally, APIs are used when programming graphical user interface (GUI) components.

2.2 OpenGL 数据类型

命令后缀	数据类型	典型的 C/C++类型	OpenGL 类型名
b	8 位整型	signed char	GLbyte

s	16 位整型	short	GLshort
i	32 位整型	int/long	Glint, GLsizei
f	32 位浮点型	float	GLfloat, GLclampf
d	64 位浮点型	double	GLdouble, GLclampd
ub	8 位无符号整型	unsigned char	GLubyte, GLboolean
us	16 位无符号整型	unsigned short	GLushort
ui	32 位无符号整型	unsigned int/long	GLuint,GLenum,GLbitfield

3. Creating a window

OpenGL 本身不具备创建窗口的功能，因为创建窗口和操作系统相关。有很多现成的 library 可以执行这个工作，比如 GLUT，SDL，SFML 和 GLFW

5.Hello Triangle

•着色器（shader）是一些在 GPU 上运行的处理图形管线步骤的小程序
•着色器语言是用 GLSL（OpenGL Shading Language）编写的
图形管线（graphics pipeline）的按着色器（shader）分类的描述：
vertex data
→vertex shader 顶点着色器（modern OpenGL 必须自定义的）
NDC（normalized device coordinates）：顶点经过顶点着色器后的坐标，在-1 到 1 之间
GLSL（OpenGL Shading Language）：着色器的编写语言（类似 C 语言的一种语言），另行编译后方可使用
→shape assembly 形状组合
→geometry shader（可选，可默认）
→tesselation shader 镶嵌（可选，可默认）
→rasterization 光栅化
→fragment shader 片段着色器（最主要的高阶效果着色器，modern OpenGL 必须自定义的）
→alpha tests and blending 透明度测试和混合

6.Shaders

6.1 GLSL(p52)
常用函数整理：
①数据构造
vec*(…) //向量构造函数，*号代表维数，括号内为每个维度的值（也可以用其他向量和数值拼凑或者只用一个值表示所有维度取同值），返回向量
例：vec4(1,2,3,4); vector3=vec3(1); vec4(vector3,1)
mat*(…) //方矩阵构造函数，*代表维数（3、4），括号内可以为其他维数向量或矩阵（可以是更高维，则为矩阵裁剪）
例：mat3(vector3,vector3,vector3); matrix4=mat4();mat3(matrix4)
②计算函数
normalize(X) //归一化向量，返回结果
length(X) //求 vec3 向量长度，返回结果
inverse(X) //矩阵求逆，返回结果

transpose(X) //矩阵转置，返回结果

pow(X,Y) //乘幂计算，X 为底数，Y 为指数，X 和 Y 可以是同维向量，返回每个分量分别乘幂的结果向量

dot(X,Y) //vec3 向量点乘，返回结果；一般的“*”计算用于向量表分量相乘，结果仍为向量

cross(X,Y)//vec3 向量叉乘，返回结果

clamp(X,Y,Z) //限定取值范围，X 为被操作变量或表达式，Y 为下限，Z 为上限值，返回结果为 X（若 X 处于 YZ 之间）或 Y（X<Y）或 Z（X>Z）

reflect(X,Y) //计算光线反射过程，X 为入射方向（光源到物体），Y 为法向，返回出射向量

refract(X,Y,Z) //计算光线折射过程，X 为入射方向（光源到物体），Y 为法向，Z 为折射率比值（入射介质/出射介质），返回出射向量

③颜色纹理处理

mix(X,Y,Z) //X,Y 是颜色，Z 是混合比例，返回颜色

texture(X,Y) //X 是 sampler2D 纹理，Y 是纹理坐标，返回 vec4 颜色

texelFetch(X, Y, Z) //X 是 sampler2DMS 纹理，Y 纹理坐标，Z 为 subsample 索引（从 0 开始的整数），返回 vec4 颜色

textureSize(X,Y) //X 是纹理，Y 是 mipmap level（从 0 开始），返回一个 vec2 代表 X 的宽度和高度尺寸（即 texel 的个数）

discard //放弃当前操作片段，用在片段着色器中

6.2 Types

int, float, double, uint, bool (C like language)

vectors, matrices

6.2.1 Vectors

- vecn: the default vector of n floats.
- bvecn: a vector of n booleans.
- ivecn: a vector of n signed integers.
- uvecn: a vector of n unsigned integers.
- dvecn: a vector of n double components

vec.x=vec.r=vec.s

vec.y=vec.g=vec.t

vec.z=vec.b=vec.p

vec.w=vec.a=vec.q

•Special Feature: **swizzling**

如果 Vec 是 vec4 类型的向量，Vec.xy 即为 vec2 类型的向量，由 Vec 的 x 和 y 坐标分量组成，xy（或其他字母 rg，st）可以是其他任意组合或任意数量的组合（≤4）

6.3 Ins and Outs

•顶点着色器需要设定顶点的特性（attribute）的对应接口，即 layout (location=?)，方便用户传递数值；OpenGL 在用户不设定时也会自动设定 location，用户需要利用 gl 函数来询问该 location

•顶点着色器的最终结果，即可见顶点的坐标必须在-1 到 1 之间（投影变换的矩阵包括将坐标转换到-1 至 1 的空间的操作），这个坐标即为 clip-space 顶点坐标或称 NDC（用于裁剪看不见的图形，使用这样的坐标更加方便，变换后的形式也叫正则视景体）；注意，透视除法是由 OpenGL 自动完成的，因此严格来说为 gl_Position 赋值时只需要保证透视除法之后所有坐标（前 3 个）在-1 到 1 范围内即可。

•裁剪之后，顶点坐标（包括深度）又被转换到[0,1]区间，是为了方便进行深度测试和后续

片段操作，片段坐标将会和屏幕坐标衔接，因此后续步骤中负值坐标的存在也没有了意义
•片段着色器必须至少有一个 vec4 或 vec3 的颜色输出变量（如果没有，整个着色器将为空程序）。可以同时向多个缓冲输出颜色（或存于其中的数据），使用 layout (location=?)来区分，在应用程序中要为当前的 framebuffer 针对性的配置多个 colorbuffer。
•顶点着色器后接片段着色器，每个前后连接的着色器输出输入的数据必须对应（类型名称相同）
•一般情况下，顶点着色器的输出 out 不仅仅是针对每个顶点，其他的像素（或片段）的对应输出值采用插值获得（光栅化过程），这样传送给片段着色器的才是对应所有片段的数据

6.4 Uniforms

•着色器内部的全局变量，可以在 game loop 中改变值来实现对着色器调控
uniform vec4 ourcolor; // 关键词+数据类型+名称
•提取位置
glGetUniformLocation(X,"Y")//获取全局变量 Y 的 location，X 是 shader program
•传送数值方法（？表示数据类型缩写 f,i,ui 等）
1）单值
glUniform1?(X,Y)
2）向量（如果是 glm 的向量类必须逐值传送）
glUniform3? (X,Y1,Y2,Y3)
glUniform4? (X,Y1,Y2,Y3,Y4)
或
glUniform4?v(X,Y,Z) //第一位置，第二元素数量，第三被传送数值的指针或数组名
3）矩阵（行或列的维数范围 2 到 4）
glUniformMatrix3fv(X,Y,GL_FALSE,glm::value_ptr(Z))
glUniformMatrix4fv(X,Y,GL_FALSE,glm::value_ptr(Z))
glUniformMatrix3x2fv(X,Y,GL_TRUE,glm::value_ptr(Z))
//第一 uniform 位置，第二元素个数（如果被操作 uniform 是个向量或矩阵数组，可以通过此方法一起传值，也可以设为 1 单独操作），第三是否需要转置，第四被传送矩阵指针（使用 glm 的函数来获得，ptr 即指针）
4）sampler2D
glUniform1i(X,Y) //第一 uniform 位置，第二 texture unit 编号，必须为 int
5）结构体和数组
方法同上，只能一个数据成员一个数据成员赋值，如果是非向量和矩阵类型的数组，也只能一个成员一个成员赋值

6.5 Our own shader class

6.6 Reading from file

7.Textures

•texture coordinate 在 x 和 y 方向上的范围都是 0 到 1
•用纹理坐标提取纹理颜色的过程叫做取样 sampling

7.1 Texture Wrapping

纹理包裹的样式选择：
GL_REPEAT：平铺重复
GL_MIRRORED_REPEAT：镜像平铺重复，相邻的图像成镜像
GL_CLAMP_TO_EDGE：拉伸，超过边缘的坐标仍按照边界坐标处理

GL_CLAMP_TO_BORDER：超过边缘的坐标给定一个自定义颜色
•对于每个纹理坐标（s，t，r）都可以设置一种方式
例 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_MIRRORED_REPEAT);
对于自定义边界颜色，要另行用语句设定
float borderColor[]={1.0f,1.0f,0.0f,1.0f};
glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_BORDER_COLOR,borderColor);

7.2 Texture Filtering

texture filtering 将纹理坐标（可以是任意值）和纹理像素（texture pixel 或 texel）对应的过程方式选择：
GL_NEAREST（nearest neighbor filtering，default）：
将中心到纹理坐标最近的像素颜色与该坐标绑定
GL_LINEAR（(bi)linear filtering）：
纹理坐标的对应颜色由附近几个像素颜色插值决定，中心到坐标越近的像素贡献越大
•特点：（尤其是低分辨率纹理情况下）
NEAREST 不影响清晰度，但是有锯齿（blocked patterns）
LINEAR 弱化了锯齿，但是图像更加模糊不清
•可以针对放大和缩小操作分别选择对应的纹理过滤方式
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

7.2.1 Mipmaps

一系列相同内容不同尺寸（即缩放）的纹理集合，每个纹理比前一个纹理缩小 1/2，用于给 OpenGL 根据物体距离观察者远近来决定采用哪一个放缩程度的纹理，以此来减小计算开销
•Mipmaps 的使用实际上是一种 texture filtering 方式，可以用来代替前面的简单方式
•Mipmaps 只能使用在 minifying 的情况下，原因不言自明
可选方式：
GL_NEAREST_MIPMAP_NEAREST
GL_LINEAR_MIPMAP_NEAREST
GL_NEAREST_MIPMAP_LINEAR
GL_LINEAR_MIPMAP_LINEAR
•Mipmap 前面的参数指定的是采样方式（同前），后面的参数是指定 mipmap 的方式，nearest 表示单独选取最适合当前尺寸的 mipmap，linear 则是选取两个最近的插值

7.3 Loading and creating textures

将图片格式，比如 png，转换成比特数组格式需要一个图像装载器（image loader），这是一个需要编写的小程序

7.4 Simple OpenGL Image Library(SOIL)

7.5 Generating a texture

7.6 Applying textures

7.7 Texture Units

OpenGL 一般有 16 个纹理单元，编号 GL_TEXTURE0 到 GL_TEXTURE15；除此，编号可以用表达式比如 GL_TEXTURE0+8=GL_TEXTURE8

8.Transformations

8.1-8.17 Basic Knowledge

8.18 OpenGL Mathematics (GLM)

•GLM 中不同版本的函数参数有的使用角度，有的使用弧度

返回操作结果矩阵：

glm::translate(X,glm::vec3(…));　//X 表示被操作矩阵 mat4，后面向量表示平移向量

glm::rotate(X,Y,glm::vec3(…));　//X 同上，Y 表示旋转角度，后面向量表示转轴

glm::scale(X,glm::vec3(…)); //X 同上，后面向量表示缩放系数 Sx,Sy,Sz

glm::lookAt(X,Y,Z); //创建视角变换矩阵 see 10.2

glm::value_ptr(X);　//X 为 glm::mat4 类型，转换结果为 GLSL 的 mat4 类型

glm::mat4(); //返回单位矩阵

其他计算函数：

glm::normalize(X); //返回结果，vec3 向量归一化

glm::cross(X,Y); //返回结果，vec3 向量叉乘，X×Y

glm::radians(X); //返回弧度，X 为角度

9.Coordinate Systems

•Local space (or Object space)

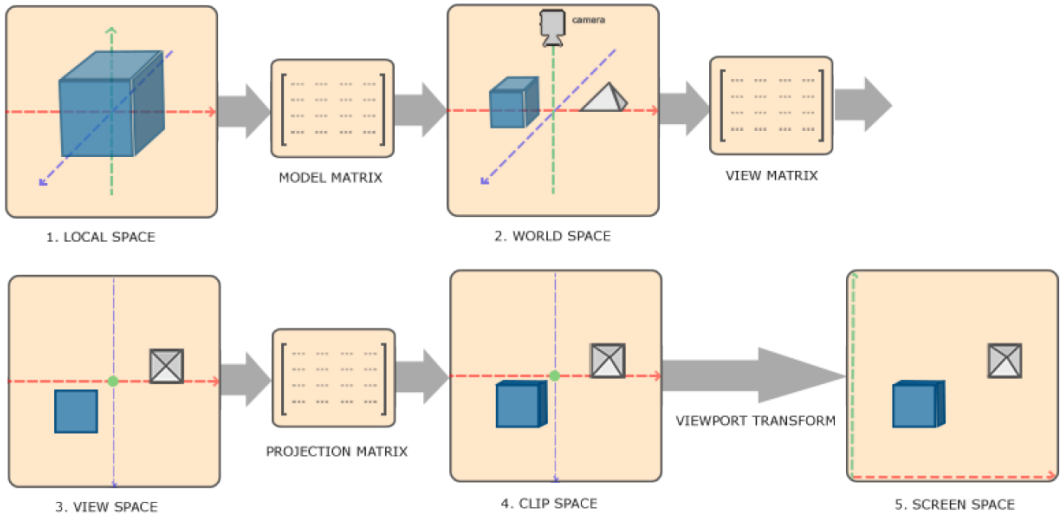
•World space

•View space (or Eye space)

•Clip space

•Screen space

9.1 The global picture



9.2 Local space

9.3 World space

9.4 View space

or camera space or eye space

9.5 Clip space

viewing box = frustum

•perspective division　在顶点着色器的最后被自动执行

9.5.1 Orthographic projection

•Feature:

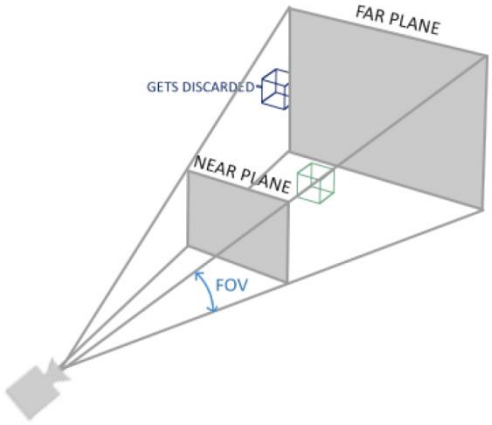
- cube-like frustum

- width,height,near,far plane

•create matrix

return mat4 - glm::ortho(float left,right,bottom,top,near,far)

9.5.2 Perspective projection



•create matrix

return mat4 - glm::perspective(fov, aspect ratio,near,far)

fov:field of view（角度单位） 一般真实情况设置为 45.0f

aspect ratio: width/height

near and far:一般设为 0.1f 和 100.0f

•application：2D 作图，或者其他一些工程或建筑作图，为了保持精确的尺寸，经常采用 orthographic projection，真实视觉模拟的情况下，采用 perspective projection

9.6 Putting it all together

V_clip=M_projection*M_view*M_model*V_local

9.7 Going 3D

•OpenGL 的世界坐标采用右手系，DirectX 采用左手系，但是对于 NDC，OpenGL 采用左手系，因为投影矩阵改变了 z 坐标符号

9.8 More 3D

9.8.1 Z-buffer

9.8.2 More cubes!

10.Camera

10.1 Camera/View space

10.1.1 Camera position

glm::vec3 cameraPos = glm::vec3(0.0f,0.0f,3.0f);

10.1.2 Camera direction

glm::vec3 cameraTarget = glm::vec3(0.0f,0.0f,0.0f);

glm::vec3 cameraDirection = glm::normalize(cameraPos - cameraTarget);

•用来描述相机位向的向量实际上是与相机的朝向相反的

10.1.3 Right axis

glm::vec3 up = glm::vec3(0.0f,1.0f,0.0f);

glm::vec3 cameraRight = glm::normalize(glm::cross(up,cameraDirection));

•这就是 OpenGL 计算相机侧向的方法，因为多数情况下无法人为指定相机顶端朝向

•一般默认将 up 选取+y 方向，因为一般的相机不会绕视轴转动

10.1.4 Up axis（不是前面说的 up）

glm::vec3 cameraUp = glm::cross(cameraDirection,cameraRight);

10.2 Look At

$$LookAt = \begin{pmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ D_x & D_y & D_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} \boldsymbol{R} & \boldsymbol{U} & \boldsymbol{D} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^T \begin{pmatrix} 1 & \boldsymbol{P} \\ 0 & 1 \end{pmatrix}^{-1}$$

$$= \begin{pmatrix} \boldsymbol{R} & \boldsymbol{U} & \boldsymbol{D} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 & \boldsymbol{P} \\ 0 & 1 \end{pmatrix}^{-1} = \left(\begin{pmatrix} 1 & \boldsymbol{P} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \boldsymbol{R} & \boldsymbol{U} & \boldsymbol{D} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right)^{-1} = \begin{pmatrix} R_x & U_x & D_x & P_x \\ R_y & U_y & D_y & P_y \\ R_z & U_z & D_z & P_z \\ 0 & 0 & 0 & 1 \end{pmatrix}^{-1}$$

•坐标系的变换矩阵是坐标的变换矩阵的逆

glm::lookAt(glm::vec3 position, glm::vec3 target, glm::vec3 up);

10.3 Walk around

•一般的 key_callback 函数一次调用（一个循环）只能应对一种按键情况，而且是激活按键（按下或抬起不包括按住）的情况下才会调用

10.4 Movement speed

•图形是一帧一帧绘制的，而每一台电脑的配置和设置可能导致每秒帧数（FPS）不同，因此和运动（movement）有关的操作不能只和 FPS 对接，应该在每一帧记录两帧时间差（deltaTime），以此将运动和真正的时间参数对接

•方法就是调用 glfwGetTime()，获得当前机器时间

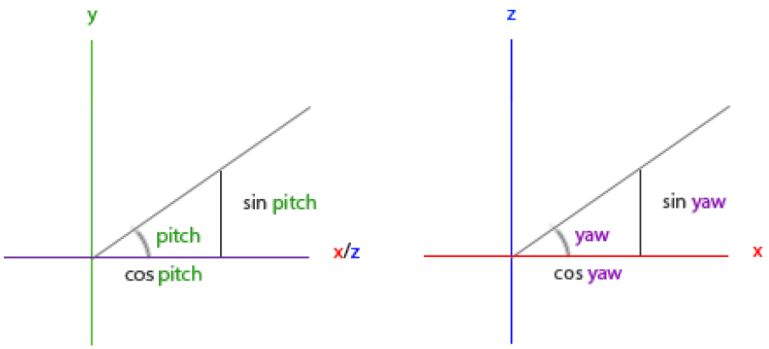
10.5 Look around

10.6 Euler angles

pitch,yaw,roll

•一般只需要 pitch 和 yaw，即飞机的点头和摇头的两个转轴

•欧拉角的系统不完美，存在 Gimbal lock，最好的是用 *quaternions*，不过如果不需要进行 roll 运动，或者 FPS 类型的游戏相机，这个系统够用了



direction.x = cos(glm::radians(pitch)) * cos(glm::radians(yaw));

direction.y = sin(glm::radians(pitch));

direction.z = cos(glm::radians(pitch)) * sin(glm::radians(yaw));

10.7 Mouse input

•鼠标的移动方式永远是屏幕上 2 维平面运动，相机的旋转运动实则是和鼠标的 2 维运动进行转换连接（相机不考虑 roll）

•因为上述原因，glfm::lookAt 函数中的目标 target，对于相机的位向实际上不是直观描述的，

因此在程序中使用 position+direction 的方式描述 target，而 direction 用以描述相机的旋转位向

10.8 Zoom

•通过鼠标滚轮的运动和 fov 的关联实现

10.9 Camera class

•将所有和相机操作有关的函数封装

•对于互动性的 callback 函数使用方法：只利用该函数进行操作设备参数的提取，而将把参数转换为实际操作的过程封装成函数

11.Review

Lighting

12. Colors

12.1 A lighting scene

•物体和光源要用不同的着色器

•一套顶点适合描述一个物体，不过一般顶点的局部坐标都以原点为中心，方便旋转和缩放；然后通过不同的模型矩阵来进行变换位置大小等

13. Basic Lighting

•Phong lighting model: ambient, diffuse, specular

13.1 Ambient lighitng

•最全面的考察场景中物体反射光对其他物体的光照的算法称为 global illumination，但是简化版的称为 ambient lighting

13.2 Diffuse lighting

13.3 Normal vectors

•法向量可以在着色器中计算，也可以手动直接导入着色器

13.4 Calculating the diffuse color

※计算漫反射时（根据使用模型），光照方向往往选取朝向光源的向量

13.5 One last thing

•法向量的模型变换为$(M^{-1})^T$，变换时，给法向量增加一个为 0 的第四坐标，和将矩阵裁剪为左上角 3*3 矩阵效果一样；法向量的模型视点变换都是$(M^{-1})^T$，因为都是仿射变换坐标系，但是透视变换是和顶点一样的，因为只是视觉效果上的变换，结果不需要保证垂直关系

•截至目前的例子中，光照计算使用的坐标是世界坐标，即在顶点着色器使用视点变换前的坐标（即便使用了视点变换，对于法向量再增加对应的变换即可）

•※逆矩阵计算非常耗费资源，见“计算图形学算法——优化习惯与技巧”

13.6 Specular Lighting

※计算镜面反射向量时，比如使用 reflect(X,Y)，光照方向朝向物体

•计算镜面反射光照最好使用视点坐标系（投影变换前），因为在这里，视点位置永远为 0，无需计算

•计算方程中指数为 shininess，越大反射斑越小，即镜面反射效果越强

※在顶点着色器中计算光照颜色，那么除了顶点外的像素颜色采用插值计算，是为 Gouraud 模型；在片段着色器中计算光照颜色，被插值的是法向量和片段位置，是为 Phong 模

型。显然，Phong 模型更适合计算光照颜色剧烈变化的情况

•如果使用 Gouraud 模型，有时会出现条带，是因为插值形成的，即片段的颜色由插值获得而不是通过计算直接获得（插值不代表绝对正确，有时根本不正确）

14. Materials

14.1 Setting materials

•将材质参数集合成一个全局结构体（GLSL 类似 C，估计没有 class）

•一般环境光和漫反射材质系数设为同值，也就是材质的颜色；如果有些第三方参考数据中环境光和漫反射材质系数不同，则有可能将光分量属性考虑进材质系数中，需要将所有光分量都设为最强白光才能获得正确结果（也可以看环境光材质系数大小来决定）

14.2 Light properties

•一般将环境光分量设为昏暗白光，漫反射光分量设为光本身颜色，镜面反射分量一般设为最亮白光（不过仍需注意实际中镜面反射光的颜色一般由光源颜色决定）

•同样将光源分量集合成为一个全局结构体，放置在 fragment shader 中，着色部分一般都在 fragment shader 中，vertex shader 多用于几何变换和几何数据获取

14.3 Different light colors

15. Lighting maps

•一个物体由不同材质的部件组成，不同部件的光照效果不同，需要区别对待，这也就是 map 名称的由来，map 实质上是 texture 的一种，其存在使得对于每个 fragment，我们都可以赋予不同的材质属性 diffuse map and specular map、法向属性 normal/bump maps and reflection map、辉光属性 emission map

15.1 Diffuse maps

•GLSL 中的 sampler2D 是一个 opaque type，只能用 uniform 方式使用，包含这个类型的结构体等也是一样

15.2 Specular maps

•specular maps 是可以通过画图软件加工 diffuse map 获得的，只要将不反光的部分去除，并填充以黑或灰色或降低亮度，反光部分增加亮度与对比度等

15.3 Sampling specular maps

16. Light casters

16.1 Directional Light

•方向光源：太阳，远距离光源等，没有衰减

16.2 Point lights

•一般实际应用中的点光源都随着距离的增加而光强减小

16.3 Attenuation（衰减）

$$\text{atten} = \frac{1}{k_c + k_l D + k_q D^2}$$

•一般 kc 取 1，只是为了保证结果分母（denominator）不小于 1，不会导致反向增加

16.3.1 Choosing the right values

•常用 k 值组合表见下，distance 表示光源彻底衰减为零的距离

Distance	Constant	Linear	Quadratic
7	1.0	0.7	1.8
13	1.0	0.35	0.44

20	1.0	0.22	0.20
32	1.0	0.14	0.07
50	1.0	0.09	0.032
65	1.0	0.07	0.017
100	1.0	0.045	0.0075
160	1.0	0.027	0.0028
200	1.0	0.022	0.0019
325	1.0	0.014	0.0007
600	1.0	0.007	0.0002
3250	1.0	0.0014	0.000007

16.3.2 Implementing attenuation

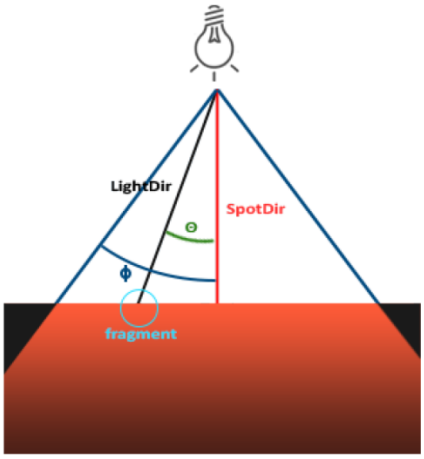
16.4 Spotlight

•聚光灯：路灯，手电灯，舞台灯，探照灯等

•四个变量描述：LightDir（片段到光源向量），SpotDir（聚光灯中心方向），Phi（聚光灯范围半角，cufoff angle），Theta（LightDir 和 SpotDir 夹角）

•如果在范围内，则正常计算光照，否则不计算漫反射和镜面反射，环境光分量应该保留，避免完全黑暗（仍然是真实性试验的结果，根据情况自己决定的）

•检测片段是否位于光锥内部时，可以直接使用和比较 phi 和 theta 的 cos 值，避免计算反三角函数，降低复杂度



16.5 Flashlight

•手电的特点是随着相机移动的，包括矿工头灯，因此其参数可以直接利用相机类的数据，camera.Position; camera.Front

16.6 Smooth/Soft edges

$$I = \frac{\theta - \gamma}{\varepsilon}$$

•theta 同上，gama 为外围光锥半角，epsilon 为内外光锥半角的差值，所有的角实际上都是 cos 值

•为了将光斑边缘柔化，引入一个 intensity 因子，实际上是对 0 到 1 的一个插值，将光的强度慢慢变化

17. Multiple lights

•本章的意义在于把光源光照计算封装成函数，实现简洁的系统的多元光照过程

•相机封装为类和头文件，着色器和编译封装成类和头文件，光照计算封装为函数在着色器内部

17.1 Directional light

17.2 Point light

17.3 Putting it all together

Model Loading

19. Assimp

•3D Modeling tools: Blender, 3DS Max, Maya

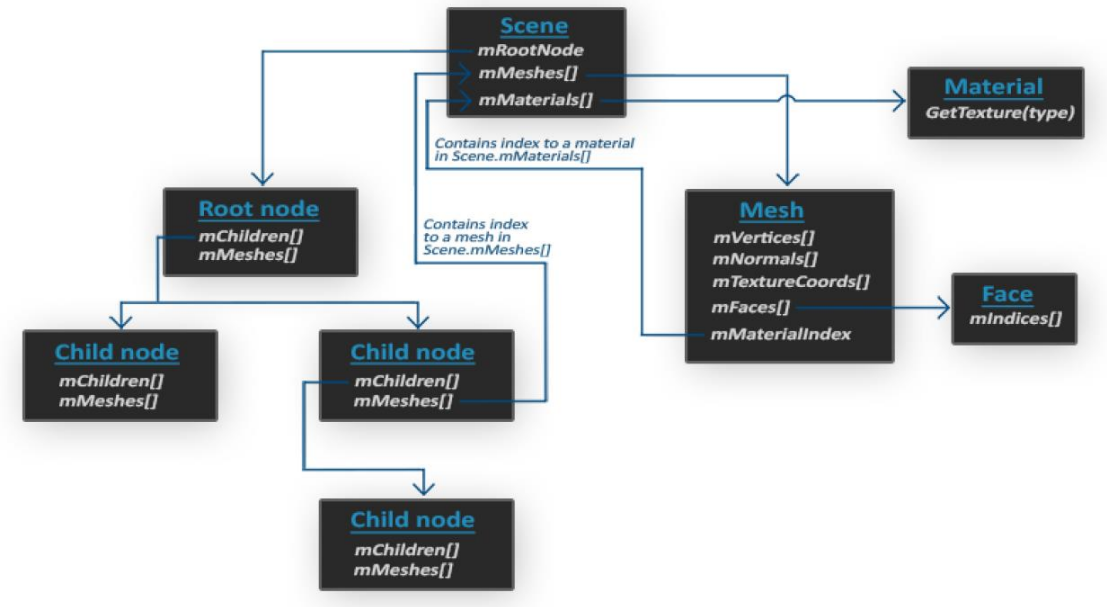
•艺术家，模型师使用 3D 模型工具制作图形和纹理（纹理制作有时用二维绘图工具，如 PS 等），附着纹理的中间过程为 uv-mapping；而图形程序员（graphics programmers）是利用生成的模型文件（包含从顶点到纹理甚至材质光照动画等信息）进行实时图形编程

19.1 A model loading library

•Assimp: Open Asset Import Library; 一个工具，可以导入导出各种不同的模型文件格式，在工具内部，将不同格式文件存储成通用数据结构，因此可以代码兼容多种不同格式

•Mesh: 可以理解为组成 model 的因子，而 mesh 是最基本的绘图需求信息的集合（顶点材质等），一个 mesh 可以绘制出一个形状（不是图元，图元比 mesh 还基本），多个 mesh 组成一个完整模型 model，多个 model 组成一个场景 scene

•Assimp 的结构：



最初的 scene 对象创建函数返回的是 scene 类的指针

Scene->mMesh[]中包含的是 aiMesh 类的指针

Scene->mRootNode 和 mRootNode->mChildren[]保存的都是 aiNode 类指针

Scene->mMaterials[]保存的是 aiMaterial 类指针

Mesh->mFaces[]中包含的是 aiFace 类对象，其他的都是实体对象或数值

19.2 Building Assimp

20. Mesh

•一个 Mesh 类最基本的需求：顶点，法向，纹理坐标和材质纹理，顶点索引等

20.1 Initialization

20.2 Rendering

•通过规则命名法来解决预先不知道有多少和几种纹理的问题，命名虽然是字符串 string 类型，但是仍然可以当作一般数据比较和操作，不过 string 和数字类型的转换根据编译器版本不同有不同繁琐程度的方法

21.Model

•node：3D 模型制作中一种用于描述结构关系和设计路线的系统；每个 node 里面包含若干个 mesh 和若干个子 node（如果有子 node 的话）；可以按照等级来理解 node：每个 node 管理几个 mesh，不同的 node 之间有等级差别不过管理的 mesh 不重合；举例来说，汽车的 model 包含引擎、方向盘、轮胎等 mesh，这些 mesh 通过一系列 node 来进行分类管理，低级 node 的 mesh 一般依附于高级 node 的 mesh；如果想要移动汽车 model，那么其附属的所有的 mesh 都可以移动；如果只是想要移动座椅位置，那么除了座椅和和座椅密切相关的配件以外，其他的部件可以不受影响。

21.1 Importing a 3D model into OpenGL

21.1.1 Assimp to Mesh+Vertices

21.1.2 Indices

21.1.3 Material

21.2 A large optimization

•优化，避免重复加载纹理

21.3 No more containers!

Advanced OpenGL

22.Depth testing

•fragment shader 中可以用 GLSL 语言调用一个 built-in 变量：gl_FragCoord，共有三个维度，xy 代表片段的屏幕坐标，左下角为原点，z 坐标代表真深度（0 到 1），即从 fragment shader 输出的用来进行进一步操作的基本坐标

•深度测试一般执行在 fragment shader 之后，stencil testing 之后

•有些 GPU 支持 early depth testing，即在 fragment shader 之前，不过要求着色器不能再对深度操作更改

•指令：

glEnable(GL_DEPTH_TEST) //启动深度测试，默认不启动，一般需要每一帧刷新深度缓存

glDepthMask(GL_FALSE) //设置不能写入（更新）深度缓存，但是仍然进行深度测试，该指令需要启动深度测试才生效

22.1 Depth test function

•设置深度测试函数的测试方式

glDepthFunc(GL_LESS)

•参数功能表

GL_ALWAYS: The depth test always passes.

GL_NEVER: The depth test never passes.

GL_LESS: Passes if the fragment's depth value is less than the stored depth value.（默认）
GL_EQUAL: Passes if the fragment's depth value is equal to the stored depth value.
GL_LEQUAL: Passes if the fragment's depth value is less than or equal to the stored depth value.
GL_GREATER: Passes if the fragment's depth value is greater than the stored depth value.
GL_NOTEQUAL: Passes if the fragment's depth value is not equal to the stored depth value.
GL_GEQUAL: Passes if the fragment's depth value is greater than or equal to the stored depth value.

22.2 Depth value precision

22.3 Visualizing the depth buffer

•透视投影得到的正则视景体，各个尺度为-1 到 1 然后进行裁剪，经过 fragment shader 之后，最终进行深度测试时，深度范围变换到 0 到 1
•使用非线性深度值的好处是，可以使用纵深很大的视景体（看到的东西远），同时近处的物体仍显示更加准确（便于进行近处物体深度测试）

22.4 Z-fighting

•含义：两个物体位置比较靠近，深度测试难以区分，造成两个物体来回切换显示的问题

22.4.1 Prevent z-fighting

常用的方法：
•手动微调增加物体之间的距离
•增大近平面距离，提高近处物体深度的变化范围
•使用精度更宽的深度缓存

23. Stencil testing

※指令：
glEnable(GL_STENCIL_TEST) //启用
glClear(GL_STENCIL_BUFFER_BIT) //清缓存，可以和其他缓存一起清除
glStencilMask(0xFF) //默认打开写操作，每个比特正常输入
glStencilMask(0x00) //关闭写操作，每个比特变成零
※道理很简单，函数的参数是用来和输入的值 AND 操作，如果 00（即每一位都是 0）自然输入被清零，如果是 FF，则不改变输入结果（0x 表示后面的是十六进制，数字末尾为 H 也一样，B 则为二进制）

23.1 Stencil functions

※设定 stencil 操作的函数功能
•glStencilFunc(GLenum func, GLint ref, GLuint mask)
①func: sets the stencil test function. This test function is applied to the stored stencil value and the glStencilFunc's ref value. Possible options are: GL_NEVER, GL_LESS, GL_LEQUAL, GL_GREATER, GL_GEQUAL, GL_EQUAL, GL_NOTEQUAL and GL_ALWAYS. The semantic meaning of these is similar to the depth buffer's functions.
②ref: specifies the reference value for the stencil test. The stencil buffer's content is compared to this value.
③mask: specifies a mask that is ANDed with both the reference value and the stored stencil value before the test compares them. Initially set to all 1s.
•glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)
①sfail: action to take if the stencil test fails.

②dpfail: action to take if the stencil test passes, but the depth test fails.
③dppass: action to take if both the stencil and the depth test pass.
以上 3 个参数可选功能：
GL_KEEP: The currently stored stencil value is kept.（默认为3个参数共同的值）
GL_ZERO: The stencil value is set to 0.
GL_REPLACE: The stencil value is replaced with the reference value set with glStencilFunc.
GL_INCR: The stencil value is increased by 1 if it is lower than the maximum value.
GL_INCR_WRAP: Same as GL_INCR, but wraps it back to 0 as soon as the maximum value is exceeded.
GL_DECR: The stencil value is decreased by 1 if it is higher than the minimum value.
GL_DECR_WRAP: Same as GL_DECR, but wraps it to the maximum value if it ends up lower than 0.
GL_INVERT: Bitwise inverts the current stencil buffer value.

23.2 Object outlining

•stencil buffer 的作用：
为物体添加轮廓，给物体以“被选中”的印象
绘制镜子中的物体
绘制实时阴影（技术名词：shadow volumes）

24. Blending

•alpha 值：即透明度，取值范围 0-1，0 代表完全透明，1 为完全不透明，中间值（如 0.5）代表这个物体的颜色有 50%由其自身颜色决定，另外 50%由其背后物体决定

24.1 Discarding fragments

•在 fragment shader 中，使用 discard 指令直接放弃当前操作片段
•使用纹理绘图时，有时候使用 REPEAT 包裹方式会造成纹理边界不协调，因为纹理边界部分的片段是由边界处重复平铺的纹理颜色插值而形成的（尤其是需要透明边界时，这种不协调很明显，因为重复平铺的纹理边界处颜色是突变的）；为了避免这个问题，可以使用 CLAMP_TO_EDGE 的方式

24.2 Blending

•混合公式 C_result = C_source*F_source+C_destination*F_destination
C_source: the source color vector. This is the color vector that originates from the texture.
C_destination: the destination color vector. This is the color vector that is currently stored in the color buffer.
F_source: the source factor value. Sets the impact of the alpha value on the source color.
F_destination: the destination factor value. Sets the impact of the alpha value on the destination color.
※指令
•glEnable(GL_BLEND) //启用颜色混合
•glBlendFunc(GLenum sfactor, GLenum dfactor) //设定混合方式函数
可选择参数表
GL_ZERO: Factor is equal to 0.默认为 F_destination
GL_ONE: Factor is equal to 1.默认为 F_source
GL_SRC_COLOR: Factor is equal to the source color vector C _source.
GL_ONE_MINUS_SRC_COLOR: Factor is equal to 1 minus the source color vector.

GL_DST_COLOR: Factor is equal to the destination color vector C_destination
GL_ONE_MINUS_DST_COLOR: Factor is equal to 1 minus the destination color vector.
GL_SRC_ALPHA: Factor is equal to the alpha component of the source color vector C_source.
GL_ONE_MINUS_SRC_ALPHA: Factor is equal to 1-alpha of the source color vector.
GL_DST_ALPHA: Factor is equal to the alpha component of the destination color vector.
GL_ONE_MINUS_DST_ALPHA: Factor is equal to 1-alpha of the destination color vector.
GL_CONSTANT_COLOR: Factor is equal to the constant color vector C_constant.
GL_ONE_MINUS_CONSTANT_COLOR: Factor is equal to 1 - the constant color vector.
GL_CONSTANT_ALPHA: Factor is equal to the alpha component of the constant color vector.
GL_ONE_MINUS_CONSTANT_ALPHA: Factor is equal to 1-alpha of the constant color vector.

- glBlendColor(GLfloat r, GLfloat g, GLfloat b, GLfloat a)
  //可以设定 C_constant（也叫作 GL_BLEND_COLOR）的值
- glBlendFuncSeparate(GLenum sfactor, GLenum dfactor, GLenum sfactor, GLenum dfactor)
  //分别设定三基色混合系数（前两个参数）和透明度的混合系数（后两个参数），可选的枚举参数（symbolic constants）都和前面 glBlendFunc 一样
- glBlendEquation(GLenum mode) //设定混合公式的运算方式

可选择参数表
GL_FUNC_ADD: the default, adds both components to each other: C_result = Src+Dst.
GL_FUNC_SUBTRACT: subtracts both components from each other: C_result = Src-Dst.
GL_FUNC_REVERSE_SUBTRACT: subtracts both components, but reverses order:
C_result =Dst -Src.

24.3 Rendering semi-transparent textures

24.4 Do not break the order

- 透明的物体和深度测试无法完美兼容，所以必须按照一定的顺序进行绘制（针对有透明物体的场景）

①先画所有不透明物体
②将所有透明物体排序
③将透明物体由远及近绘制

- 高阶技术：order independent transparency

25. Face culling

- 定义：一种优化方式，将场景中当前视角下看不见的面放弃绘制，从而节省调用着色器的开销，一般适用于闭合的多面体而言

25.1 Winding order

- 含义：通过顶点的顺序来定义物体的面的朝向，通过只绘制朝向相机的面，而放弃背向相机的面来实现

※面的外朝向和顶点顺序形成右手法则，即如果观察者面向一个面，其顶点顺序为逆时针，则该面的外朝向指向观察者（或称 front-facing，反之为 back-facing）

25.2 Face culling

※指令

- glEnable(GL_CULL_FACE) //启用 face culling
- glCullFace(GL_BACK) //设定 face culling 的机能

可选参数
①GL_BACK: Culls only the back faces.默认值

②GL_FRONT: Culls only the front faces.
③GL_FRONT_AND_BACK: Culls both the front and back faces.
•glFrontFace(GL_CCW)　//设定判别正反面的标准
可选参数
①GL_CCW: 默认值，逆时针方向判别为正面
②GL_CW:顺时针方向为正面

26. Framebuffers

•定义：帧缓存 framebuffer 是 colorbuffer，depthbuffer，stencilbuffer 的组合；启用多个帧缓存可以进行镜像效果和多种后处理（post-processing）等操作（大致的操作就是先将场景 scene 或图像 render 到自定义 framebuffer 上，此时并没有输出图像，然后在自定义 framebuffer 上可以进行对片段的操作实现后处理效果，或者将 scene 当做纹理输出出来铺在镜子上）

26.1 Creating a framebuffer(FBO)

※指令
•glGenFramebuffers(number, &fbo)　//生成 number 个 frame buffer object（FBO）
•glBindFramebuffer(GL_FRAMEBUFFER, fbo)
第一个参数可选值
①GL_FRAMEBUFFER:绑定的 FBO 可读可写
②GL_READ_FRAMEBUFFER:绑定的 FBO 只能进行读操作
③GL_DRAW_FRAMEBUFFER:绑定的 FBO 只能被进行写操作
【当不绑定 default framebuffer 时，rendering commands 对于屏幕没有影响，只是影响绑定的自定义 framebuffer，此时称为 off-screen redering，必须绑定 0 到 GL_FRAMEBUFFER 才能使用 default framebuffer】
•glCheckFramebufferStatus(GL_FRAMEBUFFER) //查看 framebuffer 的状态
可能的返回值
①GL_FRAMEBUFFER_COMPLETE: framebuffer 是完整的，即创建完成状态
•glDeleteFramebuffers(1,&fbo) //删除 framebuffer
【framebuffer 的创建不是只创建一个 FBO 就完成了，因为是一系列 buffer 的组合，所以必须 attach 相关的 buffer 才算完成，这里面必须至少有一个 colorbuffer，其他 depth/stencil 可选】

26.1.1 Texture attachment

※为 framebuffer attach color buffer 或其他 buffer 有两种办法，一种是使用 texture 来代替 buffer，另一种是使用真正的 buffer object（所以应该讲 framebuffer 包含的是 attachment，而不（全）是 render buffers）；实际上真正的 buffer object 不能用于读取数据，但是操 作 快捷方便，texture 的格式和 buffer 的格式不同，因此必要时需要转换格式，但是可以读取数据
•创建 attachment 的第一种办法，将 color/depth/stencil buffer 创建成为一个 texture，和 framebuffer 相连：
第一步：创建一个普通的 texture 对象，不需要填充其内容，设为 NULL（比如用 SOIL），只需要设定基本参数即可，包括尺寸大小等
第二步：将其 attach 到 framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER,GL_COLOR_ATTACHMENT0,GL_TEXTURE_2D,texture,0)

参数解释：

①target: the framebuffer type we’re targeting (draw, read or both).

②attachment: the type of attachment we’re going to attach. 数字表示可以有多个

1）GL_DEPTH_ATTACHMENT0: attach 一个可以用来存储 depth buffer 的 texture，则 texture image 函数设定为：

glTexImage2D(GL_TEXTURE_2D,0,GL_DEPTH_COMPONENT,800,600,0,GL_DEPTH_COMPONENT,GL_UNSIGNED_BYTE, NULL);

2）GL_STENCIL_ATTACHMENT0: attach 一个可以用来存储 stencilbuffer 的 texture，则 texture image 函数设定为：

glTexImage2D(GL_TEXTURE_2D,0,GL_STENCIL_INDEX,800,600,0,GL_STENCIL_INDEX,GL_UNSIGNED_BYTE, NULL);

3）GL_COLOR_ATTACHMENT0: 存储 colorbuffer 的 texture，和一般的 texture 设定方法一样：

glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,800,600,0,GL_RGB,GL_UNSIGNED_BYTE,NULL)

4）GL_DEPTH_STENCIL_ATTACHMENT: 将 depth 和 stencil buffer 合为一个 texture，depth buffer 使用 24bit/frag，stencil buffer 8bit/frag，则 texture image 函数设定为：

glTexImage2D(GL_TEXTURE_2D,0,GL_DEPTH24_STENCIL8,800,600,0,GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL);

③textarget: the type of the texture you want to attach；可以是 cubemap 的某个面的枚举名，如 GL_TEXTURE_CUBE_MAP_POSITIVE_X + i

④texture: the actual texture to attach；可以是一个 cubemap，则前一个参数决定是其哪个面

⑤level: the mipmap level. We keep this at 0.

26.1.2 Renderbuffer object attachment

※因为 renderbuffer 一般都是只写不读的，所以一般用作 depth 或 stencil attachment

•创建方法与指令

glGenRenderbuffers(1,&rbo)

glBindRenderbuffers(GL_RENDERBUFFER, rbo)

glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8,800,600)

//将 RBO 设为 depth and stencil buffer

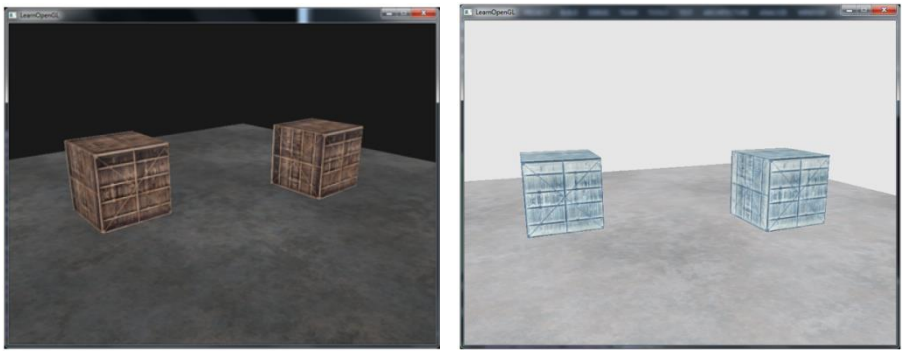
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER, rbo) //将 RBO attach 到 FBO

26.2 Rendering to a texture

26.3 Post-processing

26.3.1 Inversion

•将颜色反过来，即用 1 减去每个颜色向量的每个分量

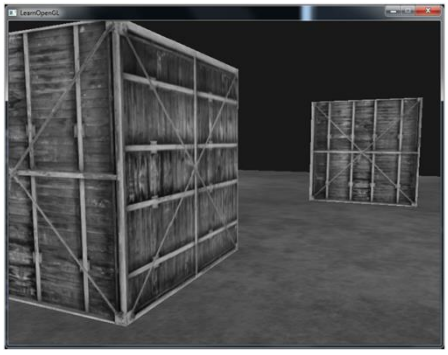


26.3.2 Grayscale（灰度化，黑白化）

•将三基色加权平均，然后将平均值一致地赋给结果的三个分量；根据人眼特点，使用以下权重得到的结果更加真实准确

average = 0.2126 * color.r + 0.7152 * color.g + 0.0722 * color.b

•灰度化可以用于检测该颜色组合的亮度 brightness，因为只输出一个值，可以作为亮度值



26.4 Kernel effects

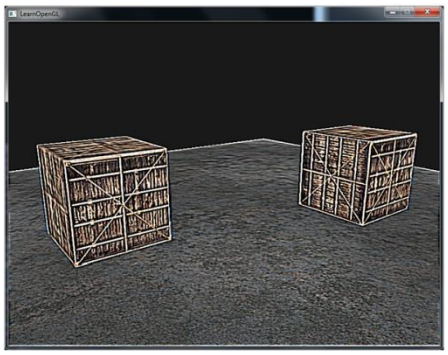
•定义：使用一个（一般）3x3 的矩阵，矩阵的所有元素之和一般为 1，将该矩阵套在每一个片段上，使用的结果是该片段的颜色由周围加自身 9 个像素加权得到，权重即为对应的矩阵元素

•权重之和不为 1 时，可能会造成结果变亮或变暗

26.4.0 Sharpen

•即锐化，使用类似下列的 Kernel

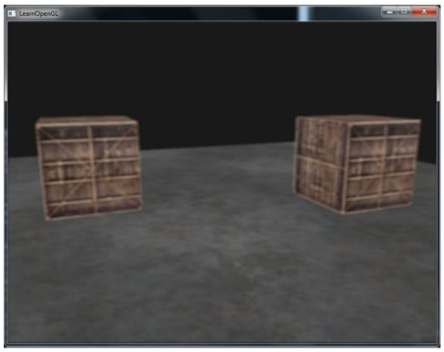
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$



26.4.1 Blur

•模糊效果，适用于：酒醉状态，近视眼，运动模糊等，使用类似下列的 Kernel

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} /16$$



26.4.2 Edge detection

•高亮边缘，暗化其他，使用类似下列的 Kernel

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

27. Cubemaps

•含义：由 6 个 2D textures 组成的一个立方体纹理，以立方体中心为原点，则 cubemaps 的纹理坐标由原点到纹理像素的方向向量决定（长度无关）

•优势：向量决定纹理坐标，而不是另外一套二维平面坐标，这样可以方便和顶点本身的位置进行关联

27.1 Creating a cubemap

※指令

•glGenTextures(1, &textureID)

•glBindTexture(GL_TEXTURE_CUBE_MAP, textureID)

•glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X,0,GL_RGB,width,height,0, GL_RGB,GL_UNSIGNED_BYTE, image)

必须针对每一个面（6 次）声明一次 glTexImage2D，第一个参数代表哪一个面

①GL_TEXTURE_CUBE_MAP_POSITIVE_X //(Right)

②GL_TEXTURE_CUBE_MAP_NEGATIVE_X //(Left)

③GL_TEXTURE_CUBE_MAP_POSITIVE_Y //(Top)

④GL_TEXTURE_CUBE_MAP_NEGATIVE_Y //(Bottom)

⑤GL_TEXTURE_CUBE_MAP_POSITIVE_Z //(Back)

⑥GL_TEXTURE_CUBE_MAP_NEGATIVE_Z //(Front)

※上面的枚举参数是按照顺序排列的，因此②的值=①+1，以此类推，可以利用循环

•glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,GL_LINEAR)

glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR)

glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_EDGE);

glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_EDGE);

glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_R,GL_CLAMP_TO_EDGE);

必须增加一个维度的纹理坐标，建议使用 CLAMP_TO_EDGE，因为两个纹理面之间有可能因为离散原因存在缝隙

※着色器 GLSL

uniform samplerCube cubemap

存放的 cubemap 的着色器数据类型必须更改

27.2 Skybox

27.3 Loading a skybox

•第一步：创建一个 cubemap，按照对应的面布置 6 个纹理

27.4 Displaying a skybox

•要点：render skybox 要一直以相机为中心，因此不能使用 model matrix，而且 view matrix 的平移量要去除（只需要将 mat4 变换为 mat3 形式，再变换回来即可清零平移量）

27.5 An optimization

•要点：最后绘制背景（skybox）因为背景的深度是确定值（最大），这样可以优化计算，避免多余计算背景

•方法：通过手动设定 vertex shader 中，经过透视变换之后的 z 值（设为=w 值），在 vertex shader 结束后的透视除法中，深度便恒为 1

27.6 Environment mapping

•含义：使用 environment cubemaps（skybox 只是使用这种技术建立背景的一种应用）的方法来为物体赋予反射（镜像）和折射特性（或纹理）的技术，称为 environment mapping。

27.7 Reflection

•方法：用视点和物体反射点连线的反射光向量来直接从 cubemap 上取样，近似模拟的情况是物体在无穷大环境下处于正中心位置

•reflection map：和 diffuse map and specular map 类似，用来专门对物体每个片段进行与反射相关的系数的取样的 texture image，与反射相关的系数如反射强度（intensity）

27.8 Refraction

•回顾

①折射率 refractive index：$n = \frac{c}{v}$，即光在真空中的速度 c 与在介质中的速度 v 之比

真空的折射率自然为 1，大部分气体（如空气）的折射率也近似为 1（比 1 大一点点）

②斯涅尔定律 Snell's Law：$n_1 \sin\theta_1 = n_2 \sin\theta_2$

光的入射和折射角的正弦比值与折射率成反比

•作用：使物体呈现透明效果，比如让物体变成玻璃材质，能够透过物体看到后面的场景（折射情况下虽然有扭曲，但是足够真实）

•注意：一般为了简化，只是在物体的一个表面折射（single-side refraction）；物理真实情况下，显然光线穿过物体至少要两次折射

27.9 Dynamic environment maps

•含义：因为反射或透明的物体周围不仅仅只有背景，还有其他的物体，这些物体可能存在着运动。通过 framebuffers 创造物体周围 6 个方向观察到的场景的 texture，然后将其制作为 cubemap 来作为物体的 environment maps，这项技术称为 dynamic environment maps，也就是实时更新的 cubemap

28. Advanced Data

•概念：

①缓存 buffer：一个管理一个特定内存区域的对象

②buffer target：将缓存绑定 bind 到一个 buffer target 上，才能通知 OpenGL 如何操作这个缓存，一般一个 buffer target 只能绑定一个 buffer

※指令：

第一种为缓存赋值方法

•glBufferData(…)

//用来向缓存输入数据的函数，整体输送的方式，如果输送的数据是 NULL，那么函数仍将划分内存给缓存，但是不会填充数据

•glBufferSubData(GL_ARRAY_BUFFER, offset, sizeof(data), &data)

//用来向缓存输入数据，可以定义从何处开始与结束，offset 即为从缓存开头处算起开始填充数据的位置（字节为单位），使用这个函数的前提是必须有 glBufferData 已经分配内存给缓存

第二种为缓存赋值方法

•void *ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY)

•memcpy(ptr, data, sizeof(data))

•glUnmapBuffer(GL_ARRAY_BUFFER)

//通知 OpenGL 结束指针操作，函数返回 GL_TRUE 表明数据成功赋值

28.1 Batching vertex attributes

•介绍另一种传送顶点属性的方式，即每个顶点属性都是一个连续的数组，而不是一个顶点的所有属性按照顺序间隔排列。方法实际上类似

28.2 Copying buffers

※指令

glCopyBufferSubData(GLenum readtarget, GLenum writetarget, GLintptr readoffset, GLintptr writeoffset, GLsizeiptr size)

//从 readtarget 复制到 writetarget，总共 size byte 大小的信息，读写的位置由 offset 决定

readtarget 和 writetarget 可以是一般的 buffertarget（VERTEX_ARRAY_BUFFER 等），只要绑定上对应的实际 buffer 即可

另外，也可将要复制和读取的 buffer 绑定在特殊的两个 target 上，GL_COPY_READ_BUFFER 和 GL_COPY_WRITE_BUFFER，这样可以解决复制两个同类的 buffer 却只能绑定一个的问题（当然特殊的 target 可以和一般的 target 混用）

29. Advanced GLSL

29.1 GLSL's built-in variables

种类描述：(所属着色器类型，输入或输出类型/可读或可写，数据维度)

gl_Position:(vertex/geometry,out,4d)clip-space 的位置坐标，即进行完模型视点透视变换之后的坐标

gl_FragCoord:(fragment,in,3d)进行完着色计算的片段的屏幕坐标和真深度

gl_PointSize: (vertex,out,float)决定图元 GL_POINTS 的输出点尺寸

gl_VertexID:(vertex,in,int)做 indexed rendering（用 glDrawElements）时，储存当前被处理的顶点的 index，非 indexed rendering（用 glDrawArrays）时，储存当前已经被处理过的顶点数量

gl_FrontFacing:(fragment,in,bool)显示当前片段是否处在一个 front face 上

gl_FragDepth:(fragment,out,float)用来自定义片段深度的量（0 到 1）

gl_in[]:(geometry,in,struct)储存输入的顶点数据的数组，包含{

```
vec4 gl_Position;//代表从顶点着色器输入的位置数据
float gl_PointSize;
float gl_ClipDistance[];}//使用时要带成员符号，无成员符号的 gl_Position 是输出量
```

gl_InstanceID:(vertex,in,int)使用 instancing 绘图时，储存当前被操作的 instance 编号，从 0 开始计数

29.2 Vertex shader variables

29.2.1 gl_PointSize

•使用 OpenGL 的另一图元 GL_POINTS 时，可以用 glPointSize 函数更改点的尺寸，也可以在 vertex shader 中对 gl_PointSize 赋值来决定其大小，后者可以使每个点的大小都不一

•在 vertex shader 中改变点的大小默认是关闭的，必须先启用

glEnable(GL_PROGRAM_POINT_SIZE)

29.2.2 gl_VertexID

29.3 Fragment shader variables

29.3.1 gl_FragCoord

•主要功能：在屏幕的不同部分进行不同的着色计算以比较结果（仍然是同一个且连续的场景，比如比较不同的光照技术产生的影响；和一般意义的分屏显示不一样，后者应该需要用不同的 viewport）

29.3.2 gl_FrontFacing

•在不使用 face culling 功能时，gl_FrontFacing 才有作用

29.3.3 gl_FragDepth

•如果片段着色器没有对 gl_FragDepth 进行操作，则自动从 gl_FragCoord.z 取值作为深度

•对 gl_FragDepth 进行操作后，将会终止所有的 early depth test；OpenGL4.2 以后，可以条件性使用 gl_FragDepth 与 early depth test

在 fragment shader 首行加入：

layout (条件) out float gl_FragDepth;

“条件”可选择：

①depth_any: The default value. Early depth testing is disabled and you lose most performance.

②depth_greater: You can only make the depth value larger compared to gl_FragCoord.z.

③depth_less: You can only make the depth value smaller compared to gl_FragCoord.z.

④depth_unchanged: If you write to gl_FragDepth, you will write exactly gl_FragCoord.z.

29.4 Interface blocks

•实际上是对着色器中的 in 和 out 数据使用类似 struct 的结构

举例：

顶点着色器	片段着色器
#version 330 core	#version 330 core
layout (location = 0) in vec3 position;	out vec4 color;
uniform mat4 model;	in VS_OUT{
out VS_OUT{	vec2 TexCoords;
vec2 TexCoords;	} fs_in;
} vs_out;	uniform sampler2D texture;
void main(){	void main(){
vs_out.TexCoords = texCoords; }	color = texture(texture, fs_in.TexCoords);}

29.5 Uniform buffer objects

•UBO 是用来储存和设置 uniform 的值，这些 uniform 在多个 shader 中共享，这样避免了重复设置，但是对于每个 shader 中独有的 uniform 仍然需要单独设定

举例：（UBO 是 OpenGL 中的对象，在 shader 中对应的是 uniform block）

#version 330 core

```
layout (location = 0) in vec3 position;
layout (std140) uniform Matrices{
mat4 projection;
mat4 view;};
uniform mat4 model;
void main(){
gl_Position = projection * view * model * vec4(position, 1.0);}
```

•uniform block 中的元素不是 block 的成员，引用时不加成员符号

29.6 Uniform block layout

•uniform block 中的值储存在 UBO 中，也就是内存的一片保留区，必须对内部储存数据的方式进行管理

uniform memory layout

①shared layout: 分配方式由 hardware 决定，但是保持数据顺序；一旦决定分配方式，不同程序之间共享，数据位置（offset）由 glGetUniformIndices 查询

②packed layout: 分配方式不同程序之间不共享，数据位置由 glGetUniformIndices 查询

③std140 layout: 分配方式和数据位置依据特定规则：

•Layout rule（N 代表 4 字节）

Scalar e.g. int or bool: Each scalar has a base alignment of N.

Vector: Either 2N or 4N, this means that a vec3 has a base alignment of 4N.

Array of scalars or vectors: Each element has a base alignment equal to that of a vec4.

Matrices: Stored as a large array of column vectors, where each of those vectors has a base alignment of vec4.

Struct: Equal to the computed size of its elements according to the previous rules, but padded to a multiple of the size of a vec4.

•base alignment: 每个数据占据的实际空间，包括自身空间还有空余空间（根据硬件不同，数据的存储不一定能做到紧密排列，空余空间称为 padding space）

•aligned offset: 在 UBO 中 uniform 数据的起始位置（相对于首地址），不仅仅由当前数据之前的数据的位置和 base alignment 决定，也必须为当前数据的 base alignment 的整数倍

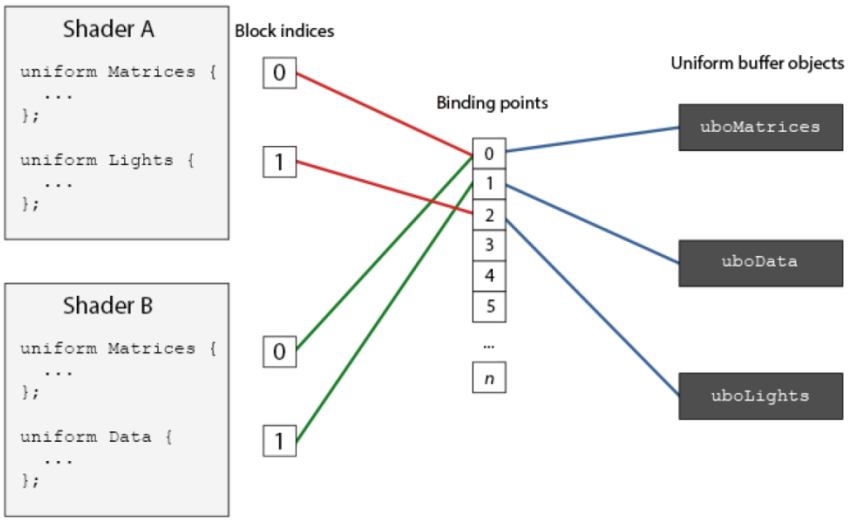
29.7 Using uniform buffers

①生成 UBO

```
GLuint uboExampleBlock;
glGenBuffers(1, &uboExampleBlock);
glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);
glBufferData(GL_UNIFORM_BUFFER, 150, NULL, GL_STATIC_DRAW);
glBindBuffer(GL_UNIFORM_BUFFER, 0);
```

②连接 uniform block 和 UBO

•uniform block 和 UBO 不是直接连接，而是通过 binding points，UBO 和 bp 是一一对应的，而不同程序（shader）中的 uniform block 根据需要选择对应的 bp



•连接 uniform block

GLuint lights_index = glGetUniformBlockIndex(shaderA.Program, "Lights");

glUniformBlockBinding(shaderA.Program, lights_index, 2);

•OpenGL4.2 以后，取消了 bp，在 shader 中使用 layout specifier 即可

layout(std140, binding = 2) uniform Lights { ... };

•连接 UBO

glBindBufferBase(GL_UNIFORM_BUFFER, 2, uboExampleBlock);

// 或者使用另一个指令

glBindBufferRange(GL_UNIFORM_BUFFER, 2, uboExampleBlock, 0, 150);

③UBO 中填充数据

方法和一般 buffer object 一样

glBindBuffer(GL_UNIFORM_BUFFER, uboExampleBlock);

GLint b = true; //GLSL 中 bool 是 4 字节

glBufferSubData(GL_UNIFORM_BUFFER, 142, 4, &b);

glBindBuffer(GL_UNIFORM_BUFFER, 0);

29.8 A simple example

•UBO 的优点：

方便管理大量 uniform 并且更改其值

提高可使用的 uniform 数据的总数（OpenGL 对于 uniform 数据有上限，可查询 GL_MAX_VERTEX_UNIFORM_COMPONENTS）

30. Geometry Shader

•含义：几何着色器处于顶点和片段着色器之间，输入的数据是以图元为单位的顶点组合（数组），输出的数据和顶点着色器直接传送给片段着色器的数据与格式相同，几何着色器可以更改顶点的数据（包括生成新的顶点）最终生成新的图元。

•作用：在绘制大量需要重复出现的图形时，如果直接使用顶点坐标（用到 vertex buffer）来一一定义这些图形不仅麻烦而且效率低（vertex buffer 利用内存），几何着色器直接利用 GPU 生成图形更快。

•代码组成：

①layout specifier

#version 330 core

```
layout (points) in;
layout (line_strip, max_vertices = 2) out;
```
输入的图元可选择：（括号中数字代表一个图元最少包含的顶点数）
points: when drawing GL_POINTS primitives (1).
lines: when drawing GL_LINES or GL_LINE_STRIP (2).
lines_adjacency: GL_LINES_ADJACENCY or GL_LINE_STRIP_ADJACENCY (4).
triangles: GL_TRIANGLES, GL_TRIANGLE_STRIP or GL_TRIANGLE_FAN (3).
triangles_adjacency:GL_TRIANGLES_ADJACENCY or
GL_TRIANGLE_STRIP_ADJACENCY(6).
输出的图元可选择：（并且定义每次运行输出最大顶点数，不是每个输出图元最大顶点数）
points
line_strip
triangle_strip
②in/out 和数据声明
```
in VS_OUT {
vec3 color;
} gs_in[];
```
几何着色器的输入是以图元为单位的顶点组合，因此所有的 in 数据都必须是数组（维度不用注明，out 数据没有此要求），其他的声明方式和其他着色器类似
③主程序
```
void main(){
fColor = gs_in[0].color;
gl_Position = position + vec4(-0.2f, -0.2f, 0.0f, 0.0f);
EmitVertex();
gl_Position = position + vec4( 0.0f,    0.4f, 0.0f, 0.0f);
fColor = vec3(1.0f, 1.0f, 1.0f);
EmitVertex();
EndPrimitive();}
```
主程序中由 EmitVertex()和 EndPrimitive()两个函数负责决定输出；每个 EmitVertex()输出一个顶点的数据（在此之前的语句需要把该顶点的所有 out 数据设定好，还有 gl_Position），输出了图元的所有顶点后由 EndPrimitive()收尾
- 可见几何着色器的输出和顶点着色器是一样的（顶点为单位）；有了几何着色器，顶点着色器的输出也没有变化，只是几何着色器的输入必须是数组（图元为单位）

30.1 Using geometry shaders
- 使用方法编译方法等和其他着色器相同

30.2 Let's build some houses

30.3 Exploding objects
- 用几何着色器将每个顶点沿着其法向量方向移动一段距离

30.4 Visualizing normal vectors
- 用来检验法向量是否设置正确，通过几何着色器将每个顶点的法向量绘制出来

31. Instancing
- 含义：将调用一次 glDrawArrays 或 glDrawElements 所绘制的对象（可认为是一个模型）实体化，即相当于重复调用这些语句，每次调用可设定一些不同数据。用于绘制大量重复

的模型，不同于几何着色器用于绘制重复的图形。因为每次调用绘制语句需要耗费大量的时间来在 CPUGPU 之间传送指令与数据，而直接让 GPU 作图是很快的。

•指令：只需要将 glDrawArrays 或 glDrawElements 替换为：

glDrawArraysInstanced(GL_TRIANGLES, 0, 6, 100); //增加的参数即为 instance 数量

glDrawElementsInstanced()

31.1 Instanced arrays

•作用：赋予不同的 instance 不同的一些数据用来区别每个 instance

•使用方法：作为 vertex attribute 进行输入，方法和其他顶点属性类似

特殊的指令：

```
glEnableVertexAttribArray(2);
glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(GLfloat), (GLvoid*)0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
glVertexAttribDivisor(2, 1);
```

//第一个参数被操作属性编号，第二个参数 attribute divisor，0 代表每顶点（每循环）更新，默认值；1 代表每 instance 更新，2 代表每 2 个 instance 更新

31.2 An asteroid field

•VAO, vertex attribute location:

每个 location 的 vertex attribute 最多只能是一个 vec4 的量，但是仍然可以通过这个方式传送一些复杂类型的量给顶点，如 mat4（作为 instanced arrays）。方法是在 glVertexAttribPointer 指令中利用多个 location 来存储复杂类型的量，在 shader 中则按照第一个 location 储存的量正常使用，当然该 attribute 后面定义的 attribute 必须留出前一个所占用的 location（即 shader 中照常使用，OpenGL 主程序中分开储存）

32. Anti Aliasing

SSAA: super sample anti-aliasing，通过取更多的采样点（提高分辨率）在原分辨率下输出

MSAA: multisample anti-aliasing

32.1 Multisampling

rasterizer: 光栅化处理器，处于顶点着色器和片段着色器之间，将顶点信息转化为像素片段

•基本原理：每个像素采用多个采样点（影响到 color, depth, stencil 所有缓冲的大小），首先仍然对每个像素中心进行颜色插值，然后将颜色赋给图形覆盖住的采样点，最后每个像素内对所有采样点求平均作为像素颜色

•采样点 sample point，sampling；插值点 interpolation point

32.2 MSAA in OpenGL

•指令：

```
glfwWindowHint(GLFW_SAMPLES,4)
//设定 default framebuffer 是 4 倍多重采样
glEnable(GL_MULTISAMPLE)
//开启多重采样功能，一般是默认开启的
```

32.3 Off-screen MSAA

32.3.1 Multisampled textured attachments

•创建 texture

```
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, tex);
glTexImage2DMultisample(GL_TEXTURE_2D_MULTISAMPLE, 4, GL_RGB, width,  height,
```

GL_TRUE);
glBindTexture(GL_TEXTURE_2D_MULTISAMPLE, 0);
•Attach texture
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_TEXTURE_2D_MULTISAMPLE, tex, 0);
32.3.2 Multisampled renderbuffer objects
•将 glRenderbufferStorage 改为
glRenderbufferStorageMultisample(GL_RENDERBUFFER, 4, GL_DEPTH24_STENCIL8,
width, height);
32.3.3 Render to multisampled framebuffer
•off-screen 的多重采样 framebuffer 没有办法直接使用（显示或后处理取样），需要用到一个转换函数
•glBlitFramebuffer 将 READ 绑定的 framebuffer 拷贝到 DRAW 绑定的 framebuffer 上，同时 resolve multisampled buffers
glBindFramebuffer(GL_READ_FRAMEBUFFER, multisampledFBO);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBlitFramebuffer(0, 0, width, height, 0, 0, width, height,
GL_COLOR_BUFFER_BIT, GL_NEAREST);
//将 multisampled off-screen framebuffer 复制到 default framebuffer，然后输出
•同理，如何要使用 multisampled framebuffer 进行后处理等操作，必须先将其转换到一个中间 framebuffer，仍然利用上述语句

32.4 Custom Anti-Aliasing algorithm

•在 shader 中仍然可以直接采样 multisampled texture image：
uniform sampler2DMS screenTextureMS　//定义 uniform 的变化
vec4 colorSample = texelFetch(screenTextureMS, TexCoords, 3)　//取 texture 颜色函数的变化

Advanced Lighting

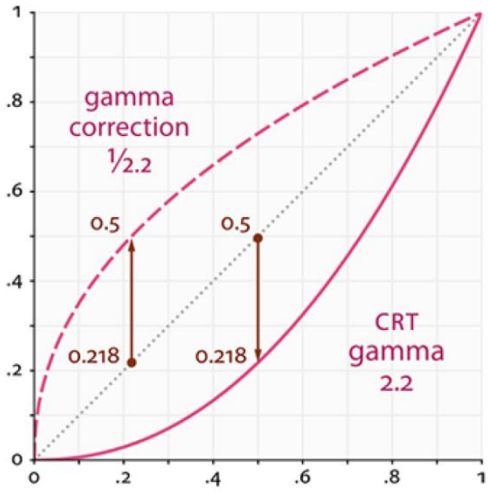
33. Advanced Lighting

33.1 Blinn-Phong

略

34. Gamma Correction

•gamma：一般显示器（cathode-ray tube，CRT）亮度与电压成指数关系，指数值叫做 gamma；人眼对于亮度的区分也不是线性的，对于较暗的亮度变化更加敏感，为了更加效率的利用这一点，显示器的输出始终保留着这种非线性转换
•gamma correction：人眼对于亮度的感觉并非是物理真实的，有时为了作图更加贴近物理真实（不管人眼，毕竟人眼也需要面对物理真实）特意将输出的亮度进行数学操作，这种操作会和显示器的内设非线性转换抵消，输出线性的物理亮度。这种操作叫做伽马校正
•非线性亮度空间示意：对于实线，横坐标为人眼感知亮度，纵坐标为物理实际亮度



伽马校正有时候不是必须的，有时候会适得其反，总之是一种对于亮度变化的优化数学操作

34.1 Gamma correction

如何进行伽马校正，两种方法

①OpenGL built-in

指令：

glEnable(GL_FRAMEBUFFER_SRGB);

•对于之后的所有的 drawing command，都进行针对 sRGB 空间的伽马校正

•sRGB 即大致相当于 gmma 值 2.2 的颜色亮度空间，也是大多数家庭设备的标准

②GLSL

•手动在片段着色器的最后，将颜色输出值每个分量乘幂 1.0/gamma

•对于每个片段着色器都需要操作，可以通过后处理的方式只进行一次操作

34.2 sRGB textures

•有些 texture 是在 sRGB 空间定义下绘制的，因此其颜色值已经进行过一次伽马校正，为了使用这些颜色值，需要先进行伽马校正的逆操作来在线性空间中使用颜色

①GLSL 方法

②OpenGL 函数

glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);

•GL_SRGB and GL_SRGB_ALPHA 是两种对应 sRGB 空间的 texture 格式，如此使用后 OpenGL 自动转换颜色值，因此用户可以照常使用颜色值，跟线性空间一样

※注意：

使用 texture 的格式上有一些习惯性的设定，diffuse texture 一般都是 sRGB 格式，因为是用来为图形上色的，而 specular 和 normal maps 等用来提取光照参数的一般都是线性

34.3 Attenuation

•使用伽马校正会对颜色亮度有影响，因此这种操作和光线衰减机制存在相互作用；举个简单例子：物理实际光线衰减与距离平方成反比，而如果不使用伽马校正，那么这种衰减实际上是与距离的 4.4 次方成反比。因此使用伽马校正对于使用物理真实的图形算法非常重要

34.4 Additional resources

www.cambridgeincolour.com: more about gamma and gamma correction.

http://www.cambridgeincolour.com/tutorials/gamma-correction.htm

blog.wolfire.com: blog post by David Rosen about the benefit of gamma correction in graphics rendering.
http://blog.wolfire.com/2010/02/Gamma-correct-lighting
renderwonk.com: some extra practical considerations.
http://renderwonk.com/blog/index.php/archive/adventures-with-gamma-correct-rendering/

35. Shadow Mapping

•实时 real-time 图形学中，一个完美的阴影算法还没有出现，shadow mapping 是当今比较常用并且易于实现的算法

35.1 Shadow mapping

•原理：

1.以光源为视点做一个全景深度纹理（depth map/texture），记录每个光照方向上被光源照亮的点的深度

2.从视点处正常生成图像，对于每个片段，先将其变换到光源视点坐标系，再与光源深度纹理进行深度测试即可得知其是否处于阴影中

35.2 The depth map

•创建一个 depth map 即创建一个只有深度 buffer 的 framebuffer，因为没有 color buffer 所以必须通过指令在 framebuffer 的配置环节指明，depth texture 中 depth 值仅仅储存在第一个元素中，即 r/x/s 分量中

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
    GL_TEXTURE_2D,depthMap, 0);
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

35.2.1 Light space transform

•对于深度的计算，根据光源类型来选择不同的投影方式，平行光选择正交投影，点光源选择透视

•视点即光源位置，一般看向场景中心

35.2.2 Render to depth map

•由于只需要 render 到深度缓存，使用特殊的简易着色器节约开销，即使用空的片段着色器（主函数内容为空，没有任何输入输出声明）

```
#version 330 core
void main(){
// gl_FragDepth = gl_FragCoord.z;}
```

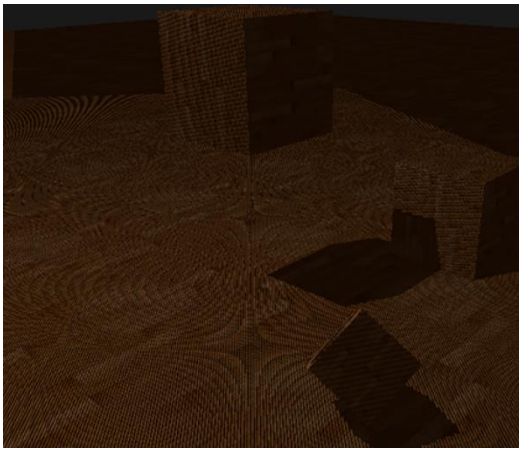
35.3 Rendering shadows

•判别片段处于阴影中之后，只需要将 diffuse 和 specular 因素消除即可，保留 ambient

35.4 Improving shadow maps

35.4.1 Shadow acne

•定义：阴影条纹，并非出现在阴影区，而是出现在本应被完全照亮的区域，呈现明暗相间的条纹；原因是 shadow map 存在分辨率限制，因此不同位置的几处片段可能在 light space 中对应相同的深度，而实际上这些片段在观察空间中深度不同，因而可能出现一部分被照亮，一部分被判别为阴影的情况



•解决方法：

```
float bias = max(0.05 * (1.0 - dot(normal, lightDir)), 0.005);
float shadow = currentDepth - bias > closestDepth ? 1.0 : 0.0;
```

设定一个 offset 值，根据片段所在平面与光线的位向来确定，通过这个值将片段的深度减小来避免这个现象

•bias 值根据场景或试验而定，一般的操作仅仅只是通过调试 bias 值来获得较好的结果为止

35.4.2 Peter panning

•因为 bias 值过大或者单纯引入 bias 导致实际 currentDepth 值修正后过小，原本应该在阴影中的片段，脱离阴影被照亮

•一般设置合理的 bias 值即可

??在进行绘制光源 depth texture 时，进行 face culling

35.4.3 Over sampling

•定义：因为光源深度纹理有一个范围，一是纹理坐标的 st 取值范围，二是深度值 0 到 1 的范围；当观察空间中片段的光源空间深度超过这个范围，就会出现问题

•解决方法：

1.将深度纹理设定为 GL_CLAMP_TO_BORDER，然后设定边界值为 vec4(1.0)，即所有超过边界的实际深度值都会小于 1.0，因此都没有阴影

2.将所有深度值超过 1.0 的片段都设为非阴影区

•概括而言，近处有阴影，远处没有

35.5 PCF

•问题：阴影锯齿

•PCF（percentage-closer filtering）：多次从 depth map 取值，每次有细微的纹理坐标差异，最后整合平均以柔化锯齿，即缓慢过渡锯齿边缘的亮度

35.6 Orthographic vs projection

35.7 Additional resources

•Tutorial 16 : Shadow mapping: similar shadow mapping tutorial by opengl-tutorial.org with a few extra notes.
http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

•Shadow Mapping - Part 1: another shadow mapping tutorial by ogldev.
http://ogldev.atspace.co.uk/www/tutorial23/tutorial23.html

•How Shadow Mapping Works: a 3-part YouTube tutorial by TheBennyBox on shadow mapping and its implementation.
https://www.youtube.com/watch?v=EsccgeUpdsM

•Common Techniques to Improve Shadow Depth Maps: a great article by Microsoft listing a large number of techniques to improve the quality of shadow maps.
https://msdn.microsoft.com/en-us/library/windows/desktop/ee416324%28v=vs.85%29.aspx

36. Point Shadows

•前面一个章节讲的是 directional shadow mapping，即平行光源的 shadow map，采用 2D texture；本章讲点光源阴影，因为点光源的阴影在各个方向，所以采用 cubemap；point (light) shadows 或称 omnidirectional shadow maps

36.1 Generating the depth cubemap

•创建一个空的 cubemap texture 来作为一个 framebuffer 的深度缓存
•指令

```
glBindFramebuffer(GL_FRAMEBUFFER, depthMapFBO);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthCubemap, 0);
//可以直接绑定 cubemap 到 framebuffer 上
glDrawBuffer(GL_NONE);
glReadBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

36.1.1 Light space transform

•因为光源的光线朝向四面八方，作为 cubemap 要将空间在光源处分成 6 个直角空间，即透视投影的视角为 90 度
•另外将 cubemap 的 6 个面或 6 个轴和世界坐标系对齐，这样 cubemap 的纹理坐标（即向量）和世界坐标系中的向量无需转换（只是平移不影响数值）

36.1.2 Depth shaders

•总共 6 个光源空间转换矩阵，1 个场景，因此将光源空间转换过程放在 geometry shader 中，在附有 cubemap texture 的 framebuffer 上作图时，几何着色器中可以使用 gl_Layer 在不同的纹理面上作图
•为方便计算，可以将深度手动设定为片段到视点（光源）距离来进行存储，方便后续比较

36.2 Omnidirectional shadow maps

36.2.1 Visualizing cubemap depth buffer

36.3 PCF

36.4 Additional resources

•Shadow Mapping for point light sources in OpenGL: omnidirectional shadow mapping tutorial by sunandblackcat.
http://www.sunandblackcat.com/tipFullView.php?l=eng&topicid=36
•Multipass Shadow Mapping With Point Lights: omnidirectional shadow mapping tutorial by ogldev.
http://ogldev.atspace.co.uk/www/tutorial43/tutorial43.html
•Omni-directional Shadows: a nice set of slides about omnidirectional shadow mapping by Peter Houska.
http://www.cg.tuwien.ac.at/~husky/RTR/OmnidirShadows-whyCaps.pdf

37. Normal Mapping

•normal mapping 或 bump mapping，用来赋予每个片段一个法向的贴图，以此来赋予一个纹理不同部分不同的光照效果，一般是用来在平面上模拟缝隙、裂痕、浮雕等效果

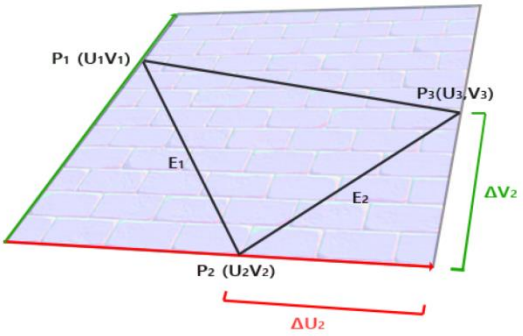
37.1 Normal mapping

•normal map，即存储 normal 向量的纹理，纹理一般都是存储 3 或 4 维颜色值，所以范围是[0,1]，法向量归一后范围[-1,1]，因此法向量是经过转换后以颜色形式存储的；一般的法向贴图都是蓝色为主，因为大部分法向量都是朝向外侧，即(0,0,1)方向，换成颜色值即蓝色

37.2 Tangent space

•与颜色不同，法向量有方向，同时也受坐标系影响，如果法向贴图使用世界坐标系，那么无论实际面如何运动，其法向量都不会变。所以规定法向贴图所使用的坐标系为tangent space，即网格单元（如三角形）表面的 local space，包含三个基，tangent，bitangent 和 normal，与世界坐标系之间的坐标系变换矩阵称为 TBN matrix

• 切线空间的 normal 是实际面（网格三角形）的法线，而另外两个基选取和法向贴图坐标系的轴相重合的方向（如下图所示）。以上是针对单个网格单元的理论定义，实用中由于顶点经常被多个相邻网格单元共享，且切线空间的基要作为顶点属性和每个顶点绑定。所以切线空间的法线等于顶点法线，和其他两个基都取相邻网格单元的平均值保存在共享顶点中。这种做法和顶点法线的平均道理是一样的，为了平滑结果。



•计算 tangent 和 bitangent 基向量（坐标系视顶点所采用的坐标系）的公式为：

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

$$= \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

• 上式一般在模型空间中计算，这样可以将切线空间的基作为顶点属性脱机保存。

37.2.1 Manual calculation of tangents and bitangents

• 注意从模型空间中变换 TBN 向量至世界空间时，N 采用法向量变换方式，但是若变换矩阵正交（无切变也无非统一缩放——二者均为仿射变换的一种），则使用原矩阵来变换法向量即可。这也有利于下文提及的将向量变换至 tangent space 的计算：若模型矩阵正交，则 TBN 向量变换至世界空间后仍彼此垂直，进而 TBN 矩阵也正交，便于计算 TBN 矩阵的逆。

37.2.2 Tangent space normal mapping

•有两种方法计算光照，一种将法向量从 tangent space 转换到世界坐标系，另一种将其他向量转换到 tangent space，然后在同一个坐标系下进行光照计算；虽然第二种方法看似有很多矩阵操作，但是这些计算可放在 vertex shader 中进行，vertex shader 的运行次数大大小于 fragment shader。因此这种方法效率更高，也是一种优化方式。实际上这种优化的原理就是利用顶点着色器中少量运算和到像素着色器之间的插值计算来代替完全在像素着色器中进行多次复杂的运算。当然是否可以这么优化需要数学证明。

• 注意使用 TBN 矩阵变换时有一些小技巧，无论变换点或向量，使用 3 维矩阵时都只考虑旋转不考虑平移。而光照计算中一般只考虑视线和光线的方向，也就是向量，因此不必使用

TBN 的 4 维矩阵进行变换，除非需要使用真正的点坐标。

37.3 Complex objects

• 使用第三方 API 来计算和加载法向贴图时需要注意很多问题，最好自己研究写代码。

37.4 One last thing

• 结合上面各节的讨论可知，切线空间向量经常会被平均化取值存于顶点属性，而这可能会导致 TBN 向量彼此不垂直。为了能够保证 TBN 矩阵正交进而节约矩阵求逆运算，可对 TBN 向量使用 Gram-Schimidt process 过程来重构正交 TBN 向量。该过程的数学运算较简单（p391）。理论上来说就是保持法向量不变，重新调整 T，然后求 B。

• 这个过程又会催生一个问题，切线空间向量是顶点属性，如果在像素着色器中使用 TBN 向量将会是多个顶点属性插值的结果——又将会改变 TBN 的正交性。而这再次证明在顶点着色器中使用 TBN 变换的优势。

37.5 Additional resources

p391

38. Parallax Mapping

•height map: 每个 texel 包含高度信息的纹理，称为 height map

•displacement mapping：根据纹理储存的数据对顶点进行位移操作（不是真正的位移，是假想）的 mapping，parallax mapping 属于其中之一。具体原理大致为：根据视线方向和片段位置两个因素，结合高度贴图计算出取样纹理坐标偏差，制造出网格表面有高度差的错觉。一般的网格纹理坐标只和片段位置有关，因此单纯视线方向变化不会导致纹理取样不同（只是光照计算不同）。

• 视差贴图用到了切线空间，其计算原理包括使用切线空间计算都是一种近似和幻象。

38.1 Parallax mapping

• Parallax mapping 的计算公式参数选取较复杂，而且看似是以参数调控的虚拟视觉。一句话概括，parallax mapping 和真正的高度差效果是不一样的，其所使用的近似方法也过于粗糙以至于像是在捏造。因此有必要使用后面小节中的技术改进。

• 视差贴图技术按照不同的参数选取也有不同的技术细分，如 Parallax mapping with offset limiting，p398

38.2 Steep Parallax Mapping

•此技术顾名思义用于改进视差贴图对于陡峭的高度变化的失真。原理就是使用连续多个纹理采样偏差来逼近得出近似的真正片段位置。

• 可以加入一种优化机制，在视线接近垂直于表面时，降低采样数量，反之增高。具体做法使用视线和表面法线的夹角作为比例插值最低和最高采样数量即可。【优化】

• Steep parallax mapping 原理上有所进步，但是由于仍然只是离散近似，只是将一次采样误差变为多次采样的误差，因此仍然有锯齿现象。有两种进一步改进方案，Relif Parallax mapping 和 POM，前者是最精确的方法但是性能差，后者是最平衡的方案。

38.3 Parallax Occlusion Mapping，POM

• 改进版 SPM，用最接近真实采样点的临近两个值插值获得近似值。

• PM 技术其实就是一种 trade-off，用少量顶点和微小性能代价来换取相反情况下的结果。这样做自然不是完美的，因此 PM 一般只用于地板和墙壁的贴图。因为贴图边缘处由于采样超出纹理边界会产生失真，而地板和墙壁的边缘处往往不受注意，另外地板和墙壁的视线一般都是接近垂直平面的，这也会少暴露 PM 的不足。

38.4 Additional resources

p405

39. HDR

•HDR(high dynamic range): 一般的显示器使用的是 LDR，即所有的光照颜色参数都在[0,1]的范围内，HDR 允许光照和颜色参数超过这个范围，在最终显示的时候再通过算法将参数转化为 LDR 范围进行显示。

•tone mapping：将 HDR 参数转化回 LDR 范围内的过程，这个过程涉及到很多不同的算法，不同的算法侧重于或明或暗区域细节的显示。如果直接将 HDR 参数线性转化为 LDR，那么较亮的区域会成为主导，如果不使用 HDR，那么较亮的区域将完全成为一片白色，失去细节。

39.1 Floating point framebuffers

•对一般的 framebuffer，被放入的颜色数据自动会被限制在[0,1]，想要扩大数据范围必须改变存储格式，使用 GL_RGB16F, GL_RGBA16F, GL_RGB32F, GL_RGBA32F 来替换一般的 GL_RGB 格式；一般的颜色分量只占用 8bit，floating point framebuffer 占用 16 或 32bit

39.2 Tone mapping

•Reinhard tone mapping:简单的算法，不使用曝光度，均匀的将 HDR 参数转化

vec3 mapped = hdrColor / (hdrColor + vec3(1.0))

•使用曝光度的简单算法

vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure)

39.2.1 More HDR

automatic exposure adjustment or eye adaptation: 用来模拟人眼效果，根据前一帧的颜色亮度来决定后一帧的曝光度

39.3 Additional resources

•Does HDR rendering have any benefits if bloom won't be applied?: a stackexchange question that features a great lengthy answer describing some of the benefits of HDR rendering.
http://gamedev.stackexchange.com/questions/62836/does-hdr-rendering-have-any-benefits-if-bloom-wont-be-applied

•What is tone mapping? How does it relate to HDR?: another interesting answer with great reference images to explain tone mapping.
http://photo.stackexchange.com/questions/7630/what-is-tone-mapping-how-does-it-relate-to-hdr

40. Bloom

•Bloom: 光晕效果，是一种后处理效果，含义顾名思义。bloom 技术和 HDR 技术并用效果更佳，而二者均可独立存在，因此是两个独立的技术。bloom 技术实现要领很简单，但是复杂在如何正确模糊处理被提取的高亮区域。

40.1 Extracting bright color

• MRT (Multiple Render Targets): 多重渲染输出目标，即使用一个片段着色器，在一次渲染过程中产生两个以上的颜色缓存信息。必要的条件是，作为被输出对象的帧缓存必须具有两个以上的颜色缓存附属。

方法：

①创建多个颜色缓存（纹理），分别附属于同一帧缓存的不同颜色缓存附属目标，即 GL_COLOR_ATTACHMENT0、GL_COLOR_ATTACHMENT1 等

②照常配置各个颜色缓存对象

③告知 OpenGL，渲染多个颜色缓存，否则只对帧缓存的第一个颜色缓存进行输出

GLuint attachments[2] = { GL_COLOR_ATTACHMENT0, GL_COLOR_ATTACHMENT1 };

glDrawBuffers(2, attachments);

④在 fragment shader 中，布置多个位置进行输出颜色

```
#version 330 core
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;
```

⑤对 FragColor 进行正常输出，在 BrightColor 中输出亮度（可以对片段颜色灰度化处理获得亮度值）高于一个阈值（使用 HDR 可以将该值设为 1.0）的片段，即相当于将原图中较亮的区域提取出来形成一幅图，存储在 BrightColor 中。

⑥对亮区提取图进行 blur 处理，然后和原图合并，即可形成光晕效果

40.2 Gaussian blur

- 高斯模糊化处理是按照一个高斯曲面进行权重取值的模糊滤镜处理方式，如果按照标准定义，那么对于一个片段将会出现 n^2 次采样。为了简化，先对所有片段进行横向一维采样处理，再对结果中所有片段进行纵向一维采样处理，如此综合来看对于每个片段只进行 2n 次采样。这样相当于将原来标准定义中的处理方式分成两步处理方式。可以多次重复这个 2 段采样处理的过程，即多次模糊化处理同一个图片，以达到理想效果。
- ping-pong framebuffers：即一对 framebuffers，用来相互进行图像输出，这种结构用于处理 2 段采样，或多次 2 段采样非常适合。一个 framebuffer 用于存储横向处理后的图像，另一个存储纵向处理后的图像。

40.3 Blending both textures

- 简单来说，bloom 效果取决于采样数量，模糊滤镜加载次数，或者是对高斯曲线的取值改进

40.4 Additionalresources

Efficient Gaussian Blur with linear sampling: descirbes the Gaussian blur very well and how to improve its performance using OpenGL's bilinear texture sampling.
http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/

Bloom Post Process Effect: article from Epic Games about improving the bloom effect by combining multiple Gaussian curves for its weights.
https://docs.unrealengine.com/udk/Three/Bloom.html

How to do good bloom for HDR rendering: Article from Kalogirou that describes how to improve the bloom effect using a better Gaussian blur method.
http://kalogirou.net/2006/05/20/how-to-do-good-bloom-for-hdr-rendering/

41. Deferred Shading

- forward rendering/forward shading: 渲染一个场景的最直接方式，对场景中每个物体分别进行渲染，效率较低，对于有大量被遮挡物体的场景，有大量光源和物体的场景来说浪费资源
- deferred shading/deferred rendering: 另外一种渲染方式，分两步；第一步 geometry pass，渲染一次场景，不进行 lighting 计算，将所有的几何信息（position，normal，diffuse map/albedo，specular）分别储存在一系列纹理中（统一称为 G-buffer）。另一层面在实质上进行了预深度检测。第二步 lighitng pass，再进行光照计算，类似于后处理。
- multiple render targets，MRT 技术可以让生成 G-buffer 多个纹理的过程一步同时完成。

41.1 The G-buffer

- 渲染一个片段的所需信息：片段坐标（插值）、漫反射贴图（albedo），法线贴图，（镜面）光照贴图，以及独立于单个片段的——所有光源位置和颜色信息，视点位置信息。前四个信息必须存于贴图纹理中，后面两个信息使用全局值存储即可。另外不一定 4 个信息需要 4

个纹理，根据不同信息所需要的数据种类和空间，两个或更多种信息可以存于同一个纹理的颜色管道中，而颜色管道的尺寸显然也可以自由选取。

41.2 The deferred lighting pass

略

41.3 Combing deferred rendering with forward rendering

• 延迟渲染方式需要解决一些问题，比如同一条视线上的片段只保留一个，因而不支持颜色混合（半透明场景）；比如渲染颜色和光照时不以物体为单位，因而不支持多种着色器或光照模型共用；以及 MSAA 无法使用等。但是每个问题都有对应的独特解法以供延迟渲染使用。

• 对于需要使用正向渲染的物体或材质（着色器）可以和其他使用延迟渲染的物体分离，在 geometry pass 结束时进行正向渲染或 blending，再正常进行 lighting pass。

41.4 A larger number of lights

• 大量的光源计算即便在延迟渲染中使用也仍然是个负担（虽然少计算很多不必要的片段，但是仍然需要针对每个可见片段进行逐光源的计算），而之所以可以实现是因为延迟渲染可以将一种叫做 light volume 的技术发挥得当。

• light volume 技术的理念就是根据光随着传播距离衰减的特性，计算出假想中的光源的最大传播距离（理论上仍然是无限传播的，即光强不可能为 0），这样形似将光源和光线视为一个有体积的物体。

41.4.1 Calculating a light’s volume or radius

略，用衰减公式求解最大半径

41.4.2 How we really use light volumes

•将光源和光线影响到的空间渲染成一个球体，该球体在屏幕空间所影响到的像素就是该光源在真实空间内影响到的区域（当然在视线深度上还有待推敲），因此可以将渲染球体的帧作为模板缓冲来渲染物体。最后将每个光源渲染出的场景帧混合。

• deferred lighting 和 tile-based deferred shading：两种更加高级的延迟渲染技术，可以使用 MSAA

41.5 Deferred rendering vs forward rendering

• 正向渲染的大部分所有技术都可以移植至延迟渲染技术替代。但是后者在简单、少量光照的场景下并不会突出其性能优势。

41.6 Additional resources

p439

42. SSAO

•SSAO 即 screen-space ambient occlusion，屏幕空间环境光遮蔽。是 AO 技术的一个简化版，便于在实时条件下实现 AO，否则需要在真实几何空间进行大量计算。

• SSAO 因为是在屏幕空间进行，因此也是基于片段周围采样的技术，但是采样是 3 维采样而不是纹理上的 2 维采样。在片段周围以一定 3 维范围对于深度缓冲采样，“被遮挡”的采样点（基于深度比较）的总比例决定了当前片段的环境光遮蔽程度。

42.1 Sample buffers

•由于是在屏幕空间进行采样和计算，SSAO 非常适合和 deferred shading 延迟渲染共同使用。

• AO 算是延迟渲染的 geometry pass 和 lighting pass 之间的一个中间渲染步骤，它可以生成所谓的 AO 纹理贴图，实际上就是每个片段所对应的环境光强度系数。

• 如果使用深度缓冲，其实可以在 fragment shader 中反求 position 信息，进而不必再花费内存存储位置缓冲。

42.2 Normal-oriented hemisphere

• 为了让结果更加贴近真实，片段周围的采样范围使用法线方向的半球体。而且为了以最少的采样点换取最好的效果，要将采样体按照一定规律随机旋转。这样可以消除带状光照，但是会产生均匀分布的光照噪点。后者可以使用 blur 来美化。

• 在切线空间生成半球取样核的逻辑 p445

42.3 Random kernel rotations

• 为了尽量减少所有能在着色器中进行的计算，将随机位向提前（脱机）生成为纹理数据存储以便后续着色器中直接调用，而不是每次现场调用随机算法。

42.4 The SSAO shader

• 文中记述的方法有一个小问题，提前存储的随机位向是以切线空间为基的，这样比较容易定义半球体的位向。然而使用 TBN 矩阵时，三个分量必须以视点空间为基。原文中直接使用法向量（应为视点空间）和随机位向生成 TBN 向量，这样其实并不影响随机化的结果，但是让特意以切线空间为基选取随机位向的行为变得多余了。

•注意比较深度不是拿深度缓冲和片段深度比较，而是拿 3 维采样点的真实深度和该采样点所在处的深度缓冲值比较，即考察它是否被遮挡。但是要加入 range check 机制，避免过远的面产生环境遮挡效果。p450

42.5 Ambient occlusion blur

模糊处理，略 p451

42.6 Applying ambient occlusion

• SSAO 是高度参数可自定化的技术

42.7 Additional resources

p456

43. Text Rendering

•OpenGL 中无法直接实现文字文本的渲染输出，除非将文字以 mesh 的方式绘制，或者以纹理的方式显示

43.1 Classical text rendering: bitmap fonts

•bitmap font: 经典的文本输出纹理；将某一字体的所有字符按照一定规则排列在一个大的纹理上，这个纹理就是 bitmap font，而每个字符称为 glyph，显然每个字符有特定的纹理坐标区域与之对应以便提取；之后将需要的 glyph 纹理渲染在一系列 2D 区域上。

优缺点：

因为是 bitmap font，所以是被预光栅化的，因此使用起来效率更高

但是放大对于这种格式会造成锯齿，所以一般都只能在固定大小固定分辨率的情况下使用，一些尺寸本身比较大的字符往往无法使用这种格式

43.2 Modern text rendering: FreeType

•TrueType Font: 一种相对于 bitmap font 更加先进的储存字体字符的方式，其所存储的字符不是以像素方式定义的或者其他不可缩放的方式，而是使用数学公式定义，因此字符的光栅化纹理可以根据需求的尺寸进行调整，而不降低质量出现锯齿

①FreeType 是一个 SDK，用来加载字体，将字体生成成位图，以及提供相关的操作，它可以加载 TrueType 字体，应用很广

②首先使用 freetype 加载某个字体的 ttf 文件，加载后储存在 face 对象中

```
FT_Library ft;
if (FT_Init_FreeType(&ft))
    std::cout << "ERROR::FREETYPE: Could not init FreeType Library" << std::endl;
```

```
FT_Face face;
if (FT_New_Face(ft, "fonts/arial.ttf", 0, &face))
    std::cout << "ERROR::FREETYPE: Failed to load font" << std::endl;
```

③设置所需字体大小 font size，字体大小（宏观概念）和后面的每个字符的 bitmap 尺寸不是一个概念，但是有影响

```
FT_Set_Pixel_Sizes(face, 0, 48);
```

设置字体宽度和高度（试验结果显示单位是像素），将宽度设为 0 使程序根据高度自行决定

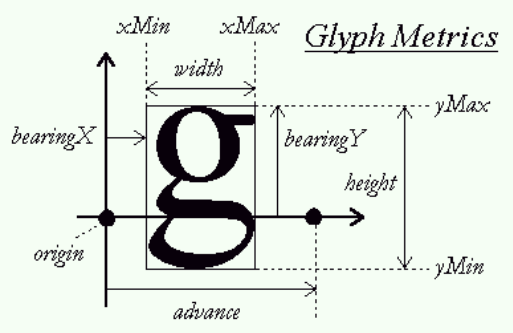
④加载后可以针对某一字符进行操作，即对某一字符进行生成 bitmap 的操作

```
if (FT_Load_Char(face, 'X', FT_LOAD_RENDER))
    std::cout << "ERROR::FREETYTPE: Failed to load Glyph" << std::endl;
```

FT_LOAD_RENDER 标明生成一个 8bit 灰度位图纹理

'X'表示被操作字符，也可用无符号整型数字 ASCII 码表示

•使用 freetype 为每个字符生成的 bitmap 图像大小不一，只是为了足够显示该字符而已，因此随着 bitmap 还有一系列 metrics 生成，用来描述字符的位置和 bitmap 的尺寸关系等。下图中，紧贴字符 g 周围的长方形框就是 bitmap 纹理的大小，而原点 origin 和 advance 则代表连续输出字符时比较合适的基准线 baseline 位置以及间距。



metrics 提取方法如下（除了 advance 以 1/64 像素为单位，其他数据以 1 像素为单位）：

width: face->glyph->bitmap.width

height: face->glyph->bitmap.rows

bearing X: face->glyph->bitmap_left

bearing Y: face->glyph->bitmap_top

advance: face->glyph->advance.x

⑤将所有所需的字符的 bitmap 图像分别生成成纹理后，储存起来以备程序使用，因为 bitmap 图像是每个像素占用一个字节的格式，不是 OpenGL 标准默认的 4 字节或 4 字节倍数的格式，必须解除以下锁定

```
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
```

⑥最后清除资源

```
FT_Done_Face(face);
FT_Done_FreeType(ft);
```

43.2.1 Shaders

•使用这种方式显示字符，需要激活 blending，用来使字符纹理的非字符本体所占位置变透明

•因为所要在 2D 方块区域显示的纹理大小不一，位置不一，所以对于每个字符纹理的渲染都需要一组新的几何参数定义的 2D 区域，即新的 VBO 和 VAO，所以应使用 GL_DYNAMIC_DRAW，并且使用 glBufferSubData 来对缓存进行操作

43.2.2 Render line of text

43.3 Going further

•为了提高性能，可以将 freetype 和 bitmap font 方法混合，即通过 freetype 的 bitmap 输出将所有的字符的 bitmap 制作成一个 bitmap font，方便之后进行取样，这样可以省去大量的纹理读取和切换时间（因为只需要一个大纹理）

•freetype 虽然比 bitmap font 先进，先进在于可以根据所需的字体大小来生成纹理，但是生成后的纹理仍然是 bitmap 格式，如果对生成后的纹理进行放大仍然会看到锯齿，除非根据放大的倍数重新生成 bitmap 纹理。

•一种更加先进的技术叫做 signed distance fields，每个像素储存的不是像素颜色，而是到最近的 glyph 边界的距离，这种方式储存的纹理不会因为放大而产生锯齿，适合 3D 渲染。3D 渲染之所以需要应对这个问题，是因为视窗具有垂直于屏幕方向上的自由度，所以势必会出现对于文本的放大效果。

详见”Papers\SIGGRAPH2007_AlphaTestedMagnification”

44. 2D Game

45. Breakout

45.1 OpenGL Breakout

46. Setting Up

•将游戏的整体机能设为一个类

46.1 Utility

•shader 和 texture 的编译连接创建等设为单独类

46.2 Resource management

47. Rendering Sprites

•对于输出的顶点坐标（亦即输出显示的内容）如果在主程序中定义（VAO，VBO 等），会非常繁琐并且违背原则，应设置类型来处理。sprite 即 render-able image/texture objects we use in a 2D game.

•对于 2D 游戏，每一个物体单元都是一个平面加纹理的结构，这样的一个组合单元中纹理图像部分即一个 sprite

47.1 2D projection matrix

•对于 2D 游戏可以将视景体和屏幕坐标结合在一起，变得更加直观

47.2 Rendering sprites

47.2.1 Initialization

47.2.2 Rendering

47.3 Hello sprite

48. Levels

48.1 Within the game

48.1.1 The player paddle

49. Ball

50. Collision detection

•为了简化碰撞检测的算法复杂度，一般选取简单图形来近似原物体，虽然这样有时候会导致物体没有被碰撞，但是仍然检测为碰撞

50.1 AABB-AABB collisions

•AABB: axis-aligned bounding box，用一个边和坐标轴对齐的长方体来近似物体，碰撞检测很简单

•AABB-AABB 即两个物体都为 AABB 的情况下进行碰撞检测，只需要分别检验 x 和 y 方向的两个边界是否同时有重合即可

```
bool collisionX = one.Position.x + one.Size.x >= two.Position.x && two.Position.x +
two.Size.x >= one.Position.x;
bool collisionY = one.Position.y + one.Size.y >= two.Position.y && two.Position.y +
two.Size.y >= one.Position.y;
return collisionX && collisionY;
```

50.2 AABB - Circle collision detection

•圆形和 AABB 的碰撞检测算法(p499)

51. Collision resolution

51.0.1 Collision repositioning

51.0.2 Collision direction

51.0.3 AABB-Circle collision resolution

自行解决算法

51.1 Player - ball collisions

51.1.1 Sticky paddle

•如果 paddle 运动过速，可能导致计算碰撞检测时，球的中心处于板子的内部，因为没有引入相应的算法，所以，圆球可能会持续在板子内部运动

51.1.2 The bottom edge

51.2 A few notes

•物理和碰撞系统可以说是游戏引擎最复杂的部分之一，比较高阶的万用 Collision scheme 比如 ***separating axis theorem***

•Box2D is a perfect 2D physics library for implementing physics and collision detection in your applications.

52. Particles

•粒子系统，用来模拟火焰烟雾等效果；一般是由很多的粒子和一个粒子发生器（particle emitter or particle generator）两个结构组成。

53. Postprocessing

53.0.1 Shake it

54. Powerups

54.0.1 Spawning Powerups

…

54.0.2 Activating Powerups

…

54.0.3 Updating PowerUps

…

55. Audio

•3D audio：具有 3D 空间效果的音效，即可以通过该声音来区分声音来源的远近与方向

55.1 Irrklang

•Irrklang 是一个高级 2D 或 3D 多平台（windows，mac os x，linux）声效引擎和声效库，可以播放 wav，MP3，ogg，flac 等格式文件。

```
#include <irrklang/irrKlang.h>
using namespace irrklang;
ISoundEngine *SoundEngine = createIrrKlangDevice();
void Game::Init()
{
[...]
SoundEngine->play2D("audio/breakout.mp3", GL_TRUE);
}
```

55.1.2 Adding sounds

•More irrKlang knowledge

http://www.ambiera.com/irrklang/tutorials.html

56. Render text

56.1 Player lives

56.2 Level selection

•对于按键控制的处理，要注意，人按键的速度相对于帧处理速度非常慢，无论是按击还是按住在计算机看来都是按住。因此对于不同的操作，要设计不同的按键处理方式，

56.3 Winning

57. Final thoughts

57.1 Optimizations

①sprite sheet/texture atlas: 可以将所有用到的 texture 组成一个大 texture，像 bitmap font 一样，这样可以提高纹理读取的效率，减少切换纹理的性能消耗

②instanced rendering: 因为大部分的 sprite 都是由相同的顶点组成，只是模型变换矩阵不同，可以使用 instanced renderer 进行一批次输出，减少对 draw 命令的调用，可以增加每一帧能够渲染的 sprite 总数

③triangle strips: 可以输出 TRIANGLE_STRIP 代替三角形，使用更少的顶点，这样可以节约传送给 GPU 的很大一部分数据空间

④space partitioning algorithms: 使用空间细分算法，如 BSP，Octrees 或 k-d trees，可以减少对碰撞检测的计算数量，即只计算和被测物体处于同一空间的其他物体

⑤minimize state changes: 状态切换对于 openGL 是一个非常损耗性能的操作，比如绑定纹理，切换着色器，切换帧缓存等，因此最大程度减少这样的切换是一个通用原则。

•比如可以将渲染操作进行排序，先渲染所有使用着色器 1 的物体，再渲染所有使用着色器

2 的物体，以此类推可以推广到纹理和帧缓存。

- 使用一个自定义的 state manager 管理和存储当前的 state，只有在必须更改 state 时才调用 openGL 命令，这样可以避免不必要的 state change。（可能是有时访问查询状态时也会造成性能损耗？）

57.2 Get creative

看 obj 格式

解决 nanosuit 问题

Omnidirectional Shadow Maps and Cascaded Shadow Maps).

PBR online

记录 map 和 vector 成员函数　　at()