Games101

2023-02-28 11:45

Table of Contents

Lecture1: Overview

- 如何判断好: 画面亮就是好(全局光照)
- 内容: 光栅化 / 几何 / 光线追踪 / 动画模拟

光栅化:

- 将几何投影, 实时(30fps)

几何:

- 如何表示物体, 如何变化

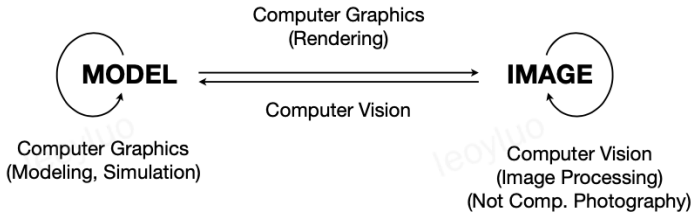
光线追踪:

- 慢, 但是真实, 需要离线
- 实时光线追踪

动画模拟:

OpenGL/DirectX/Vulcan: 只是图形学api, 在此不提

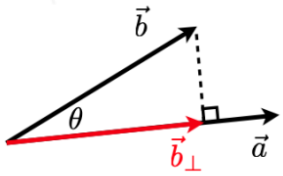
计算机视觉 vs 图形学: 界限越来越模糊, 一个是推理理解猜测, 另一个真实建模



==========

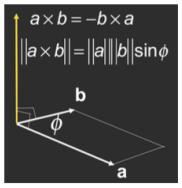
Lecture2: linear algebra

向量点乘



- dot product: a · b = ||a|| ||b|| cos(theta)
- in Cartesian Coord: a · b = (xa ya) (xb yb) = xaxb + yayb
- 用于找向量夹角, 向量投影

向量叉乘



- cross product: a x b = ||a|| ||b|| sin(theta)
- 右手法则: 从a到b大拇指方向. (opengl等api可能用左手坐标系)
- a x b = – b x a; a x a = 0

Cross Product: Cartesian Formula?

$$\vec{a} \times \vec{b} = \begin{pmatrix} y_a z_b - y_b z_a \\ z_a x_b - x_a z_b \\ x_a y_b - y_a x_b \end{pmatrix}$$

- Later in this lecture

$$\vec{a} \times \vec{b} = A^* b = \begin{pmatrix} 0 & -z_a & y_a \\ z_a & 0 & -x_a \\ -y_a & x_a & 0 \end{pmatrix} \begin{pmatrix} x_b \\ y_b \\ z_b \end{pmatrix}$$

dual matrix of vector a

- 用于求解两个向量的顺序/ 求解内外(点是否在多边形内部)

构建坐标系

Orthonormal Coordinate Frames

- Any set of 3 vectors (in 3D) that

$$\|\vec{u}\| = \|\vec{v}\| = \|\vec{w}\| = 1$$

$$\vec{u} \cdot \vec{v} = \vec{v} \cdot \vec{w} = \vec{u} \cdot \vec{w} = 0$$

$$\vec{w} = \vec{u} \times \vec{v} \quad \text{(right-handed)}$$

$$\vec{p} = \underbrace{(\vec{p} \cdot \vec{u})\vec{u} + (\vec{p} \cdot \vec{v})\vec{v} + (\vec{p} \cdot \vec{w})\vec{w}}_{\text{(projection)}}$$

- 单位向量互相垂直, 分解p向量

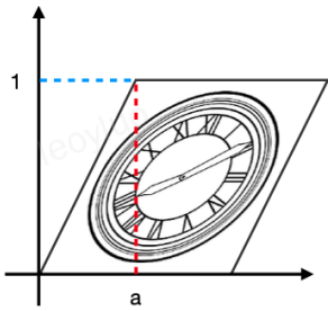
==========

Lecture3: Transformation

Scale

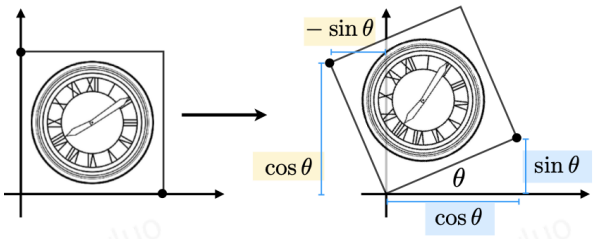
$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Shear Matrix



$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} 1 & a \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotate



$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

- 线性变换: x'=Mx

Homogeneous Coord:

- Why needs it?

  Translation cannot be represented in matrix form

  $$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

  (So, translation is NOT linear transform!)
- \- >

- How to represent?

  Add a third coordinate (*w-coordinate*)

  - 2D point = (x, y, 1)ᵀ
  - 2D vector = (x, y, 0)ᵀ
- →

  In homogeneous coordinates,

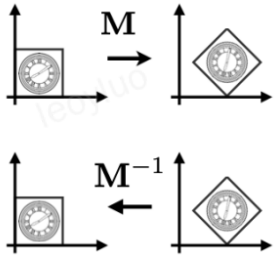
  $$\begin{pmatrix} x \\ y \\ w \end{pmatrix} \text{ is the 2D point } \begin{pmatrix} x/w \\ y/w \\ 1 \end{pmatrix}, \quad w \neq 0$$
-
- point + point = mid_point in Homo Coord

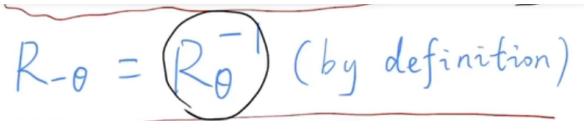
Affine Transformation

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

. = linear + translation

Inverse





正交矩阵

Compose

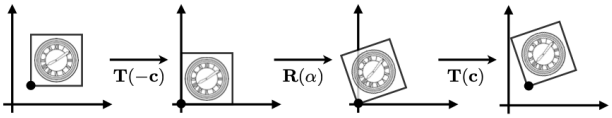
$$A_n(\dots A_2(A_1(\mathbf{x}))) = \underbrace{\mathbf{A}_n \cdots \mathbf{A}_2 \cdot \mathbf{A}_1}_{\text{Pre-multiply } n \text{ matrices to obtain a single matrix representing combined transform}} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

. Order Matters !

Rotate on given point

How to rotate around a given point c?

1. **Translate center to origin**
2. **Rotate**
3. **Translate back**



Matrix representation?

$$\mathbf{T}(\mathbf{c}) \cdot \mathbf{R}(\alpha) \cdot \mathbf{T}(-\mathbf{c})$$

3D transformation

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

==========

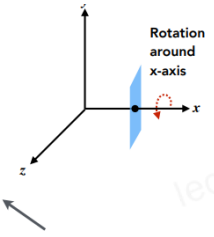
Lecture4: Transformation (cont)

Rotation around axis

$$\mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\mathbf{R}_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Anything strange about $\mathbf{R}_y$

y x z = x

x x y = z

z x x = y, so y's order is different (–a)

Euler Angles

Compose any 3D rotation from $\mathbf{R}_x, \mathbf{R}_y, \mathbf{R}_z$?

$$\mathbf{R}_{xyz}(\alpha, \beta, \gamma) = \mathbf{R}_x(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\gamma)$$

- **So-called *Euler angles***

Rodrigues' Rotation Formula

Rotation by angle α around axis n

$$\mathbf{R}(\mathbf{n},\alpha) = \cos(\alpha)\mathbf{I} + (1-\cos(\alpha))\mathbf{n}\mathbf{n}^T + \sin(\alpha)\underbrace{\begin{pmatrix} 0 & -n_z & n_y \\ n_z & 0 & -n_x \\ -n_y & n_x & 0 \end{pmatrix}}_{\mathbf{N}}$$

N是叉乘矩阵

推导可见 https://sites.cs.ucsb.edu/~lingqi/teaching/resources/GAMES101_Lecture_04_supp.pdf https://zhuanlan.zhihu.com/p/79061355

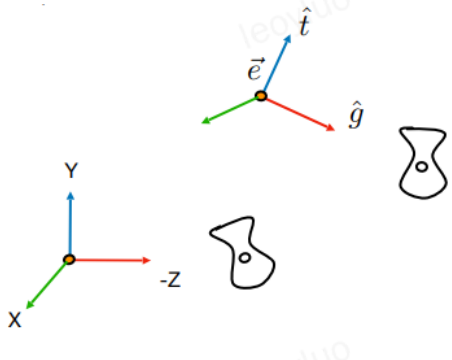
View/Camera Transformation

- M_{view} in math?
  - Let's write $M_{view} = R_{view}T_{view}$
  - Translate e to origin

$$T_{view} = \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

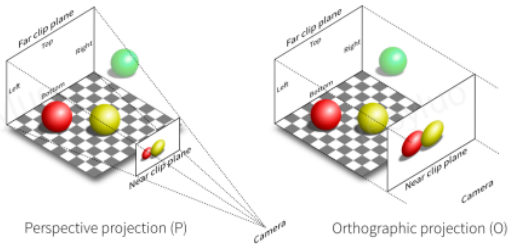
  - Rotate g to -Z, t to Y, (g x t) To X
  - Consider its inverse rotation: X to (g x t), Y to t, Z to -g

$$R_{view}^{-1} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & x_t & x_{-g} & 0 \\ y_{\hat{g}\times\hat{t}} & y_t & y_{-g} & 0 \\ z_{\hat{g}\times\hat{t}} & z_t & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{WHY?} \quad R_{view} = \begin{bmatrix} x_{\hat{g}\times\hat{t}} & y_{\hat{g}\times\hat{t}} & z_{\hat{g}\times\hat{t}} & 0 \\ x_t & y_t & z_t & 0 \\ x_{-g} & y_{-g} & z_{-g} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Projection

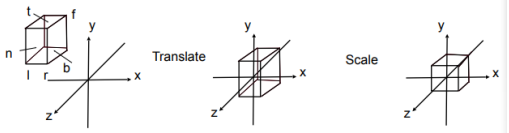
- Perspective projection vs. orthographic projection



Orthographic Projection

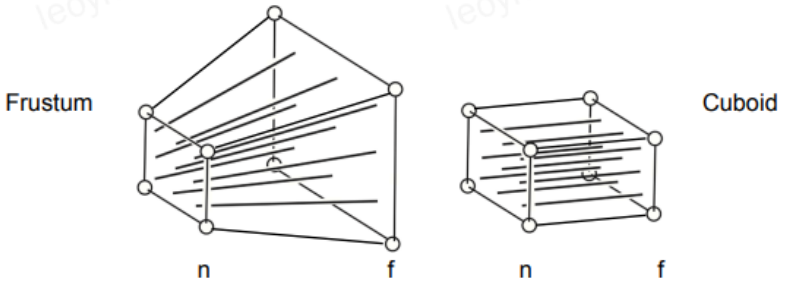
- Transformation matrix?
  - Translate (**center** to origin) first, then scale (length/width/height to **2**)

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



目的是为了将3d点转换到一个(–1, 1)三次方的cube当中

Perspective Projection



在这里插入图片描述

constrain：对于在近平面和远平面的点， z不会变化。 近平面上的点xyz都不变

对于点(x, y, z, 1), 变换之后得到(nx/z, ny/z, n+f – nf/z, 1), 有几个收获：

- 当z=n时， 新的点仍为(x, y, z, 1), 近的点没有变化
- 当z=f时， 新的点为(nx/f, ny/f, f, 1), 深度z不变， xy向内收， (0,0,f,1)仍在原处
- 设t = z'–z = n+f – nf/z – z = (z–n)(f–z)/z
  - 当z=n或f时， t不变， z‘=z深度不变
  - 当z>f或z<n时，t<0, z'<z. 当点在frustum外面时， 变化使得z缩小
  - 当n<z<f时， t>0, z'>z, 当点在内部时， 变化使得z变大

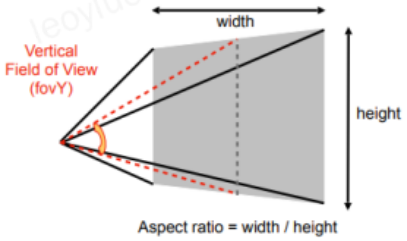
投影总结

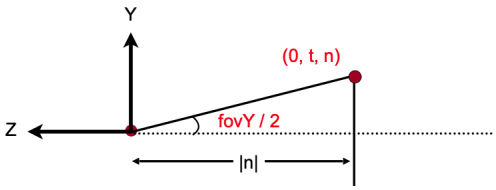
所以，总的来说，从世界坐标转换到屏幕坐标，其变换过程为：

1. MVP变换求投影空间坐标：$ProjPoint = WorldPoint * Model * View * Project$
2. 由投影空间坐标求NDC坐标：$NDCPoint(x, y, z, w) = ProjPoint(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$
3. 由NDC坐标转空间坐标：
   $ScreenPoint(x, y, z) = NDCPoint(\frac{width}{2} * x + \frac{width}{2} + left, \frac{height}{2} * y + \frac{height}{2} + top, z)$

==========

Lecture5: Rasterization 1 (Triangles)



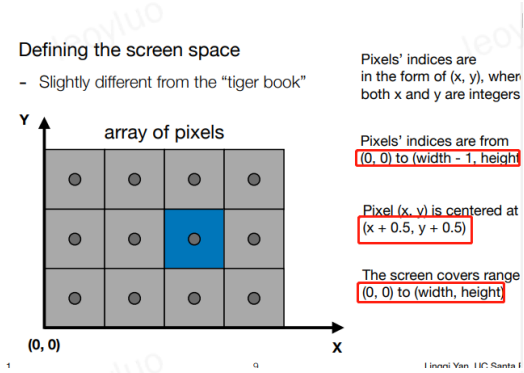


$$\tan\frac{fovY}{2} = \frac{t}{|n|}$$

$$aspect = \frac{r}{t}$$

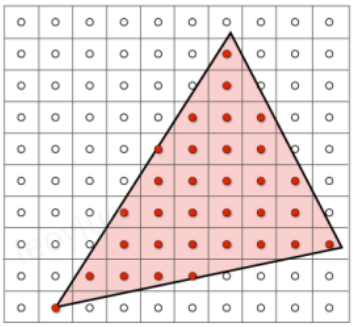
Rasterize： color on screen

屏幕空间



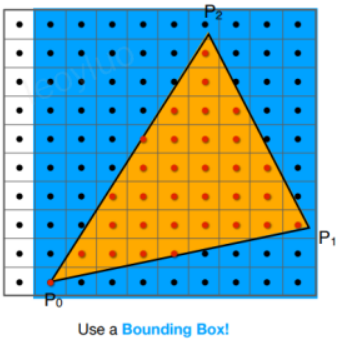
$$M_{viewport} = \begin{pmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

三角形的光珊化



判断每个像素中心是不是在三角形内： 使用叉积，判断pi–>o和pi→pi+1的叉积符号。 全正或者全负才是在里面的

优化： 只需要在bbox内的才需要被计算

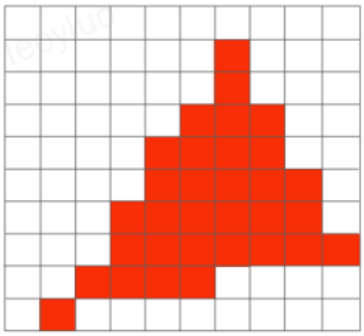


问题：锯齿、走样！因为像素的有一定的大小导致

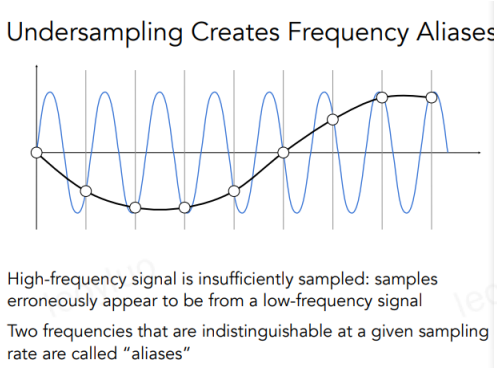
==========

Lecture6: Rasterization 2 (Antialiasing and Z-Buffering)

锯齿问题



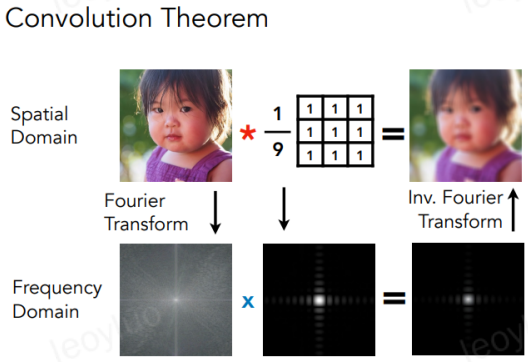
走样–定义



滤波：

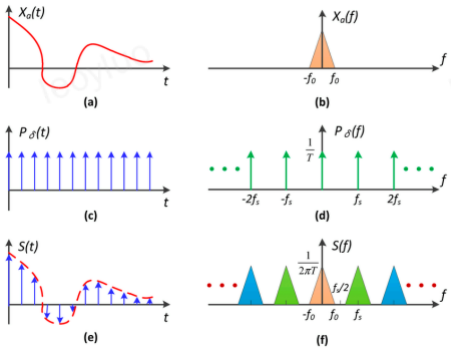
- 高通滤波 （high pass）：只留下**高频**的信息， 多是**edge**
- 低通滤波（low pass）：只留下**低频**的信息， 丢失细节，只剩下**模糊色块**

卷积与傅里叶变换



- 在图像上滤波（时域） = 频域 x 卷积核的频域
- 越大（小）的卷积核 = 越低频（高）的信息

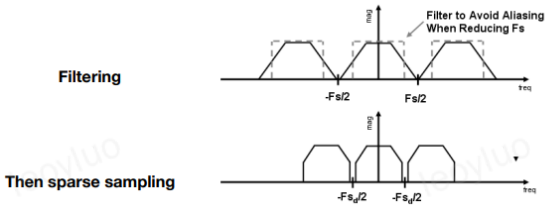
采样



- 采样就是重复频域图
- 当频谱的间隔太小时， 到时重叠，带来走样

Anti-aliasing

- 高频显示器采样（开销高

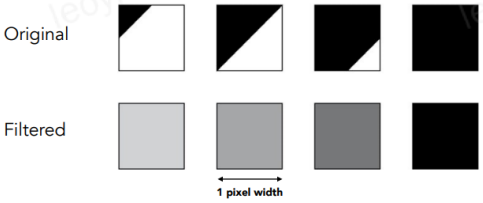


- 先模糊（去除高频），再采样避免重叠。 对于此处图形学而言， 就是边界色彩需要模糊加权

方法：用三角形覆盖像素面积作为filter结果

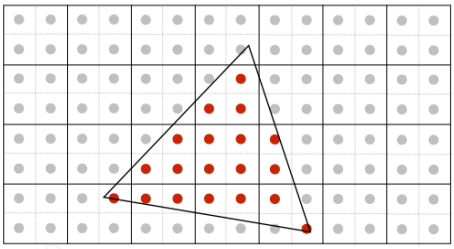
Antialiasing by Computing Average Pixel Value

In rasterizing one triangle, the average value inside a pixel area of f(x,y) = inside(triangle,x,y) is equal to the area of the pixel covered by the triangle.



Antialiasing By Supersampling (MSAA)

- 一种采样方法， 近似求出average area
  Take NxN samples in each pixel.

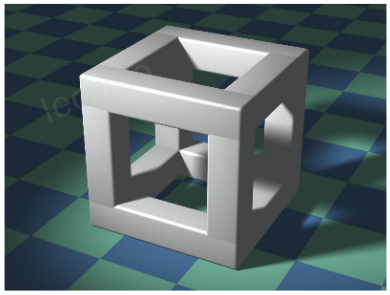


2x2 supersampling

- cons：增大了计算量

Z-Buffer

解决遮挡关系带来的影响



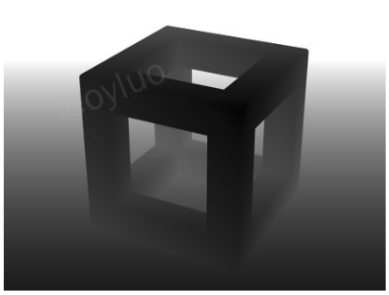


Image source: Dominic Alves, flickr.

Rendering Depth / Z buffer

Z-Buffer Algorithm

Initialize depth buffer to ∞

During rasterization:

```
for (each triangle T)
    for (each sample (x,y,z) in T)
        if (z < zbuffer[x,y])              // closest sample so far
            framebuffer[x,y] = rgb;        // update color
            zbuffer[x,y] = z;              // update depth
        else
            ;                  // do nothing, this sample is occluded
```

==========

Lecture7:Shading 1 (Illumination, Shading and Graphics Pipeline)

- 定义

Shading: Definition

- In Merriam-Webster Dictionary

  shad·ing, ['ʃeɪdɪŋ], noun
  The darkening or coloring of an illustration or diagram with parallel lines or a block of color.

- In this course

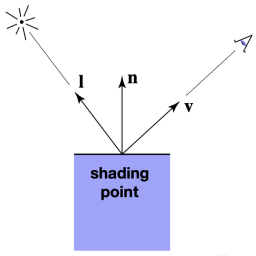
  The process of applying a material to an object.

- 输入 (what to RGB)

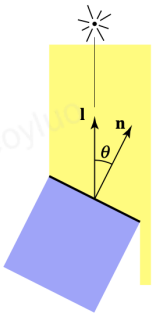
Compute light reflected toward camera at a specific shading point

Inputs:

- Viewer direction, v
- Surface normal, n
- Light direction, l (for each of many lights)
- Surface parameters (color, shininess, …)



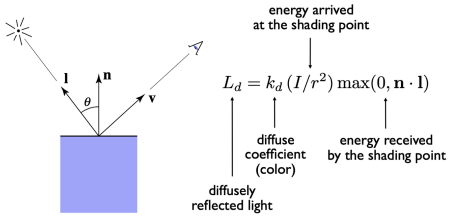
- 能量接收



In general, light per unit area is proportional to cos θ = l • n

Lambertian (Diffuse) Shading

Shading independent of view direction

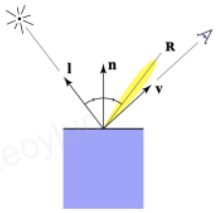


- 漫反射颜色建模
- 高光
  - 高光方向, R为镜面反射方向. 当v R接近的时候, 可以观测到高光

Specular Term (Blinn-Phong)

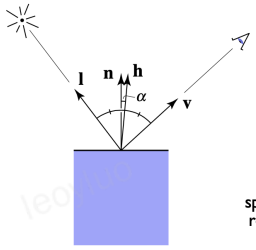
Intensity depends on view direction

- Bright near mirror reflection direction



V close to mirror direction ⇔ **half vector** near normal

- Measure "near" by dot product of unit vectors



$$\mathbf{h} = \text{bisector}(\mathbf{v}, \mathbf{l})$$
(半程向量)
$$= \frac{\mathbf{v}+\mathbf{l}}{\|\mathbf{v}+\mathbf{l}\|}$$

$$L_s = k_s\,(I/r^2)\max(0, \cos\alpha)^p$$
$$= k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$$

specularly reflected light

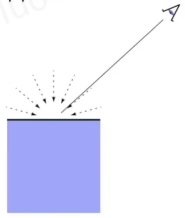
specular coefficient

等价于法向量和h是否接近, p控制高光大小

- 环境光

Shading that does not depend on anything

- Add constant color to account for disregarded illumination and fill in black shadows
- This is approximate / fake!



$$L_a = k_a\,I_a$$

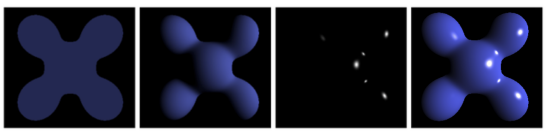
ambient coefficient

reflected ambient light

与观测位置无关, 常数颜色(大胆的假设)

- blinn–phong着色模型

Blinn-Phong Reflection Model



Ambient + Diffuse + Specular = Blinn-Phong Reflection

$$L = L_a + L_d + L_s$$
$$= k_a\,I_a + k_d\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{l}) + k_s\,(I/r^2)\max(0, \mathbf{n}\cdot\mathbf{h})^p$$

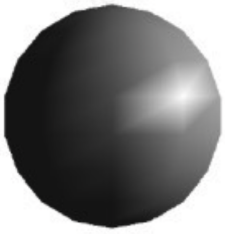
主要就是跟光照系数, 以及光照/观测方向相关

==========

Lecture8: Shading 2 (Illumination, Shading and Graphics Pipeline)

- Shading 频率

What caused the shading difference?


-
- (1) 每个三角形算一次normal　　(2) 每个三角形算3次　　(3) 每个三角形, 插值中间的normal, 每个像素都算一次

Flat shading　　Gouraud shading　　Phone shading

当面的数量足够复杂, 三种方式达到统一

- 法线计算

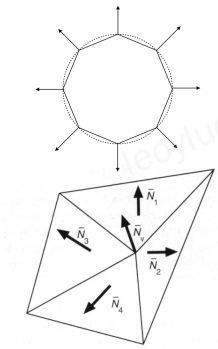
Best to get vertex normals from the underlying geometry

- e.g. consider a sphere

Otherwise have to infer vertex normals from triangle faces

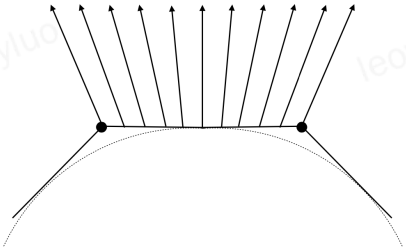
- Simple scheme: **average surrounding face normals**

$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$



逐顶点发现就是附近三角形中心发现的平均(简单平均, 加权平均等等)

Barycentric interpolation (introducing soon) of vertex normals

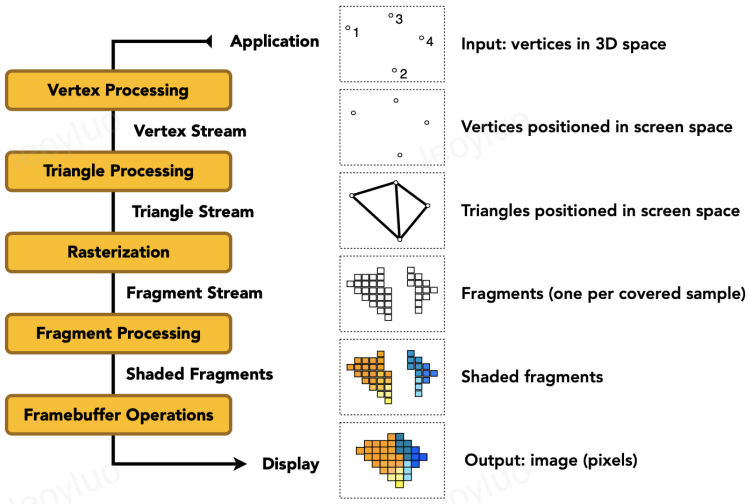


Don't forget to normalize the interpolated directions

逐像素计算, 要重心坐标

- 图形管线

Graphics Pipeline



Example GLSL fragment shader program

```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(–lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

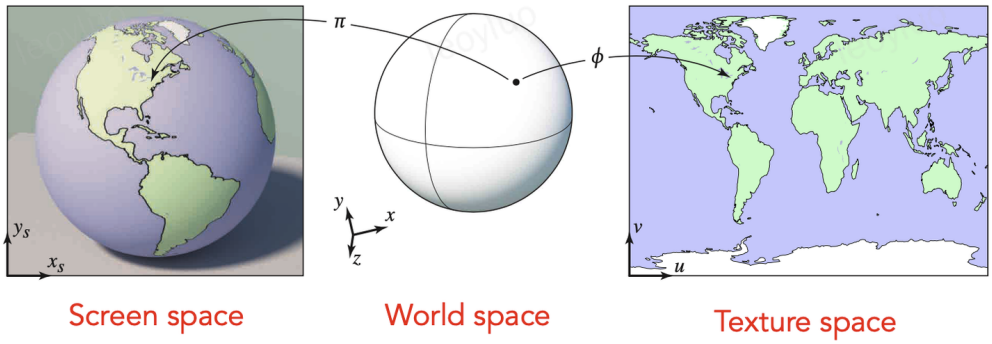
- Shader function executes once per fragment.
- Outputs color of surface at the current fragment's screen sample position.
- This shader performs a texture lookup to obtain the surface's material color at this point, then performs a diffuse lighting calculation.

简单的shader函数, 处理每个像素的phong漫反射着色器

==========

Lecture9:Shading 3 (Illumination, Shading and Graphics Pipeline)

- texture 纹理映射



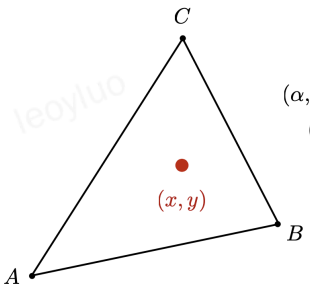
三维坐标的uv映射

纹理可以被映射多次(需要纹理设计的好, 否则有间隙) tiled

- **重心坐标**
  - 三角形内任意属性都可以插值(位置, 深度, 颜色, 法向量, 材质)
  - 投影后会变, 需要在三维间xyz插值

Barycentric Coordinates

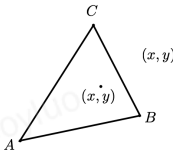
What's the barycentric coordinate of the centroid?



$$(\alpha,\beta,\gamma)=\left(\tfrac{1}{3},\tfrac{1}{3},\tfrac{1}{3}\right)$$
$$(x,y)=\tfrac{1}{3}A+\tfrac{1}{3}B+\tfrac{1}{3}C$$

等面积的三角形。

Barycentric Coordinates: Formulas



$$(x,y)=\alpha A+\beta B+\gamma C$$
$$\alpha+\beta+\gamma=1$$

$$\alpha=\frac{-(x-x_B)(y_C-y_B)+(y-y_B)(x_C-x_B)}{-(x_A-x_B)(y_C-y_B)+(y_A-y_B)(x_C-x_B)}$$
$$\beta=\frac{-(x-x_C)(y_A-y_C)+(y-y_C)(x_A-x_C)}{-(x_B-x_C)(y_A-y_C)+(y_B-y_C)(x_A-x_C)}$$
$$\gamma=1-\alpha-\beta$$

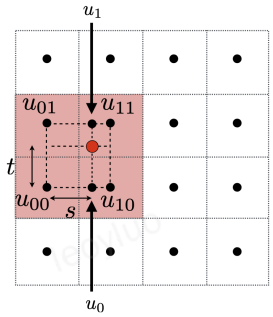
任意坐标插值

- **纹理应用**

(1) 纹理扩增

- 从低分辨率到高分辨率的变化 – 双线性插值

Bilinear Interpolation



Linear interpolation (1D)

$$\text{lerp}(x,v_0,v_1)=v_0+x(v_1-v_0)$$

Two helper lerps

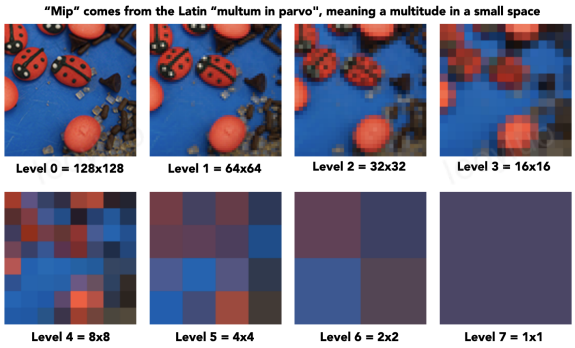
$$u_0=\text{lerp}(s,u_{00},u_{10})$$
$$u_1=\text{lerp}(s,u_{01},u_{11})$$

Final vertical lerp, to get result:

$$f(x,y)=\text{lerp}(t,u_0,u_1)$$

(2) 不同位置的像素对应同样的纹理，覆盖范围不一样

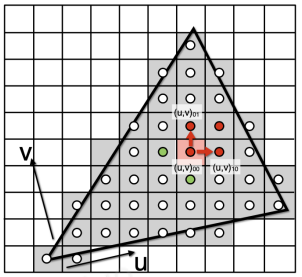
- mipmap (范围查询) – fast/approx/square

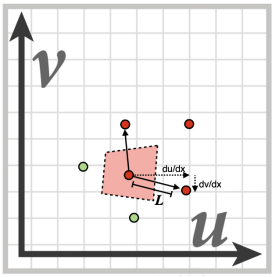


log(n)层，从最大重新生成，多了1/3的存储空间

mipmap level计算

Computing Mipmap Level D



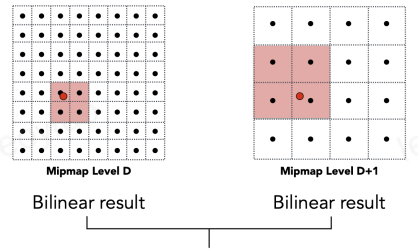


$$D=\log_2 L \qquad L=\max\left(\sqrt{\left(\frac{du}{dx}\right)^2+\left(\frac{dv}{dx}\right)^2},\sqrt{\left(\frac{du}{dy}\right)^2+\left(\frac{dv}{dy}\right)^2}\right)$$

对应纹理空间上的近似求解，在log2(L)上查询mipmap

mipmap两层之间插值

Trilinear Interpolation



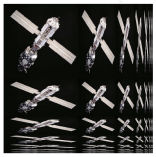
，能够让远近过渡区域颜色渐变

缺陷

- mipmap都是正方形区域, 不能解决长方形区域的变化
- 各向异性过滤

Anisotropic Filtering

Ripmaps and summed area tables
- Can look up axis-aligned rectangular zones
- Diagonal footprints still a problem



Wikipedia

EWA filtering
- Use multiple lookups
- Weighted average
- Mipmap hierarchy still helps
- Can handle irregular footprints

Greene & Heckbert '86

纹理究竟是什么

Many, Many Uses for Texturing

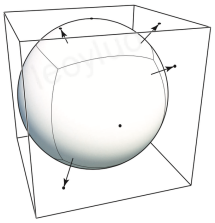
In modern GPUs, texture = memory + range query (filtering)
- General method to bring data to fragment calculations

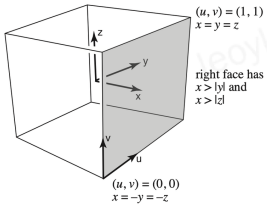
Many applications
- Environment lighting
- Store microgeometry
- Procedural textures
- Solid modeling
- Volume rendering
- …

一块内存, 方便快速查询, 可以表示很多东西

- **应用**

Cube Map



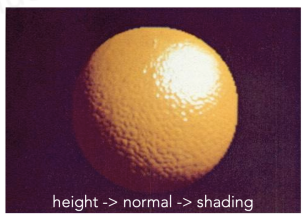


A vector maps to cube point along that direction.
The cube is textured with 6 square texture maps.

环境贴图

- Textures doesn't have to only represent colors
  - What if it stores the height / normal?
  - Bump / normal mapping
  - **Fake** the detailed geometry

Relative height to the underlying surface



凹凸贴图(切线扰动获取新法线)

==========

Lecture10: Geometry I (Representation)

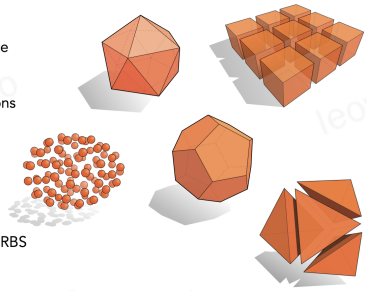
- 各种类型: 曲面, 布料, 水滴, 多层次场景, 毛发, 原子

表示方式

Many Ways to Represent Geometry

Implicit
- algebraic surface
- level sets
- distance functions
- ...

Explicit
- point cloud
- polygon mesh
- subdivision, NURBS
- ...



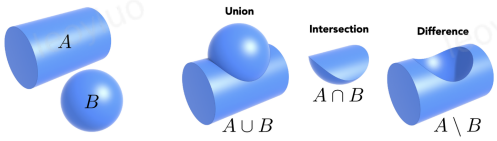
Each choice best suited to a different task/type of geometry

- 显式
  - 点云, mesh
  - parametric mapping
- 隐式
  - f(x, y, z)=0 满足表达式, 表示平面 – 很多时候比较难知道几何形状是什么样的
  - 容易判断某个点在内还是在外

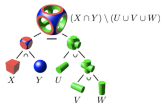
CSG – 基本几何的组合

Constructive Solid Geometry (Implicit)

Combine implicit geometry via Boolean operations



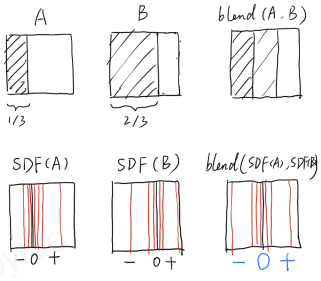
Boolean expressions:



sdf – 距离函数

Distance Functions (Implicit)

An Example: Blending (linear interp.) a moving boundary



边界融合

- mesh文件格式

The Wavefront Object File (.obj) Format

Commonly used in Graphics research

Just a text file that specifies vertices, normals, texture coordinates **and their connectivities**

```
# This is a comment

v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000

vt 0.748573 0.750412
vt 0.749279 0.501284
vt 0.999110 0.501077
vt 0.999455 0.750380
vt 0.250471 0.500702
vt 0.249682 0.749677
vt 0.001085 0.750380
vt 0.001517 0.499994
vt 0.499422 0.500239
vt 0.500149 0.750166
vt 0.748355 0.998230
vt 0.500193 0.998728
vt 0.498993 0.250415
vt 0.748953 0.250920
```

```
vn 0.000000 0.000000 -1.000000
vn -1.000000 -0.000000 -0.000000
vn -0.000000 -0.000000 1.000000
vn -0.000001 0.000000 1.000000
vn 1.000000 -0.000000 0.000000
vn 1.000000 0.000000 0.000001
vn 0.000000 1.000000 -0.000000
vn -0.000000 -1.000000 0.000000

f 5/1/1 1/2/1 4/3/1
f 5/1/1 4/3/1 8/4/1
f 3/5/2 7/6/2 8/7/2
f 3/5/2 8/7/2 4/8/2
f 2/9/3 6/10/3 3/5/3
f 6/10/4 7/6/4 3/5/4
f 1/2/5 5/1/5 2/9/5
f 5/1/6 6/10/6 2/9/6
f 5/1/7 8/11/7 6/10/7
f 8/11/7 7/12/7 6/10/7
f 1/2/8 2/9/8 3/13/8
f 1/2/8 3/13/8 4/14/8
```

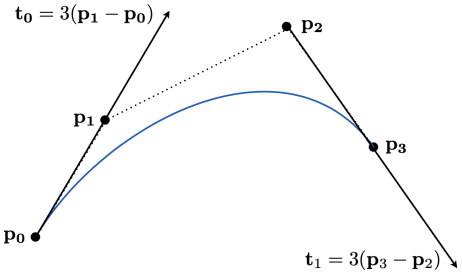
v: 顶点, vn: 顶点法线, vt:每个面的纹理坐标, f: 哪些顶点组成面片, v/vt/vn

==========

Lecture11: Geometry II (Curve and Surface)

- Bezier Curves

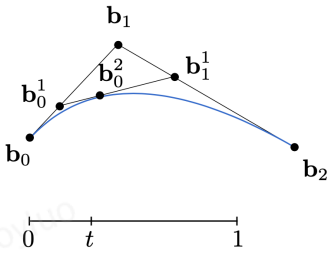
Defining Cubic Bézier Curve With Tangents



四点决定曲线, p0起, p3终结, 切线经过p1, p2

- de Casteljau 算法

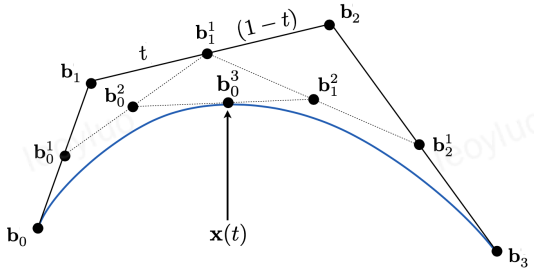
Run the same algorithm for every t in [0,1]



三角形为例, 两条边的t切割, 递归继续lerp

Four input points in total

Same recursive linear interpolations



四个点, 递归三次

n个点的代数形式

Bézier Curve – General Algebraic Formula

Bernstein form of a Bézier curve of order n:

$$\mathbf{b}^n(t) = \mathbf{b}_0^n(t) = \sum_{j=0}^{n} \mathbf{b}_j B_j^n(t)$$

- $\mathbf{b}^n(t)$: Bézier curve order n (vector polynomial of degree n)
- $B_j^n(t)$: Bernstein polynomial (scalar polynomial of degree n)
- $\mathbf{b}_j$: Bézier control points (vector in $\mathbb{R}^N$)

Bernstein polynomials:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

性质

Properties of Bézier Curves

Interpolates endpoints

- For cubic Bézier: $\mathbf{b}(0) = \mathbf{b}_0; \quad \mathbf{b}(1) = \mathbf{b}_3$

Tangent to end segments

- Cubic case: $\mathbf{b}'(0) = 3(\mathbf{b}_1 - \mathbf{b}_0); \quad \mathbf{b}'(1) = 3(\mathbf{b}_3 - \mathbf{b}_2)$

Affine transformation property

- Transform curve by transforming control points

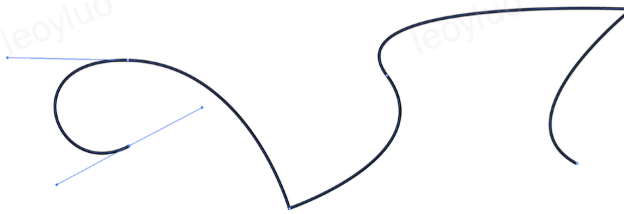
Convex hull property

- Curve is within convex hull of control points

过起始点 / 切线过p1, p2 / 几何变换只要变换顶点就可以 / 凸包性质

piecewise bezier曲线

Piecewise cubic Bézier the most common technique



Widely used (fonts, paths, Illustrator, Keynote, …)

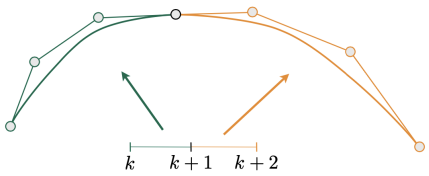
避免太高阶的计算, 每次只用四个点进行曲线计算, 并联合曲线(所以中间有急剧变化点)

Piecewise Bézier Curve – Continuity

Two Bézier curves

$$\mathbf{a} : [k, k+1] \to \mathbb{R}^N$$
$$\mathbf{b} : [k+1, k+2] \to \mathbb{R}^N$$

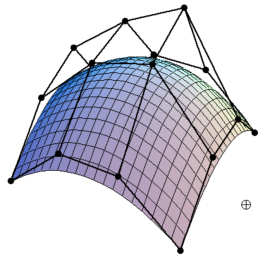
Assuming integer partitions here, can generalize



连续性: c0连续(交界点一致即可) / c1连续(左右切线连续且一样大)

贝塞尔曲面

Bicubic Bézier Surface Patch



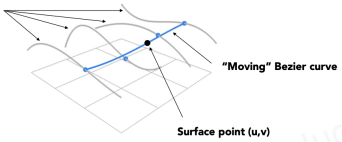
Bezier surface and 4 x 4 array of control points

Method: Separable 1D de Casteljau Algorithm

Goal: Evaluate surface position corresponding to (u,v)

(u,v)-separable application of de Casteljau algorithm

- Use de Casteljau to evaluate point u on each of the 4 Bezier curves in u. This gives 4 control points for the "moving" Bezier curve
- Use 1D de Casteljau to evaluate point v on the "moving" curve



每4个一列的点构建曲线, 再插值获取曲面

- 可以通过u, v获取面上任意一个xyz点

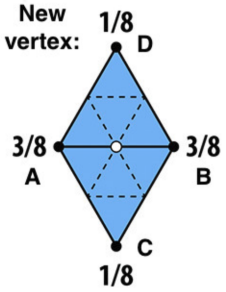
==========

Lecture12: Geometry III (Mesh Processing)

Loop subdivision

- 只能用于三角形面

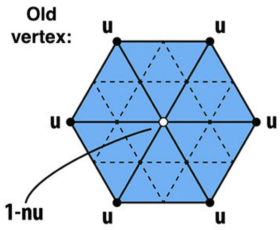
For new vertices:



Update to:
3/8 * (A + B) + 1/8 * (C + D)

Loop Subdivision — Update

For old vertices (e.g. degree 6 vertices here):



Update to:
(1 - n*u) * original_position +
u * neighbor_position_sum

n: vertex degree

u: 3/16 if n=3, 3/(8n) otherwise

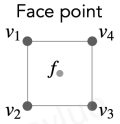
老顶点和新的顶点都要进行调整

Catmull–Clark Subdivision

- **引入奇异点, 消除非四边形面. 之后的细分再也不会有新的奇异点**

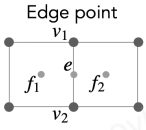
FYI: Catmull-Clark Vertex Update Rules (Quad Mesh)

Face point



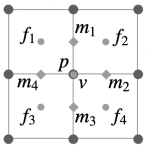
$$f = \frac{v_1+v_2+v_3+v_4}{4}$$

Edge point



$$e = \frac{v_1+v_2+f_1+f_2}{4}$$

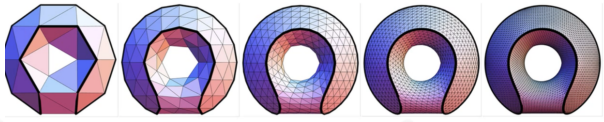
Vertex point



$$v = \frac{f_1+f_2+f_3+f_4+2(m_1+m_2+m_3+m_4)+4p}{16}$$

m midpoint of edge
p old "vertex point"

Loop with Sharp Creases



Catmull-Clark with Sharp Creases

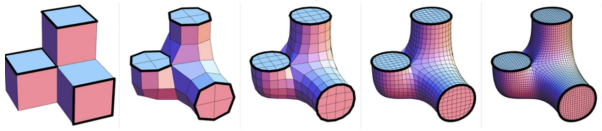


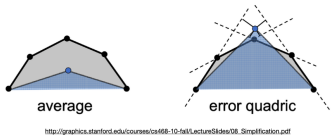
Figure from: Hakenberg et al. Volume Enclosed by Subdivision Surfaces with Sharp Creases

细分都是平滑化

Mesh简化

Quadric Error Metrics
（二次误差度量）

- How much geometric error is introduced by simplification?
- Not a good idea to perform local averaging of vertices
- Quadric error: new vertex should minimize its sum of square distance (L2 distance) to previously related triangle planes!



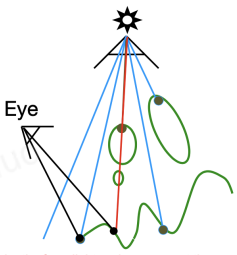
http://graphics.stanford.edu/courses/cs468-10-fall/LectureSlides/08_Simplification.pdf

通过最小堆逐渐优化 大的误差

Shadow Mapping

Pass 2B: Project to light

- Project visible points in eye view back to light source

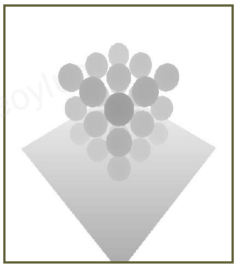


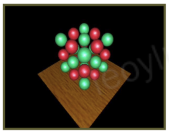
(Reprojected) depths from light and eye are not the same. BLOCKED!!

阴影应该是不能同时被观测点和光源看到的区域

Visualizing Shadow Mapping

- The depth buffer from the light's point-of-view





FYI: from the light's point-of-view again

光源到场景的深度图, 观测点深度对比判断阴影

==========

Lecture13: Ray Tracing

光线的假设

Light Rays

Three ideas about light rays

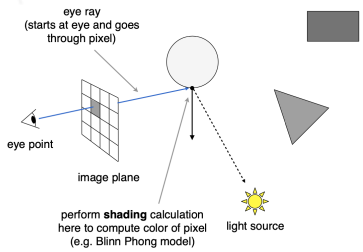
1. Light travels in straight lines (though this is wrong)
2. Light rays do not "collide" with each other if they cross (though this is still wrong)
3. Light rays travel from the light sources to the eye (but the physics is invariant under path reversal - reciprocity).

"And if you gaze long into an abyss, the abyss also gazes into you." — Friedrich Wilhelm Nietzsche (translated)

走直线, 不碰撞, 从光源到眼睛

Ray Casting - Shading Pixels (Local Only)

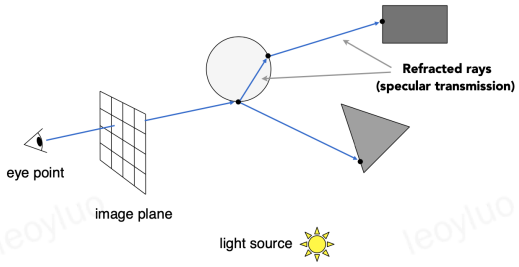
Pinhole Camera Model



从眼睛发出光源, 第一个交点, 反射光与光源的判断

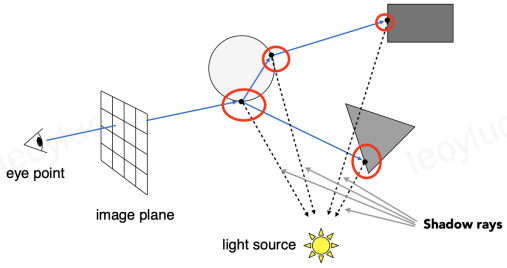
光的折射与反射

Recursive Ray Tracing



Whitted-Style Ray Tracing

Recursive Ray Tracing



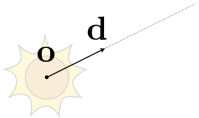
左右的碰撞点都对pixel的颜色有所贡献(每个点都是shading函数)

Ray与几何的交点

Ray Equation

Ray is defined by its origin and a direction vector

Example:



Ray equation:

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d} \qquad 0 \le t < \infty$$

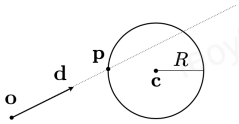
point along ray "time" origin (normalized) direction

ray的定义, 为射线

Ray Intersection With Sphere

Solve for intersection:

$$(\mathbf{o}+t\,\mathbf{d}-\mathbf{c})^2-R^2=0$$



$$a\,t^2+b\,t+c=0,\ \text{where}$$

$$a=\mathbf{d}\cdot\mathbf{d}$$

$$b=2(\mathbf{o}-\mathbf{c})\cdot\mathbf{d}$$

$$c=(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-R^2$$

$$t=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

光线与球的交点

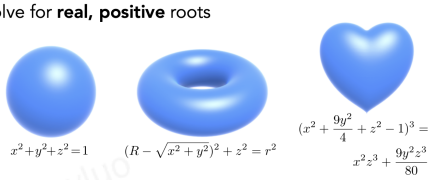
Ray Intersection With Implicit Surface

Ray: $\mathbf{r}(t)=\mathbf{o}+t\,\mathbf{d},\ 0\le t<\infty$

General implicit surface: $\mathbf{p}:f(\mathbf{p})=0$

Substitute ray equation: $f(\mathbf{o}+t\,\mathbf{d})=0$

Solve for **real, positive** roots



与任何隐式表面的求根 (数值计算)

光线与mesh求交

- 与每个三角形求交(数量级太大), 找到最近的t

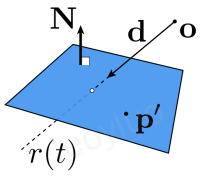
  ## Ray Intersection With Plane

  Ray equation:

  $$\mathbf{r}(t)=\mathbf{o}+t\,\mathbf{d},\ 0\le t<\infty$$

  Plane equation:

  $$\mathbf{p}:(\mathbf{p}-\mathbf{p}')\cdot\mathbf{N}=0$$

  Solve for intersection

  Set $\mathbf{p}=\mathbf{r}(t)$ and solve for t

  $$(\mathbf{p}-\mathbf{p}')\cdot\mathbf{N}=(\mathbf{o}+t\,\mathbf{d}-\mathbf{p}')\cdot\mathbf{N}=0$$

  $$t=\frac{(\mathbf{p}'-\mathbf{o})\cdot\mathbf{N}}{\mathbf{d}\cdot\mathbf{N}}$$ **Check:** $0\le t<\infty$

- 与平面先求交, 判断点是否在三角形内
- Möller Trumbore Algorithm (直接进行ray与三角形的判断) 使用了重心坐标

  ## Möller Trumbore Algorithm

  A faster approach, giving barycentric coordinate directly

  Derivation in the discussion section!

  $$\vec{\mathbf{O}}+t\vec{\mathbf{D}}=(1-b_1-b_2)\vec{\mathbf{P}}_0+b_1\vec{\mathbf{P}}_1+b_2\vec{\mathbf{P}}_2$$

  $$\begin{bmatrix}t\\b_1\\b_2\end{bmatrix}=\frac{1}{\vec{\mathbf{S}}_1\cdot\vec{\mathbf{E}}_1}\begin{bmatrix}\vec{\mathbf{S}}_2\cdot\vec{\mathbf{E}}_2\\\vec{\mathbf{S}}_1\cdot\vec{\mathbf{S}}\\\vec{\mathbf{S}}_2\cdot\vec{\mathbf{D}}\end{bmatrix}$$

  Cost = (1 div, 27 mul, 17 add)

  Where:

  $$\vec{\mathbf{E}}_1=\vec{\mathbf{P}}_1-\vec{\mathbf{P}}_0$$

  $$\vec{\mathbf{E}}_2=\vec{\mathbf{P}}_2-\vec{\mathbf{P}}_0$$

  $$\vec{\mathbf{S}}=\vec{\mathbf{O}}-\vec{\mathbf{P}}_0$$

  $$\vec{\mathbf{S}}_1=\vec{\mathbf{D}}\times\vec{\mathbf{E}}_2$$

  $$\vec{\mathbf{S}}_2=\vec{\mathbf{S}}\times\vec{\mathbf{E}}_1$$

  Recall: How to determine if the "intersection" is inside the triangle?

  Hint:
  (1-b1-b2), b1, b2 are barycentric coordinates!

包围盒

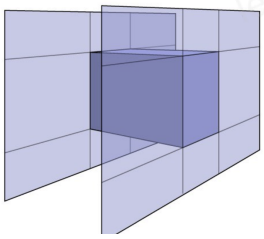
Ray-Intersection With Box

Understanding: **box is the intersection of 3 pairs of slabs**

Specifically:

We often use an **Axis-Aligned Bounding Box (AABB)**
(轴对齐包围盒)

i.e. any side of the BB is along either x, y, or z axis



用包围盒可以减少对物体上的三角面的计算

包围盒求交

Ray Intersection with Axis-Aligned Box

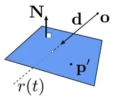
- Recall: a box (3D) = three pairs of infinitely large slabs
- Key ideas
  - The ray enters the box **only when** it enters all pairs of slabs
  - The ray exits the box **as long as** it exits any pair of slabs
- For each pair, calculate the t_{min} and t_{max} (negative is fine)
- For the 3D box, $t_{enter} = \max\{t_{min}\}$, $t_{exit} = \min\{t_{max}\}$
- If $t_{enter} < t_{exit}$, we know the ray **stays a while** in the box (so they must intersect!) (not done yet, see the next slide)

Ray Intersection with Axis-Aligned Box

- However, ray is not a line
  - Should check whether t is negative for physical correctness!
- What if $t_{exit} < 0$?
  - The box is "behind" the ray — no intersection!
- What if $t_{exit} >= 0$ and $t_{enter} < 0$?
  - The ray's origin is inside the box — have intersection!
- In summary, ray and AABB intersect iff
  - $t_{enter} < t_{exit}$ && $t_{exit} >= 0$

Why Axis-Aligned?

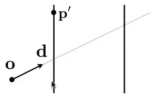
General



$$t = \frac{(\mathbf{p}' - \mathbf{o}) \cdot \mathbf{N}}{\mathbf{d} \cdot \mathbf{N}}$$

3 subtractions, 6 multiplies, 1 division

Slabs perpendicular to x-axis



$$t = \frac{\mathbf{p}'_x - \mathbf{o}_x}{\mathbf{d}_x}$$

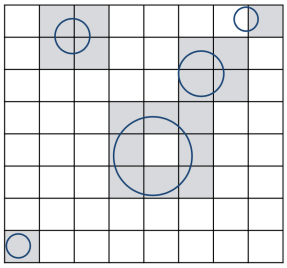
1 subtraction, 1 division

==========

Lecture14: Ray Tracing

如何使用aabb包围盒加速?

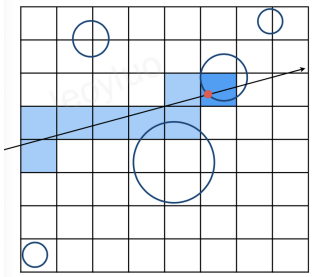
Preprocess – Build Acceleration Grid



1. Find bounding box
2. Create grid
3. Store each object in overlapping cells

提前构建包围盒, 并且把包围盒划分成grid

Ray-Scene Intersection



Step through grid in ray traversal order

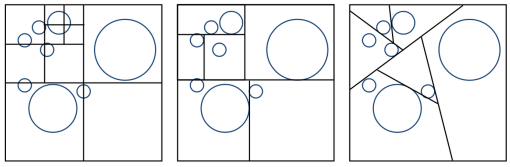
For each grid cell
Test intersection with all objects stored at that cell

光线经过格子, 判断格子是否包含, 在判断与物体求交 (需要判断ray经过哪些格子)

- 格子数量的划分影响了计算效率(太少没有加速, 太多遍历格子的数量太多)

Spatial Partition空间划分

Spatial Partitioning Examples

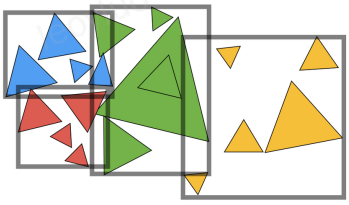


Oct-Tree **KD-Tree** BSP-Tree

Note: you could have these in both 2D and 3D. In lecture we will illustrate principles in 2D.

Bounding Volume Hierarchy (BVH) – object partition

Summary: Building BVHs

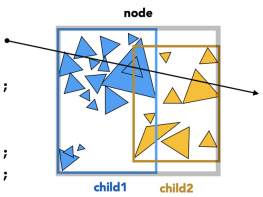


- Find bounding box
- Recursively split set of objects in two subsets
- **Recompute** the bounding box of the subsets
- Stop when necessary
- Store objects in each leaf node

- 每个物体属于单独某个包围盒
- 包围盒之间会有overlap

BVH Traversal

```
Intersect(Ray ray, BVH node) {
  if (ray misses node.bbox) return;

  if (node is a leaf node)
     test intersection with all objs;
     return closest intersection;

  hit1 = Intersect(ray, node.child1);
  hit2 = Intersect(ray, node.child2);

  return the closer of hit1, hit2;
}
```



bvh traversal, 递归算法

==========

Lecture15: Ray Tracing

- 辐射度量学

Definition: Radiant energy is the energy of electromagnetic radiation. It is measured in units of joules, and denoted by the symbol:

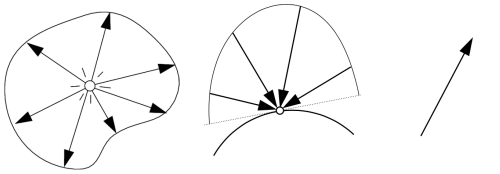
$$Q \ [\mathrm{J} = \mathrm{Joule}]$$

Definition: Radiant flux (power) is the energy emitted, reflected, transmitted or received, per unit time.

$$\Phi \equiv \frac{\mathrm{d}Q}{\mathrm{d}t} \ [\mathrm{W} = \mathrm{Watt}] \ [\mathrm{lm} = \mathrm{lumen}]^{*}$$

能量与功率

Important Light Measurements of Interest



Light Emitted From A Source

"Radiant Intensity"

Light Falling On A Surface

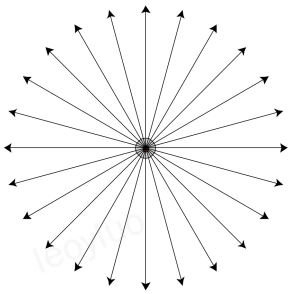
"Irradiance"

Light Traveling Along A Ray

"Radiance"

定义

Isotropic Point Source



$$\Phi = \int_{S^2} I \, \mathrm{d}\omega$$
$$= 4\pi I$$

$$I = \frac{\Phi}{4\pi}$$

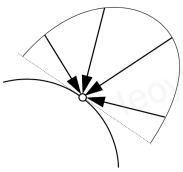
均匀分布时候, 每个角度的intensity

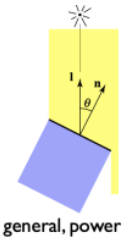
Irradiance

Definition: The irradiance is the power per unit area incident on a surface point.

$$E(\mathbf{x}) \equiv \frac{\mathrm{d}\Phi(\mathbf{x})}{\mathrm{d}A}$$

$$\left[\frac{\mathrm{W}}{\mathrm{m}^2}\right] \ \left[\frac{\mathrm{lm}}{\mathrm{m}^2} = \mathrm{lux}\right]$$





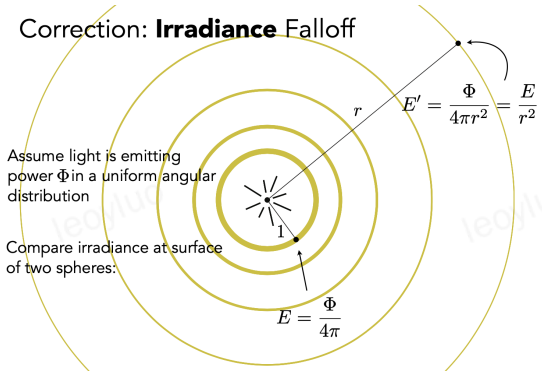
In general, power per uni area is proportional to $\cos\theta = l \cdot n$

$$E = \frac{\Phi}{A}\cos\theta$$

power per area (intensity: power per second)

面积与光线方向相关

Correction: **Irradiance** Falloff



intensity不衰减, irradiance会衰减

Radiance

Definition: The radiance (luminance) is the power emitted, reflected, transmitted or received by a surface, per unit solid angle, per projected unit area.

$$L(\mathrm{p},\omega) \equiv \frac{\mathrm{d}^2\Phi(\mathrm{p},\omega)}{\mathrm{d}\omega\,\mathrm{d}A\cos\theta}$$

$\cos\theta$ accounts for projected surface area

$$\left[\frac{\mathrm{W}}{\mathrm{sr\,m^2}}\right]\left[\frac{\mathrm{cd}}{\mathrm{m^2}} = \frac{\mathrm{lm}}{\mathrm{sr\,m^2}} = \mathrm{nit}\right]$$

是intensity和irradiance的结合, per area * per solid angle

Irradiance vs. Radiance

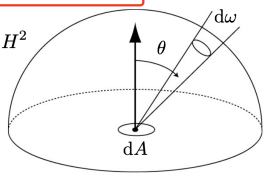
Irradiance: total power received by area dA

Radiance: power received by area dA from "direction" dω

$$dE(\mathrm{p},\omega) = L_i(\mathrm{p},\omega)\cos\theta\,\mathrm{d}\omega$$

$$E(\mathrm{p}) = \int_{H^2} L_i(\mathrm{p},\omega)\cos\theta\,\mathrm{d}\omega$$

Unit Hemisphere: H^2

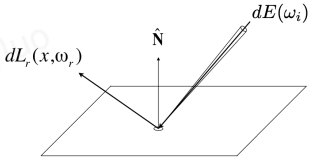


- BRDF
- 定义某个入射方向, 对某个反射方向的效果

Reflection at a Point

Radiance from direction ω_i turns into the power E that dA receives

Then power E will become the radiance to any other direction ω_o



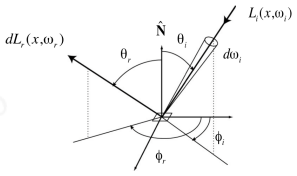
Differential irradiance incoming: $dE(\omega_i) = L(\omega_i)\cos\theta_i\,d\omega_i$

Differential radiance exiting (due to $dE(\omega_i)$): $dL_r(\omega_r)$

固定入射角, 分配到每个出方向的比例

BRDF

The Bidirectional Reflectance Distribution Function (BRDF) represents how much light is reflected into each outgoing direction ω_r from each incoming direction



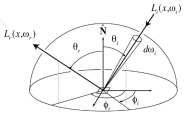
$$f_r(\omega_i \to \omega_r) = \frac{\mathrm{d}L_r(\omega_r)}{\mathrm{d}E_i(\omega_i)} = \frac{\mathrm{d}L_r(\omega_r)}{L_i(\omega_i)\cos\theta_i\,\mathrm{d}\omega_i} \quad \left[\frac{1}{\mathrm{sr}}\right]$$

镜面反射: 只往一个方向; 漫反射: 多个方向分布

Challenge: Recursive Equation

Reflected radiance depends on incoming radiance

$$L_r(\mathrm{p},\omega_r) = \int_{H^2} f_r(\mathrm{p},\omega_i \to \omega_r) L_i(\mathrm{p},\omega_i) \cos\theta_i\,\mathrm{d}\omega_i$$

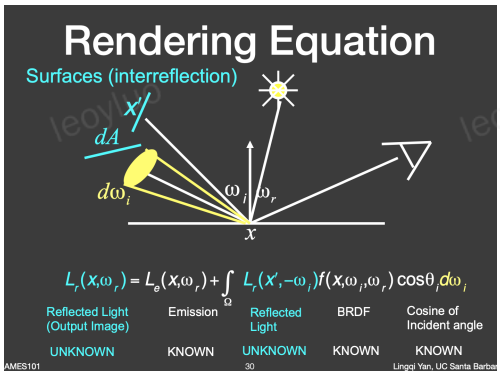


But incoming radiance depends on reflected radiance (at another point in the scene)

某个方向出射, 需要积分所有输入方向的贡献

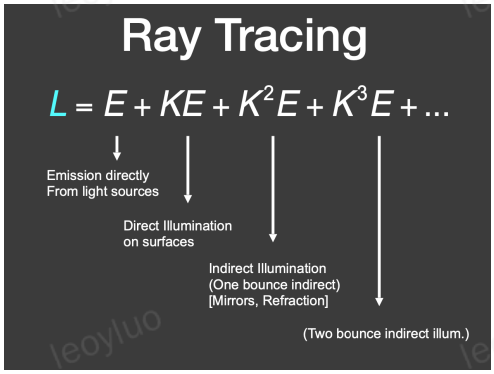
- **渲染方程**
- 包含自己发射的光, 和入射的光
- The Rendering Equation

$$L_o(p,\omega_o) = L_e(p,\omega_o) + \int_{\Omega^+} L_i(p,\omega_i) f_r(p,\omega_i,\omega_o)(n\cdot\omega_i)\,\mathrm{d}\omega_i$$



入射源: 点光源/面光源/其他物体反射的光源 (有递归)

简化式



直接+间接光照

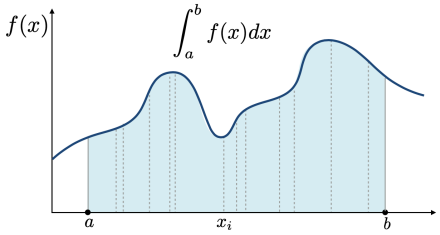
==========

Lecture16: Ray Tracing(Monte Carlo path tracing)

- 蒙特卡洛积分

Monte Carlo Integration

Why: we want to solve an integral, but it can be too difficult to solve analytically.



Monte Carlo Integration

Let us define the Monte Carlo estimator for the definite integral of given function $f(x)$

Definite integral	$\int_a^b f(x)dx$
Random variable	$X_i \sim p(x)$
Monte Carlo estimator	$F_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}$

大量采样, 取平均值

- Path tracing
- 基于whitted-style ray tracing(反射会弹射, 漫反射终止) – 实际上假设不完善

A Simple Monte Carlo Solution

$$L_o(p,\omega_o) \approx \frac{1}{N}\sum_{i=1}^{N}\frac{L_i(p,\omega_i)f_r(p,\omega_i,\omega_o)(n\cdot\omega_i)}{p(\omega_i)}$$

```
shade(p, wo)
    Randomly choose N directions wi~pdf
    Lo = 0.0
    For each wi
       Trace a ray r(p, wi)
       If ray r hit the light
          Lo += (1 / N) * L_i * f_r * cosine / pdf(wi)
    Return Lo
```

基于蒙特卡洛的brdf积分(此处只有**直接光照**)

Introducing Global Illumination

```
shade(p, wo)
    Randomly choose N directions wi~pdf
    Lo = 0.0
    For each wi
       Trace a ray r(p, wi)
       If ray r hit the light
          Lo += (1 / N) * L_i * f_r * cosine / pdf(wi)
       Else If ray r hit an object at q
          Lo += (1 / N) * shade(q, -wi) * f_r * cosine
          / pdf(wi)
    Return Lo
```

Is it done? **No.**

全局光照 (递归其他的物体! 间接光源)

间接光照 – 问题

- 指数爆炸(每次递归非常多ray, 栈溢出)
  - 只选择一个反射方向, 由反射probability决定 (这就是**path tracing**!)
  - 每个像素选择多个ray进行平均, 减少variance(ray generation的offset)
- 不会停止(无限反射)
  - russian roulette(RR): 随机选择是否射出光线

Solution: Russian Roulette (RR)

Previously, we always shoot a ray at a shading point and get the shading result **Lo**

Suppose we manually set a probability P (0 < P < 1)

With probability P, shoot a ray and return the **shading result divided by P**: **Lo / P**

With probability 1-P, don't shoot a ray and you'll get **0**

In this way, you can still **expect** to get Lo!:

E = P * **(Lo / P)** + (1 - P) * **0** = **Lo**

Solution: Russian Roulette (RR)

```
shade(p, wo)
    Manually specify a probability P_RR
    Randomly select ksi in a uniform dist. in [0, 1]
    If (ksi > P_RR) return 0.0;

    Randomly choose ONE direction wi~pdf(w)
    Trace a ray r(p, wi)
    If ray r hit the light
        Return L_i * f_r * cosine / pdf(wi) / P_RR
    Else If ray r hit an object at q
        Return shade(q, -wi) * f_r * cosine / pdf(wi) / P_RR
```

- 低效光线采样
  - 均匀的采样, 许多的ray打不到光源会被浪费

Sampling the Light

Need to make the rendering equation as an integral of dA

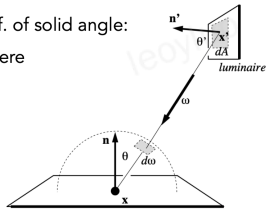
Need the relationship between dω and dA

Easy! Recall the alternative def. of solid angle:

Projected area on the unit sphere

$$d\omega = \frac{dA\cos\theta'}{\|x' - x\|^2}$$

(Note: θ', not θ)



把光源投影到采样域上, 再光源投射区域上采样

  - 新的概率在A上进行

Sampling the Light

```
shade(p, wo)
    # Contribution from the light source.
    Uniformly sample the light at x' (pdf_light = 1 / A)
    L_dir = L_i * f_r * cos θ * cos θ' / |x' - p|^2 / pdf_light

    # Contribution from other reflectors.
    L_indir = 0.0
    Test Russian Roulette with probability P_RR
    Uniformly sample the hemisphere toward wi (pdf_hemi = 1 / 2pi)
    Trace a ray r(p, wi)
    If ray r hit a non-emitting object at q
        L_indir = shade(q, -wi) * f_r * cos θ / pdf_hemi / P_RR

    Return L_dir + L_indir
```

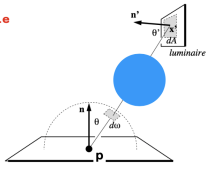
直接光源高效采样, 间接光源RR

Sampling the Light

One final thing: how do we know if the sample on the light is not blocked or not?

```
# Contribution from the light source.
L_dir = 0.0
Uniformly sample the light at x' (pdf_light = 1 / A)
Shoot a ray from p to x'
If the ray is not blocked in the middle
    L_dir = …
```

Now path tracing is finally done!



考虑光源是否被遮盖

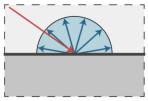
- radiance和最后的rgb并不一定一致, 有可能经过更多的图像处理获取最后的rgb

==========

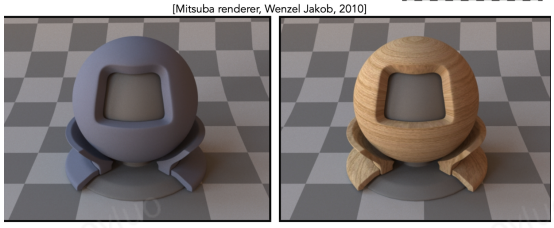
Lecture17: Material and Appearance

materail == brdf

Diffuse / Lambertian Material (BRDF)



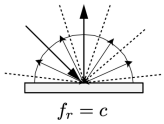
[Mitsuba renderer, Wenzel Jakob, 2010]



Uniform colored diffuse BRDF　　Textured diffuse BRDF

Diffuse / Lambertian Material

Light is equally reflected in each output direction



$f_r = c$

Suppose the incident lighting is uniform:

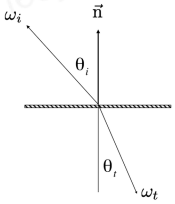
$$\begin{aligned} L_o(\omega_o) &= \int_{H^2} f_r L_i(\omega_i) \cos\theta_i \, \mathrm{d}\omega_i \\ &= f_r L_i \int_{H^2} \cancel{(\omega_i)} \cos\theta_i \, \mathrm{d}\omega_i \\ &= \pi f_r L_i \end{aligned}$$

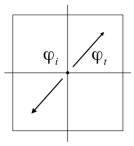
$$f_r = \frac{\rho}{\pi}$$ — albedo (color)

- 各种反射/折射形式(略) – call散射
  - brdf, btdf, bsdf
- 折射定律

Snell's Law

Transmitted angle depends on
index of refraction (IOR) for incident ray
index of refraction (IOR) for exiting ray





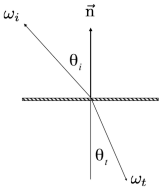
$\varphi_t = \varphi_i \pm \pi$

Medium	η^*
Vacuum	1.0
Air (sea level)	1.00029
Water (20°C)	1.333
Glass	1.5-1.6
Diamond	2.42

* index of refraction is wavelength dependent (these are averages)

- $\eta_i \sin\theta_i = \eta_t \sin\theta_t$

Law of Refraction



$$\eta_i \sin\theta_i = \eta_t \sin\theta_t$$

$$\begin{aligned} \cos\theta_t &= \sqrt{1 - \sin^2\theta_t} \\ &= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 \sin^2\theta_i} \\ &= \sqrt{1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2\ \theta_i)} \end{aligned}$$

Total internal reflection:

$$1 - \left(\frac{\eta_i}{\eta_t}\right)^2 (1 - \cos^2\theta_i) < 0$$

When light is moving from a more optically dense medium to a less optically dense medium: $\frac{\eta_i}{\eta_t} > 1$

Light incident on boundary from large enough angle will not exit medium.

有可能无法折射(折射率大–》小), 全反射

- brdf性质

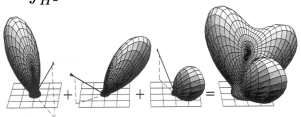
Properties of BRDFs

- Non-negativity

$$f_r(\omega_i \to \omega_r) \geq 0$$

- Linearity

$$L_r(\mathrm{p}, \omega_r) = \int_{H^2} f_r(\mathrm{p}, \omega_i \to \omega_r) \, L_i(\mathrm{p}, \omega_i) \, \cos\theta_i \, \mathrm{d}\omega_i$$

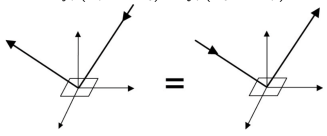


[Sillion et al. 1990]

Properties of BRDFs

- Reciprocity principle

$$f_r(\omega_r \to \omega_i) = f_r(\omega_i \to \omega_r)$$



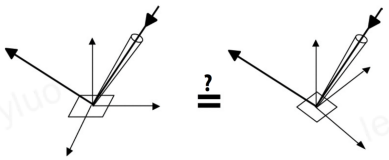
- Energy conservation

$$\forall \omega_r \int_{H^2} f_r(\omega_i \to \omega_r) \, \cos\theta_i \, \mathrm{d}\omega_i \leq 1$$

Properties of BRDFs

- Isotropic vs. anisotropic
  - If isotropic, $f_r(\theta_i, \phi_i; \theta_r, \phi_r) = f_r(\theta_i, \theta_r, \phi_r - \phi_i)$
  - Then, from reciprocity,

$$f_r(\theta_i, \theta_r, \phi_r - \phi_i) = f_r(\theta_r, \theta_i, \phi_i - \phi_r) = f_r(\theta_i, \theta_r, |\phi_r - \phi_i|)$$



==========

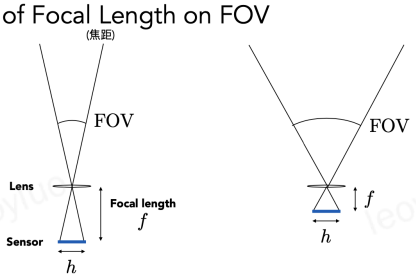
Lecture18: Advanced Topics In Rendering

pass

==========

Lecture20: Cameras, Lens

Effect of Focal Length on FOV



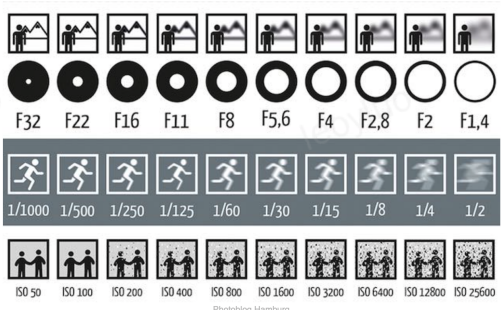
For a fixed sensor size, decreasing the focal length increases the field of view.

$$\text{FOV} = 2\arctan\left(\frac{h}{2f}\right)$$

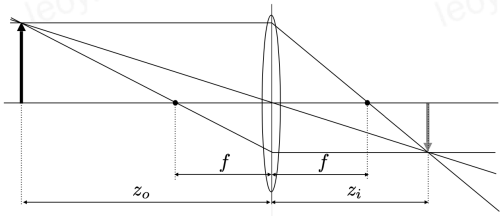
Exposure

- H = T x E
- Exposure = time x irradiance
- Exposure time (T)
  - Controlled by shutter
- Irradiance (E)
  - Power of light falling on a unit area of sensor
  - Controlled by lens aperture and focal length

Exposure: Aperture, Shutter, Gain (ISO)



The Thin Lens Equation



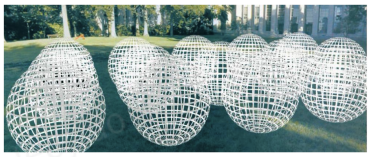
$$\frac{1}{f} = \frac{1}{z_i} + \frac{1}{z_o}$$

==========

Lecture21: Color And Perception

- 全光函数 (VR)

The Plenoptic Function



$$P(\theta,\phi,\lambda,t,V_X,V_Y,V_Z)$$

- Can reconstruct every possible view, at every moment, from every position, at every wavelength
- Contains every photograph, every movie, everything that anyone has ever seen! it completely captures our visual reality! Not bad for a function…

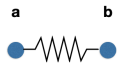
==========

Lecture22&23: Animation

- 弹簧系统

Non-Zero Length Spring

Spring with non-zero rest length



$$\boldsymbol{f}_{a\to b} = k_s \frac{\boldsymbol{b}-\boldsymbol{a}}{||\boldsymbol{b}-\boldsymbol{a}||}(||\boldsymbol{b}-\boldsymbol{a}|| - l)$$

Rest length

Internal Damping for Spring

Damp only the internal, spring-driven motion

Relative velocity of b, assuming a is static (vector)

$$\boldsymbol{f}_{\boldsymbol{b}} = -k_d \frac{\boldsymbol{b}-\boldsymbol{a}}{||\boldsymbol{b}-\boldsymbol{a}||}(\dot{\boldsymbol{b}}-\dot{\boldsymbol{a}}) \cdot \frac{\boldsymbol{b}-\boldsymbol{a}}{||\boldsymbol{b}-\boldsymbol{a}||}$$

Damping force applied on b

Relative velocity projected to the direction from a to b (scalar)

Direction from a to b

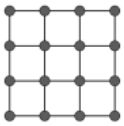
- Viscous drag only on change in spring length
  - Won't slow group motion for the spring system (e.g. global translation or rotation of the group)
- Note: This is only one specific type of damping

拉力与摩擦力

- 高维弹簧结构

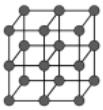
Structures from Springs

Sheets



Blocks

Others

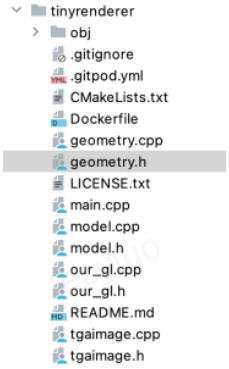


- 粒子系统, 动力学/逆动力学

==========

Tiny-render代码

- https://github.com/ssloy/tinyrenderer
- 用于学习光栅化基本知识的实现
- 基本框架



- (1) MVP相机参数和几何计算
- geometry, our_gl.cpp

```
#include "our_gl.h"

mat<4,4> ModelView;
mat<4,4> Viewport;
mat<4,4> Projection;

void viewport(const int x, const int y, const int w, const int h) {
    Viewport = {{{w/2., 0, 0, x+w/2.}, {0, h/2., 0, y+h/2.}, {0,0,1,0}, {0,0,0,1}}};
}

void projection(const double f) { // check https://en.wikipedia.org/wiki/Camera_matrix
    Projection = {{{1,0,0,0}, {0,-1,0,0}, {0,0,1,0}, {0,0,-1/f,0}}};
}

void lookat(const vec3 eye, const vec3 center, const vec3 up) { // check https://github.com/ssloy/tinyrenderer/wiki/Lesson-5-Moving-the-camera
    vec3 z = (center-eye).normalized();
    vec3 x =  cross(up,z).normalized();
    vec3 y =  cross(z, x).normalized();
    mat<4,4> Minv = {{{x.x,x.y,x.z,0}, {y.x,y.y,y.z,0}, {z.x,z.y,z.z,0}, {0,0,0,1}}};
    mat<4,4> Tr   = {{{1,0,0,-eye.x}, {0,1,0,-eye.y}, {0,0,1,-eye.z}, {0,0,0,1}}};
    ModelView = Minv*Tr;
}
```

- (2) obj模型读取与texture/diff/spec
- model.cpp

```
Model::Model(const std::string filename) {
    std::ifstream in;
    in.open(filename, std::ifstream::in);
    if (in.fail()) return;
    std::string line;
    while (!in.eof()) {
        std::getline(in, line);
        std::istringstream iss(line.c_str());
        char trash;
        if (!line.compare(0, 2, "v ")) {
            iss >> trash;
            vec3 v;
            for (int i=0;i<3;i++) iss >> v[i];
            verts.push_back(v);
        } else if (!line.compare(0, 3, "vn ")) {
            iss >> trash >> trash;
            vec3 n;
            for (int i=0;i<3;i++) iss >> n[i];
            norms.push_back(n.normalized());
        } else if (!line.compare(0, 3, "vt ")) {
            iss >> trash >> trash;
            vec2 uv;
            for (int i=0;i<2;i++) iss >> uv[i];
            tex_coord.push_back({uv.x, 1-uv.y});
        }  else if (!line.compare(0, 2, "f ")) {
            int f,t,n;
            iss >> trash;
            int cnt = 0;
            while (iss >> f >> trash >> t >> trash >> n) {
```

```
class Model {
    std::vector<vec3> verts{};    // array of vertices
    std::vector<vec2> tex_coord{}; // per-vertex array of tex coords
    std::vector<vec3> norms{};    // per-vertex array of normal vectors
    std::vector<int> facet_vrt{};
    std::vector<int> facet_tex{};  // per-triangle indices in the above arrays
    std::vector<int> facet_nrm{};
    TGAImage diffusemap{};         // diffuse color texture
    TGAImage normalmap{};          // normal map texture
    TGAImage specularmap{};        // specular map texture
    void load_texture(const std::string filename, const std::string suffix, TGAImage &img);
public:
    Model(const std::string filename);
    int nverts() const;
    int nfaces() const;
    vec3 normal(const int iface, const int nthvert) const; // per triangle corner normal vertex
    vec3 normal(const vec2 &uv) const;                     // fetch the normal vector from the normal
    vec3 vert(const int i) const;
    vec3 vert(const int iface, const int nthvert) const;
    vec2 uv(const int iface, const int nthvert) const;
    const TGAImage& diffuse()  const { return diffusemap;  }
    const TGAImage& specular() const { return specularmap; }
};
```

(3) blind-phong模型计算

- 类似opengl的shader构建
- main.cpp, our_gl.cpp

```
int main(int argc, char** argv) {
    if (2>argc) {
        std::cerr << "Usage: " << argv[0] << " obj/model.obj" << std::endl;
        return 1;
    }
    TGAImage framebuffer(width, height, TGAImage::RGB); // the output image
    lookat(eye, center, up);                            // build the ModelView matrix
    viewport(width/8, height/8, width*3/4, height*3/4); // build the Viewport matrix
    projection((eye-center).norm());                    // build the Projection matrix
    std::vector<double> zbuffer(width*height, std::numeric_limits<double>::max());

    for (int m=1; m<argc; m++) { // iterate through all input objects
        Model model(argv[m]);
        Shader shader(model);
        for (int i=0; i<model.nfaces(); i++) { // for every triangle
            vec4 clip_vert[3]; // triangle coordinates (clip coordinates), written by VS, read by FS
            for (int j : {0,1,2})
                shader.vertex(i, j, clip_vert[j]); // call the vertex shader for each triangle vertex
            triangle(clip_vert, shader, framebuffer, zbuffer); // actual rasterization routine call
        }
    }
    framebuffer.write_tga_file("framebuffer.tga");
    return 0;
}
```

MVP Model, basic loop for each face processing

```
struct Shader : IShader {
    const Model &model;
    vec3 uniform_l;       // light direction in view coordinates
    mat<2,3> varying_uv;  // triangle uv coordinates, written by the vertex shader, read by the fragment shader
    mat<3,3> varying_nrm; // normal per vertex to be interpolated by FS
    mat<3,3> view_tri;    // triangle in view coordinates

    Shader(const Model &m) : model(m) {
        uniform_l = proj<3>((ModelView*embed<4>(light_dir, 0.))).normalized(); // transform the light vector to view coordinates
    }

    virtual void vertex(const int iface, const int nthvert, vec4& gl_Position) {
        varying_uv.set_col(nthvert, model.uv(iface, nthvert));
        varying_nrm.set_col(nthvert, proj<3>((ModelView).invert_transpose()*embed<4>(model.normal(iface, nthvert), 0.)));
        gl_Position= ModelView*embed<4>(model.vert(iface, nthvert));
        view_tri.set_col(nthvert, proj<3>(gl_Position));
        gl_Position = Projection*gl_Position;
    }

    virtual bool fragment(const vec3 bar, TGAColor &gl_FragColor) {
        vec3 bn = (varying_nrm*bar).normalized(); // per-vertex normal interpolation
        vec2 uv = varying_uv*bar; // tex coord interpolation

        // for the math refer to the tangent space normal mapping lecture
        // https://github.com/ssloy/tinyrenderer/wiki/Lesson-6bis-tangent-space-normal-mapping
        mat<3,3> AI = mat<3,3>{ {view_tri.col(1) - view_tri.col(0), view_tri.col(2) - view_tri.col(0), bn} }.invert();
        vec3 i = AI * vec3{varying_uv[0][1] - varying_uv[0][0], varying_uv[0][2] - varying_uv[0][0], 0};
        vec3 j = AI * vec3{varying_uv[1][1] - varying_uv[1][0], varying_uv[1][2] - varying_uv[1][0], 0};
        mat<3,3> B = mat<3,3>{ {i.normalized(), j.normalized(), bn} }.transpose();

        vec3 n = (B * model.normal(uv)).normalized(); // transform the normal from the texture to the tangent space
        double diff = std::max(0., n*uniform_l); // diffuse light intensity
        vec3 r = (n*(n*uniform_l)*2 - uniform_l).normalized(); // reflected light direction, specular mapping is described here: https://github.com/ssloy/tinyrenderer/wiki/Le
        double spec = std::pow(std::max(-r.z, 0.), 5+sample2D(model.specular(), uv)[0]); // specular intensity, note that the camera lies on the z-axis (in view), therefore s

        TGAColor c = sample2D(model.diffuse(), uv);
        for (int i : {0,1,2})
            gl_FragColor[i] = std::min<int>(10 + c[i]*(diff + spec), 255); // (a bit of ambient light, diff + spec), clamp the result

        return false; // the pixel is not discarded
    }
};
```

vert project, triangle rasterization, blind-phong model

```
void triangle(const vec4 clip_verts[3], IShader &shader, TGAImage &image, std::vector<double> &zbuffer) {
    vec4 pts[3]  = { Viewport*clip_verts[0],    Viewport*clip_verts[1],    Viewport*clip_verts[2]    };  // triangle screen coordinates before persp. division
    vec2 pts2[3] = { proj<2>(pts[0]/pts[0][3]), proj<2>(pts[1]/pts[1][3]), proj<2>(pts[2]/pts[2][3]) };  // triangle screen coordinates after  persp. division

    int bboxmin[2] = {image.width()-1, image.height()-1};
    int bboxmax[2] = {0, 0};
    for (int i=0; i<3; i++)
        for (int j=0; j<2; j++) {
            bboxmin[j] = std::min(bboxmin[j], static_cast<int>(pts2[i][j]));
            bboxmax[j] = std::max(bboxmax[j], static_cast<int>(pts2[i][j]));
        }
#pragma omp parallel for
    for (int x=std::max(bboxmin[0], 0); x<=std::min(bboxmax[0], image.width()-1); x++) {
        for (int y=std::max(bboxmin[1], 0); y<=std::min(bboxmax[1], image.height()-1); y++) {
            vec3 bc_screen = barycentric(pts2, {static_cast<double>(x), static_cast<double>(y)});
            vec3 bc_clip   = {bc_screen.x/pts[0][3], bc_screen.y/pts[1][3], bc_screen.z/pts[2][3]};
            bc_clip = bc_clip/(bc_clip.x+bc_clip.y+bc_clip.z); // check https://github.com/ssloy/tinyrenderer/wiki/Technical-difficulties-linear-interpolation-with-perspectiv
            double frag_depth = vec3{clip_verts[0][2], clip_verts[1][2], clip_verts[2][2]}*bc_clip;
            if (bc_screen.x<0 || bc_screen.y<0 || bc_screen.z<0 || frag_depth > zbuffer[x+y*image.width()]) continue;
            TGAColor color;
            if (shader.fragment(bc_clip, color)) continue; // fragment shader can discard current fragment
            zbuffer[x+y*image.width()] = frag_depth;
            image.set(x, y, color);
        }
    }
}
```

z_buffer and bbox for triangle

==========

Tiny-raytracing代码

- https://github.com/ssloy/tinyraytracer

- 材质定义

```
struct Material {
    float refractive_index = 1;
    float albedo[4] = {2,0,0,0};
    vec3 diffuse_color = {0,0,0};
    float specular_exponent = 0;
};
```

```
constexpr Material      ivory = {1.0, {0.9,  0.5, 0.1, 0.0}, {0.4, 0.4, 0.3},   50.};
constexpr Material      glass = {1.5, {0.0,  0.9, 0.1, 0.8}, {0.6, 0.7, 0.8},  125.};
constexpr Material red_rubber = {1.0, {1.4,  0.3, 0.0, 0.0}, {0.3, 0.1, 0.1},   10.};
constexpr Material     mirror = {1.0, {0.0, 16.0, 0.8, 0.0}, {1.0, 1.0, 1.0}, 1425.};
```

- 球体, 光源定义

```
constexpr Sphere spheres[] = {
    {{-3,    0,   -16}, 2,      ivory},
    {{-1.0, -1.5, -12}, 2,      glass},
    {{ 1.5, -0.5, -18}, 3, red_rubber},
    {{ 7,    5,   -18}, 4,     mirror}
};

constexpr vec3 lights[] = {
    {-20, 20,  20},
    { 30, 50, -25},
    { 30, 20,  30}
};
```

- 反射与折射

```
vec3 reflect(const vec3 &I, const vec3 &N) {
    return I - N*2.f*(I*N);
}

vec3 refract(const vec3 &I, const vec3 &N, const float eta_t, const float eta_i=1.f) { // Snell's law
    float cosi = - std::max(-1.f, std::min(1.f, I*N));
    if (cosi<0) return refract(I, -N, eta_i, eta_t); // if the ray comes from the inside the object, swap the air and the media
    float eta = eta_i / eta_t;
    float k = 1 - eta*eta*(1 - cosi*cosi);
    return k<0 ? vec3{1,0,0} : I*eta + N*(eta*cosi - std::sqrt(k)); // k<0 = total reflection, no ray to refract. I refract it anyways, this has no physical meaning
}
```

- 初始逻辑

```
int main() {
    constexpr int   width  = 1024;
    constexpr int   height = 768;
    constexpr float fov    = 1.05; // 60 degrees field of view in radians
    std::vector<vec3> framebuffer(width*height);
#pragma omp parallel for
    for (int pix = 0; pix<width*height; pix++) { // actual rendering loop
        float dir_x =  (pix%width + 0.5) -  width/2.;
        float dir_y = -(pix/width + 0.5) + height/2.; // this flips the image at the same time
        float dir_z = -height/(2.*tan(fov/2.));
        framebuffer[pix] = cast_ray(vec3{0,0,0}, vec3{dir_x, dir_y, dir_z}.normalized());
    }

    std::ofstream ofs("./out.ppm", std::ios::binary);
    ofs << "P6\n" << width << " " << height << "\n255\n";
    for (vec3 &color : framebuffer) {
        float max = std::max(1.f, std::max(color[0], std::max(color[1], color[2])));
        for (int chan : {0,1,2})
            ofs << (char)(255 *  color[chan]/max);
    }
    return 0;
}
```

每个像素射出一条光线, 向场景中进行计算(可openmp并行加速)

- cast ray
  - 检测与最近球体的碰撞, 递归计算(最多四次)折射与反射曲线曲线的颜色
  - 与球体碰撞点的颜色, 是通过光源与碰撞点的颜色逆向获取的
  - 着色函数类似于blind-phong(考虑diffuse, spec等等), 考虑光线方向的系数, 添加reflect和refract的颜色

```
vec3 cast_ray(const vec3 &orig, const vec3 &dir, const int depth=0) {
    auto [hit, point, N, material] = scene_intersect(orig, dir);
    if (depth>4 || !hit)
        return {0.2, 0.7, 0.8}; // background color

    vec3 reflect_dir = reflect(dir, N).normalized();
    vec3 refract_dir = refract(dir, N, material.refractive_index).normalized();
    vec3 reflect_color = cast_ray(point, reflect_dir, depth + 1);
    vec3 refract_color = cast_ray(point, refract_dir, depth + 1);

    float diffuse_light_intensity = 0, specular_light_intensity = 0;
    for (const vec3 &light : lights) { // checking if the point lies in the shadow of the light
        vec3 light_dir = (light - point).normalized();
        auto [hit, shadow_pt, trashnrm, trashmat] = scene_intersect(point, light_dir);
        if (hit && (shadow_pt-point).norm() < (light-point).norm()) continue;
        diffuse_light_intensity  += std::max(0.f, light_dir*N);
        specular_light_intensity += std::pow(std::max(0.f, -reflect(-light_dir, N)*dir), material.specular_exponent);
    }
    return material.diffuse_color * diffuse_light_intensity * material.albedo[0] + vec3{1., 1., 1.}*specular_light_intensity * material.albedo[1] + reflect_color*material.albedo[2] + refract_color*material.albe
}
```

- scene intersect

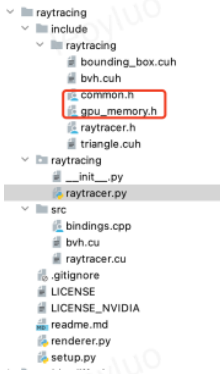
```
std::tuple<bool,vec3,vec3,Material> scene_intersect(const vec3 &orig, const vec3 &dir) {
    vec3 pt, N;
    Material material;

    float nearest_dist = 1e10;
    if (std::abs(dir.y)>.001) { // intersect the ray with the checkerboard, avoid division by zero
        float d = -(orig.y+4)/dir.y; // the checkerboard plane has equation y = -4
        vec3 p = orig + dir*d;
        if (d>.001 && d<nearest_dist && std::abs(p.x)<10 && p.z<-10 && p.z>-30) {
            nearest_dist = d;
            pt = p;
            N = {0,1,0};
            material.diffuse_color = (int(.5*pt.x+1000) + int(.5*pt.z)) & 1 ? vec3{.3, .3, .3} : vec3{.3, .2, .1};
        }
    }

    for (const Sphere &s : spheres) { // intersect the ray with all spheres
        auto [intersection, d] = ray_sphere_intersect(orig, dir, s);
        if (!intersection || d > nearest_dist) continue;
        nearest_dist = d;
        pt = orig + dir*nearest_dist;
        N = (pt - s.center).normalized();
        material = s.material;
    }
    return { nearest_dist<1000, pt, N, material };
}
```

与每个球进行碰撞, 或者检测与checkboard的碰撞

==========

torch-bind ray-tracing代码

- https://github.com/ashawkey/raytracing
- 基于torch的数据结构和tensor运算, 外加cuda/c++ extension的加速计算
- gui基于dearpygui
- 引入tinycudann中的cuda kernel template和计算函数



实现bounding box和bvh

- bvh构建 (核心实现都在bvh.cu中) 与计算
  - 通过所有的triangles, 首先构建最大3d的aabb bbox
  - 通过分层split构建bvh tree
  - 渲染的时候通过ray和整个bvh进行碰撞检测, 找到对应的三角形, 然后计算该点的颜色,深度,法向量