Software Renderer

👀软光栅项目：用于深入理解图形api、GPU硬件以及图形渲染管线～

0.参考资料

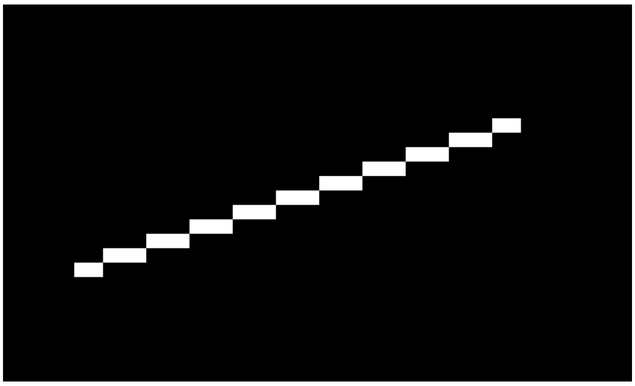
- ssloy - tinyrender：300行代码实现基础软光栅
  - > https://github.com/ssloy/tinyrenderer/wiki
  - 图片格式：.tga；模型数据：.obj

1.画线

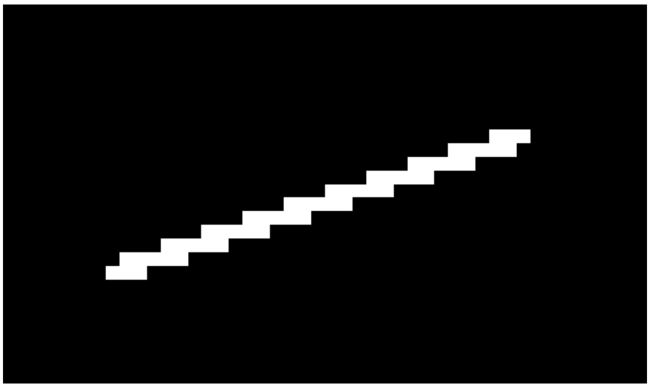
给定起点终点，画一条线段，**~~线段跨过的像素会被着色~~（这个理解是错误的🙅）**

=> 我们以下图为例分析：这张图是我们实际光栅化得到的结果**【Bresenham画线算法】**

分析特征可以发现：它一定在某一个方向上扫描过去只有一个点，比如下图每个x值只对应一个y



如果每个被跨过的像素都被着色，那么结果是：明显是不对的



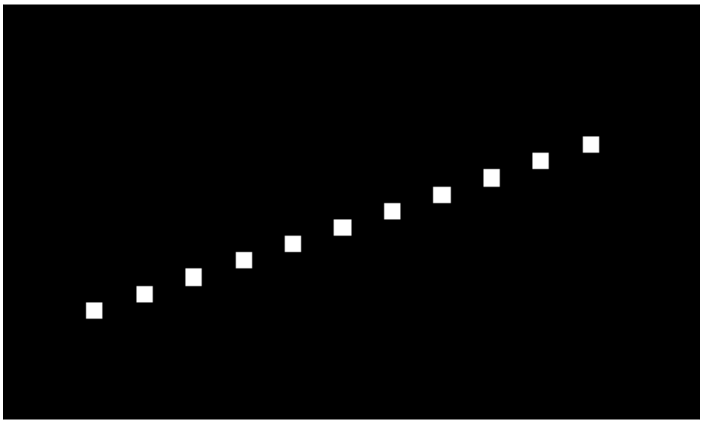
Bresenham画线算法

画线算法的核心思想：以x为基准，每次将x递增，同时根据斜率计算y的值，且完全使用整数计算

```
// version 1
for(int x = x0; x <= x1; x++)
{
    float t = (x - x0) / float(x1 - x0);
    int y = y0 * (1.0 - t) + y1 * t;
    image.set(x, y, color);
}
```

Version 1代码有一些问题：（1）没有考虑x0与x1相等的情况，也就是斜率不存在（2）循环默认了x0必须小于x1（3）没有考虑四舍五入

改进的Bresenham算法的步骤：

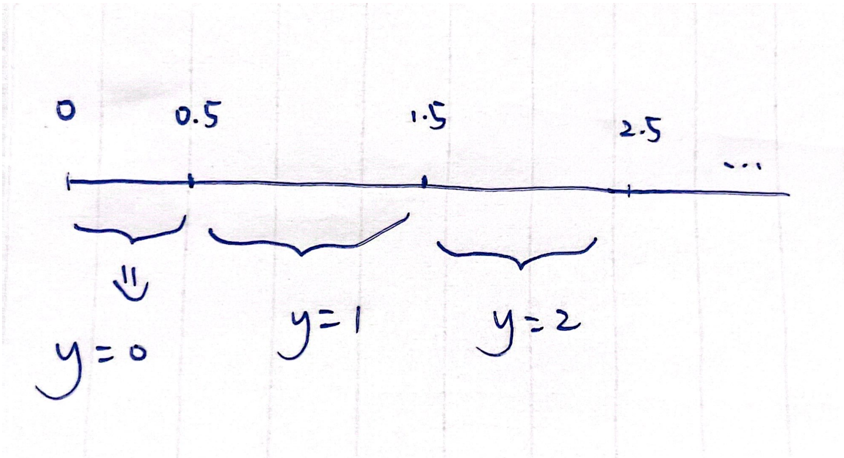
- **【确定方向为基准迭代】**确定x/y哪个方向横跨长度更大，我们以这个为基准进行迭代递增 => 为什么？实践发现，如果选择横跨长度小的方向，图像是断开的，如下👇：因为横跨长度越大，该方向的变化速率越大，假如k_y<k_x，选择y作为迭代对线，就是让Δy=1，那么Δx>1，就会有可能断开
    - 这样做的好处，可以避免出现x0=x1或者y0=y1造成除法错误
    - 
- **【以绝对值较小的作为起点】**便于for循环的书写
- **【四舍五入误差机制】**如果用float值，然后用round对其四舍五入，可能会因为浮点数的精度问题产生误差 => 改进的Bresenham算法，**完全使用整数计算，避免了误差**
    - 我们来研究一下这个误差机制：

      最初版本，我们通过浮点数来研究，x方向每递进循环一次，|y|其实会增加Δx·|dy/dx|，我们可以单独拿一个初始值为0.0的变量error来记录y的小数部分，每次循环error+=|dy/dx|，当error大于0.5时，我们可以四舍五入认为y可以+1了，最后我们将error-=1.0

      ```
      // version 1
      ...
      int dx = x1-x0;
      int dy = y1-y0;
      float derror = std::abs(dy/float(dx));
      float error = 0;
      int y = y0;
      for (int x=x0; x<=x1; x++) {
          if (steep) {
              image.set(y, x, color);
      ```

```
    } else {
        image.set(x, y, color);
    }
    error += derror; // 这个版本跟round()思路一致
    if (error>.5) {
        y += (y1>y0?1:-1);
        error -= 1.;
    }
}
```

所以我们观察y或者error这个变量，起初值是0.0，当它第一大于0.5时，他就进入了一个循环，如下图👇

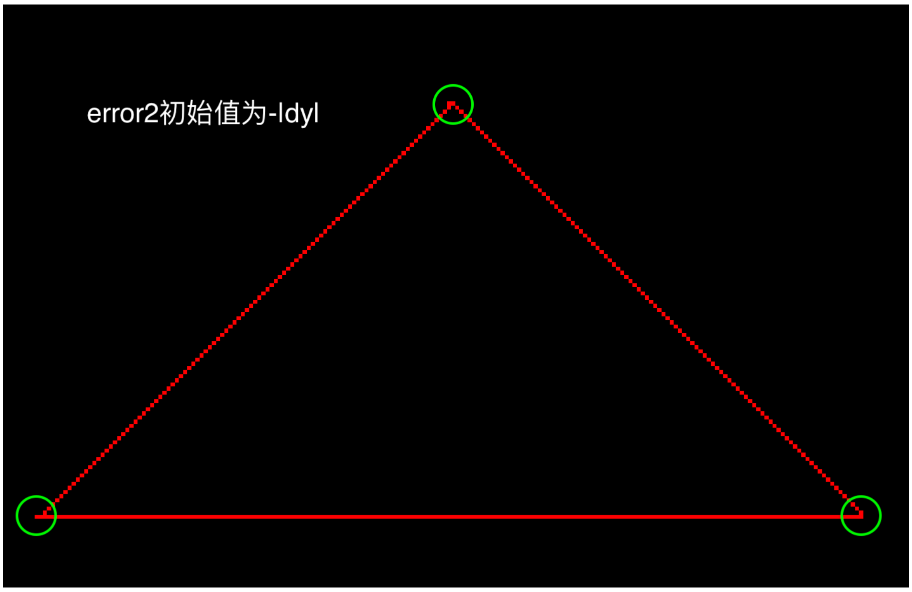


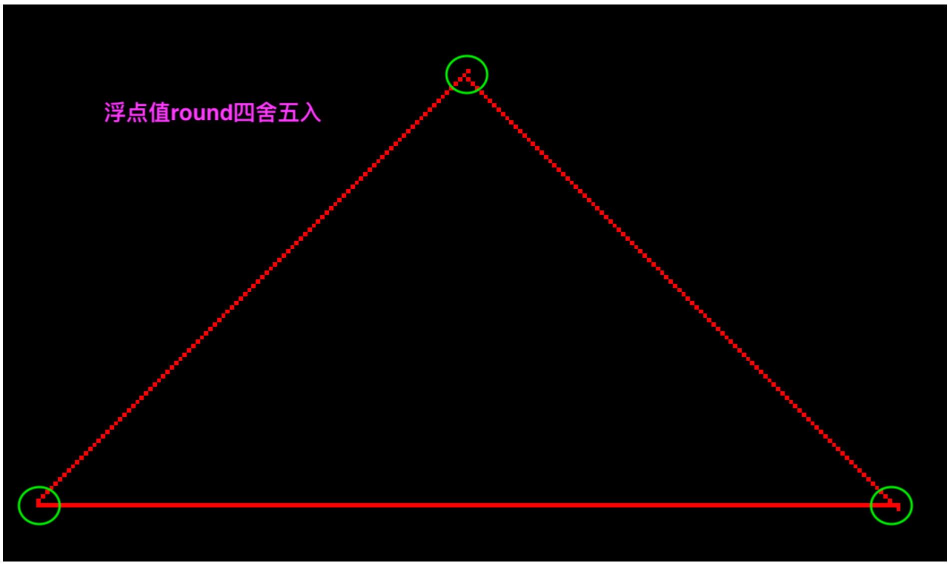
那么我们能用整数实现这个思路吗，避免了因浮点数精度问题导致的误差？首先相关系数整体乘dx，再整体成2：

```
// final version
...
int dx = x1-x0;
int dy = y1-y0;
int derror = std::abs(dy) * 2;
float error = 0;
int y = y0;
for (int x = x0; x <= x1; x++) {
    if (steep) {
        image.set(y, x, color);
    } else {
        image.set(x, y, color);
    }
    error += derror;
    if (error > dx) {
        y += (y1 > y0 ? 1 : -1);
        error -= 2 * dx;
    }
}
```

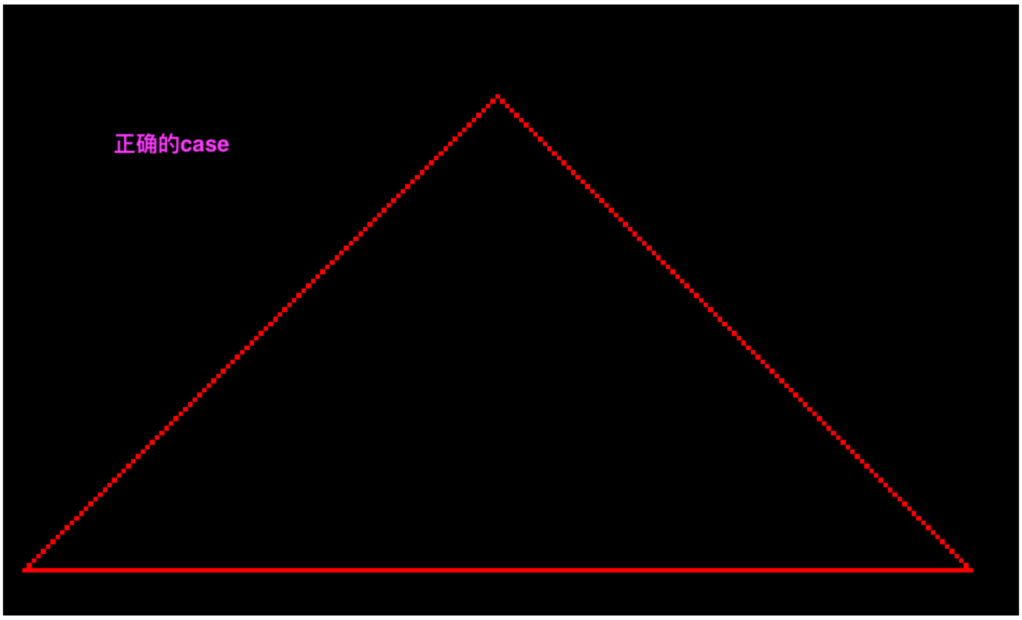
- => 这个相当影响最终效果，我们可以画一个三角形，看线段收尾是否连接丝滑，没有超出或断开

- 比如误差机制错误的情况：





- 正确计算的情况：



代码如下：

```
// final version
void line(int x0, int y0, int x1, int y1, TGAImage& image, TGAColor color)
{
  bool steep = false;
  if(abs(x0 - x1) < abs(y0 - y1))
  {
    swap(x0, y0);
    swap(x1, y1);
    steep = true;
  }

  if(x0 > x1)
  {
    swap(x0, x1);
    swap(y0, y1);
  }

  int dx = x1 - x0;
  int dy = y1 - y0;
  int derror2 = std::abs(dy) * 2;
  int error2 = 0;
  int y = y0;
  for(int x = x0; x <= x1; x++)
  {
    if(steep)
      image.set(y, x, color);
    else
      image.set(x, y, color);

    error2 += derror2;
    if (error2 > dx) {
      y += (y1 > y0 ? 1 : -1);
      error2 -= dx * 2;
    }
  }
}
```

2.光栅化

任何多边形可以被拆分为多个三角形（硬件可以为我们轻松做到），这里就不仔细研究了，这里我们研究三角形的光栅化。三角形的光栅化沿用上面的画线算法，在三角形内填充像素

光栅化方法1

（延用画线算法）一般进行水平扫描（迭代y）【为了防止三角形边畸形，建议还是和画线算法一样判断xy的跨度，选择跨度大的方向进行迭代】，所以我们得知道左右边界 => 我们可以把点按照y的大小排成p0、p1、p2，那么p0～p2就是左边轮廓，p0～p1、p1～p2就是右轮廓，但是现在右轮廓是两段线段，影响代码的优美性，我们可以将三角形从中间劈开，劈成两个三角形，然后分别绘制上下三角形

```
// version 1
void triangle(Vec2i t0, Vec2i t1, Vec2i t2, TGAImage &image, TGAColor color) {
    if (t0.y==t1.y && t0.y==t2.y) return; // i dont care about degenerate triangles
    if (t0.y>t1.y) std::swap(t0, t1);
    if (t0.y>t2.y) std::swap(t0, t2);
    if (t1.y>t2.y) std::swap(t1, t2);
    int total_height = t2.y-t0.y;
    for (int i=0; i<total_height; i++) {
        bool second_half = i>t1.y-t0.y || t1.y==t0.y;
        int segment_height = second_half ? t2.y-t1.y : t1.y-t0.y;
        float alpha = (float)i/total_height;
        float beta  = (float)(i-(second_half ? t1.y-t0.y : 0))/segment_height; // be careful: with above conditions no division by zero here
        Vec2i A =               t0 + (t2-t0)*alpha;
        Vec2i B = second_half ? t1 + (t2-t1)*beta : t0 + (t1-t0)*beta;
        if (A.x>B.x) std::swap(A, B);
        for (int j=A.x; j<=B.x; j++) {
            image.set(j, t0.y+i, color); // attention, due to int casts t0.y+i != A.y
        }
    }
}
```

这里可以优化的点：（1）舍弃浮点数，用整数表达（2）x、y方向应该根据跨度进行修改

光栅化方法2

找到三角形的bounding box(x/y的最大最小值)，对bounding box中每个像素进行判断是否在三角形内，以进行着色 => 这种方法看似麻烦，但它的计算具有一定的并行性，符合GPU计算的特性

```
triangle(vec2 points[3]) {
    vec2 bbox[2] = find_bounding_box(points);
    for (each pixel in the bounding box) {
        if (inside(points, pixel)) {
            put_pixel(pixel);
        }
    }
}
```

重心坐标（二维）

$$\begin{gathered}\triangle ABC, 点P \\ \vec{AP} = u\vec{AB} + v\vec{AC} \\ \begin{cases} u\vec{AB}.x + v\vec{AC}.x + \vec{PA}.x = 0 \\ u\vec{AB}.y + v\vec{AC}.y + \vec{PA}.y = 0 \end{cases}\end{gathered} \tag{1}$$

这是一个二元一次方程组的求解问题，方法有：

- 方法1：【带入消元法】

- 方法2：【**叉乘法**】对应相乘其实我们可以写成点乘（或者矩阵相乘）的形式，点乘为0，意味着两个向量正交，那么向量a=(u,v,1)与向量b=(AB.x, AC.x, PA.x)、向量c=(AB.y, AC.y, PA.y)分别正交，那相当于a与cross(b, c)共线
    - 特殊情况：当b与c共线/平行，叉乘为0向量，意味着无解，同时b、c叉乘也失去了几何意义（垂直于平面的向量）=> bc共线意味着k_AB = k_AC = k_PA，那就是三角形不存在，或者说ABC三点共线
- 方法3：【克莱姆法则】
    - 构造方程组：刚好是两个方程两个未知数，利用行列式求解uv => **实现过程中发现，跟叉乘法原理基本一致，行列式其实就是叉乘**
    - 这里注意，这么构造相当于将u+v+w=1的约束条件融入方程组，且等式右侧不为0，保证方程只有唯一解
    - 特殊情况：当系数矩阵的行列式为0，克莱姆法则无解或者有无穷多解，但是我们知道，只要AB、AC不平行，方程一定有且有唯一解，所以行列式为0就代表了ABC三角形坍缩成一条直线，或者说AB、AC平行，确实也是如此

$$\vec{AB}\cdot u + \vec{AC}\cdot v = \vec{AP} \tag{2}$$

最后还原成重心方程：

$$\begin{aligned} u\vec{AB} + v\vec{AC} + \vec{PA} = 0 \\ u(\vec{PB} - \vec{PA}) + v(\vec{PC} - \vec{PA}) + \vec{PA} = 0 \\ (1-u-v)\vec{PA} + u\vec{PB} + v\vec{PC} = 0 \end{aligned} \tag{3}$$

```
vec3 barycentric(const array<vec2i, 3>& pts, const vec2i& P)
{
    vec3i a = {pts[1].x - pts[0].x, pts[2].x - pts[0].x, pts[0].x - P.x};
    vec3i b = {pts[1].y - pts[0].y, pts[2].y - pts[0].y, pts[0].y - P.y};

    vec3i u = a.cross(b);

    if (std::abs(u.z) < 1) // 因为后续要除以z,这里要事先剔除
        return vec3(-1, 1, 1);

    return vec3(1.f - (u.x + u.y) / (float)u.z, u.x / (float)u.z, u.y / (float)u.z);
}
/*
abs(u.z) < 1的意思是u.z=0,u.z存储的是AB×AC的结果,叉乘为0代表AB、AC共线,三角形坍缩成直线
*/
```

使用重心坐标进行光栅化：

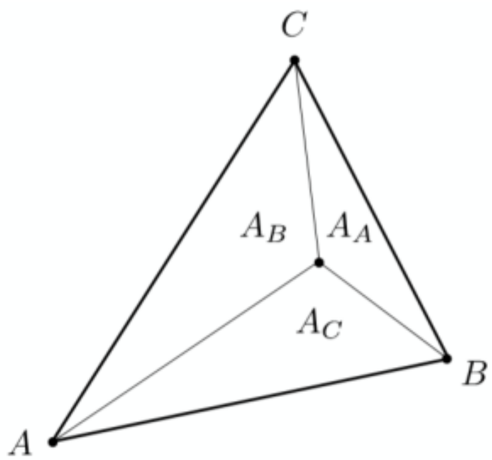
```
// version 2
void triangle(const array<vec2i, 3>& pts, TGAImage &image, TGAColor color)
{
    vec2i bboxmin(image.get_width() - 1,  image.get_height() - 1);
```

```
    vec2i bboxmax(0, 0);
    vec2i clamp(image.get_width() - 1, image.get_height() - 1);
    for (int i = 0; i < 3; i++)
    {
        bboxmin.x = std::max(0, std::min(bboxmin.x, pts[i].x));
        bboxmin.y = std::max(0, std::min(bboxmin.y, pts[i].y));
        bboxmax.x = std::min(clamp.x, std::max(bboxmax.x, pts[i].x));
        bboxmax.y = std::min(clamp.y, std::max(bboxmax.y, pts[i].y));
    }
    vec2i P;
    for (P.x = bboxmin.x; P.x <= bboxmax.x; P.x++)
    {
        for (P.y = bboxmin.y; P.y <= bboxmax.y; P.y++)
        {
            vec3 bc_screen = barycentric(pts, P);
            if (bc_screen.x < 0 || bc_screen.y < 0 || bc_screen.z < 0)
                continue;
            image.set(P.x, P.y, color);
        }
    }
}
```

拓展：三维重心坐标

三维我们通过一种更简单的方式进行计算：**面积法**

$$\alpha\vec{PA}+\beta\vec{PB}+\gamma\vec{PC}=0 \tag{4}$$

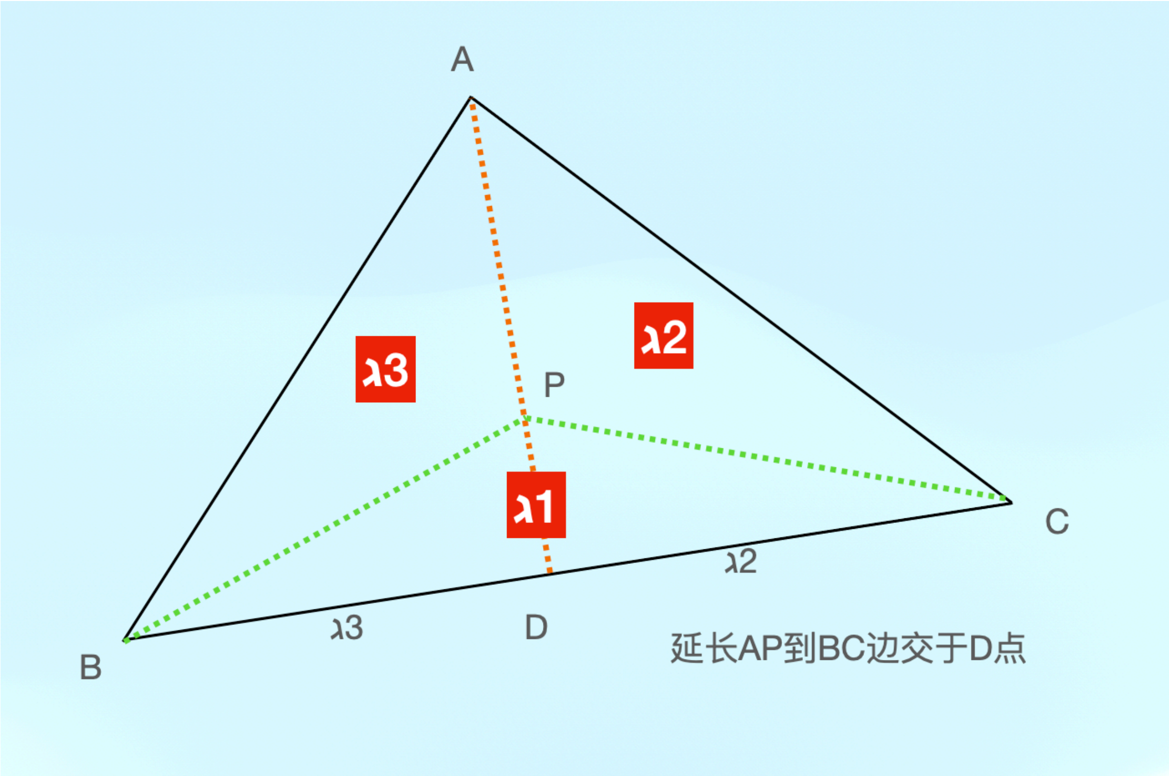


$$\alpha=\frac{A_A}{A_A+A_B+A_C}$$
$$\beta=\frac{A_B}{A_A+A_B+A_C}$$
$$\gamma=\frac{A_C}{A_A+A_B+A_C}$$

重心坐标αβγ其实代表了三个子三角形的面积：我们直接计算面积即可计算出重心坐标

$$\frac{\triangle PBC}{\triangle ABC}=\alpha \quad \frac{\triangle PCA}{\triangle ABC}=\beta \quad \frac{\triangle PAB}{\triangle ABC}=\gamma \tag{5}$$

证明：重心坐标就是面积坐标



假设三个子三角形的面积之比为：ג1:ג2:ג3，那么ג1+ג2+ג3=1。因为ΔABP和ΔACP有共同边AP，所以BD:BC=ג3:ג2，所以D点坐标：

$$D = \frac{\lambda_2 B + \lambda_3 C}{\lambda_2 + \lambda_3} \tag{6}$$

又因为ΔPBC和ΔABC有共同边BC，所以PD:AD=ג1:(ג1+ג2+ג3)，或者说DP:AP=ג1:(ג2+ג3)，所以P点坐标：

$$\begin{aligned} P &= \frac{\lambda_1 A + (\lambda_2 + \lambda_3) D}{\lambda_1 + \lambda_2 + \lambda_3} \\ &= \lambda_1 A + \lambda_2 B + \lambda_3 C \end{aligned} \tag{7}$$

具体代码：三角形面积：通过两个向量叉乘的模再除以2即可 -- 因为向量叉乘的模是以这两边为平行四边形的面积

```
bool isPointInTriangle3D(const vec3& p, const std::array<vec3, 3>& pts, vec3& bary)
{
    // use subtriangle area to calculate barycentric coordinates
    bary = {-1.0, 1.0, 1.0};
    float epsilon = 0.0001;

    vec3 a = pts[0];
    vec3 b = pts[1];
    vec3 c = pts[2];

    vec3 ab = b - a;
    vec3 bc = c - b;
    vec3 ca = a - c;

    float S = ab.cross(ca).norm() / 2.0;
```

```
    if(abs(S) < epsilon)
        return false;

    vec3 pbc = (p - b).cross(bc);
    vec3 pca = (p - c).cross(ca);
    vec3 pab = (p - a).cross(ab);

    if(pbc.dot(pca) * pbc.dot(pab) < 0.0) // 判断三个向量是否共线
        return false;

    float S_PBC = pbc.norm() / 2.0;
    float S_PCA = pca.norm() / 2.0;
    float S_PAB = pab.norm() / 2.0;

    bary.x = S_PBC / S;
    bary.y = S_PCA / S;
    bary.z = S_PAB / S;

    if(abs(bary.x + bary.y + bary.z - 1.0) > epsilon)
        return false;
    return true;
}
```

拓展2：三维空间中射线与三角形是否有交点

> https://blog.csdn.net/charlee44/article/details/104348131

克莱姆法则：假设线性方程组的系数行列式不为0，则线性方程有且仅有唯一解：**【解决多元一次方程组】**

PS：行列式为0，并不代表方程一定无解 => 方程可能有无穷解，或者无解

一、克莱姆法则

定理二(克莱姆法则) 设线性方程组

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

的系数行列式

$$D = \begin{vmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{vmatrix} \neq 0$$

则该线性方程组有且仅有唯一解:

$$x_1 = \frac{D_1}{D}, x_2 = \frac{D_2}{D}, \cdots, x_n = \frac{D_n}{D}$$

其中D_j (j=1,2,...,n)是把系数行列式D中第j列的元素用常数项$b_1, b_2, \cdots, b_n$代替后得到的n阶行列式. 即

$$D_j = \begin{vmatrix} a_{11} & \cdots & a_{1,j-1} & b_1 & a_{1,j+1} & \cdots & a_{1n} \\ a_{21} & \cdots & a_{2,j-1} & b_2 & a_{2,j+1} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{n1} & \cdots & a_{n,j-1} & b_n & a_{n,j+1} & \cdots & a_{nn} \end{vmatrix}$$

定理中包含三个结论:

(1)方程组有解

(2)解是唯一的

(3)解由公式 $x_j = \frac{D_j}{D}$ (j=1,2,...,n)给出

注: 用克莱姆法则解线性方程组必须有两个前提条件:

(1)未知数个数等于方程个数

(2)系数行列式$D\neq0$

背面剔除

在齐次裁剪空间中，ΔABC中AB×AC，再与观察向量点乘即可（或者直接判断z的正负值）；在齐次裁剪空间中，因为已经经过view变换，所以得到的坐标已经是面朝我们的坐标

3.z-buffer

z-buffer是二维的数组，存储最小的z值，渲染时如果当前z值小于存储的z，就覆写z-buffer和屏幕颜色缓冲

4.shader

视口变换（viewport）

```
mat4 viewport(int x, int y, int w, int h);
```

x、y、w、h分别代表屏幕上展示的范围，视口变换就是将NDC空间映射到屏幕空间上，[-1,1]=>[x,x+w]、[-1,1]=>[y,y+h]、[-1,1] => [min_depth,max_depth]（一般设置为0～1或-1～1）

⚠️这里注意：不同api对y的起始大小规定不同，所以可能需要在viewport矩阵上对y进行翻转；另外对于ndc空间的定义也是不同的（影响z分量）**【这里以OpenGL为例】**

$$\begin{bmatrix} w/2 & 0 & 0 & x+w/2 \\ 0 & h/2 & 0 & y+w/2 \\ 0 & 0 & (max\_depth - min\_depth)/2 & (max\_depth + min\_depth)/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{8}$$

投影（projection）

【以OpenGL为例】n的范围是-1～1

$$M_{projection} = \begin{pmatrix} \frac{1}{r \cdot tan\frac{\alpha}{2}} & 0 & 0 & 0 \\ 0 & \frac{1}{tan\frac{\alpha}{2}} & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & 1 \\ 0 & 0 & \frac{-2nf}{f-n} & 0 \end{pmatrix} \tag{9}$$

摄像机空间（view）/ LookAt矩阵

view矩阵本身只包括旋转和平移，又因为world=>view需要求逆操作，所以我们分别对旋转和平移矩阵求逆，再将两个变化矩阵合并起来

$$V = (RT)^{-1} = T^{-1}R^{-1} \tag{10}$$

$$M_{view} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -T_x & -T_y & -T_z & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -u \cdot T & -v \cdot T & -w \cdot T & 1 \end{bmatrix} \tag{11}$$

渲染管线

💡**要想写好软光栅，必须深入了解渲染管线每个具体环节～**

> 图形学01-渲染管线概述 - Autumns-AA的文章 - 知乎 https://zhuanlan.zhihu.com/p/593640899

1. CPU端
    1. 数据加载进显存，设置渲染状态，调用draw call

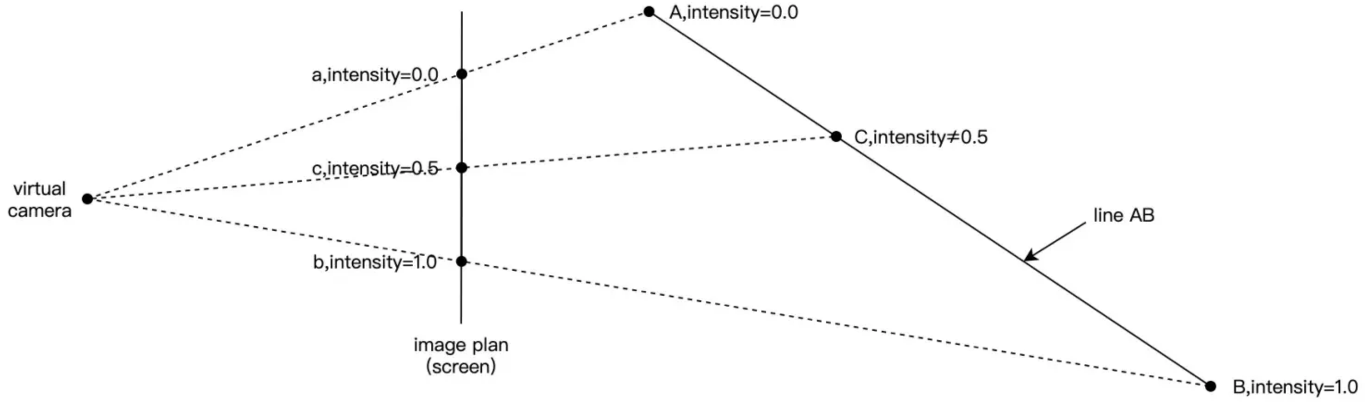
    2. **视锥体剔除**：根据摄像机参数，对模型和视锥体进行碰撞检测，剔除掉不在视锥体内的 => 常见手法：通过AABB描述模型
    3. 确定物体渲染顺序
    4. 打包数据，比如模型数据、相关Uniform参数等
2. GPU端
    1. 顶点着色器（VS）
    2. 曲面细分着色器
    3. 几何着色器（GS）
    4. **裁剪（Clipping）**：删除不在摄像机区域里的顶点和面片，可配置的阶段，是在齐次裁剪空间中进行
        - > 参考文章：https://zhuanlan.zhihu.com/p/162190576
    5. 屏幕映射
    6. 光栅化阶段
        1. 三角形设置
        2. 三角形遍历
    7. 片元着色器（PS）
    8. 逐片元操作

透视矫正

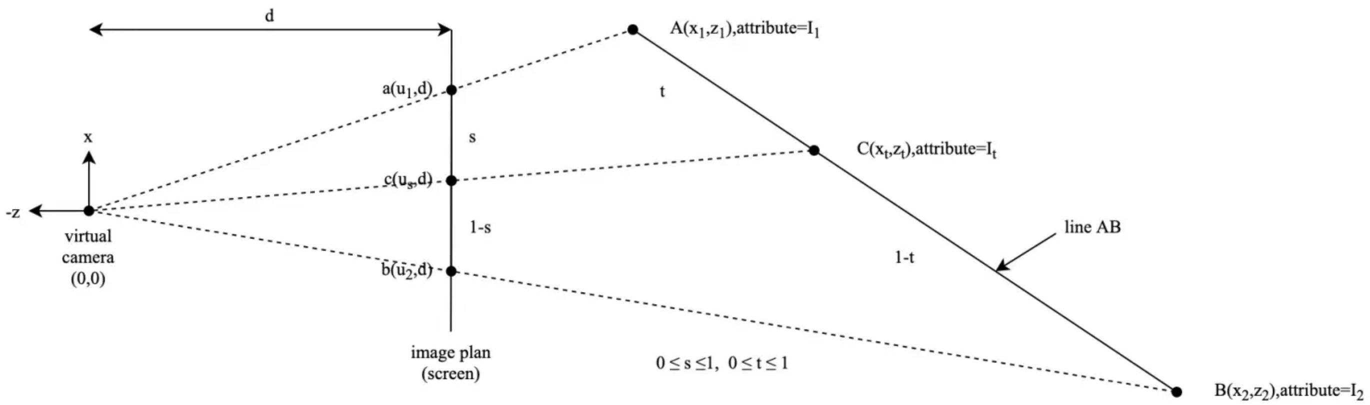
> 参考论文：https://www.comp.nus.edu.sg/~lowkl/publications/lowk_persp_interp_techrep.pdf

进行透视投影时，影响了三个点之间的位置关系，导致插值错误，产生透视扭曲

【例】：三个点在世界空间中可能C并不是AB的中点，但是经过投影后（vertex shader）在齐次裁剪空间/NDC空间站中，C成为了AB的中点，如下图👇



如果用插值的uv来采样纹理，可能导致纹理扭曲 =》 **结论：直接在屏幕空间中对属性做<u>线性插值</u>可能存在透视扭曲的现象**



透视扭曲发生的根本原因是：透视投影矩阵对z的变换不是线性的

根据推导我们知道：

$$\frac{I_t}{Z_t} = \frac{I_1}{Z_1} + s(\frac{I_2}{Z_2} - \frac{I_1}{Z_1}) \tag{12}$$

透视校正插值：使用线性的深度值和非线性空间下的插值系数s来计算出线性空间下的插值系数t，通过真正的t来进行插值

那么硬件实际是怎么悄无声息地为我们进行透视校正的?

- 在我们实现投影时，我们会将深度值保存在w分量中，这本身是为了进行透视除法。但是，当进行完透视除法后，这个分量并没有被丢弃，**而是保留下来进行后续的透视校正插值**
- 插值发生在光栅化阶段，这个阶段，硬件利用我们之前保存的线性空间中z值，对各属性进行插值 => 也就是说，只有经过光栅化的属性才会被自动进行透视校正插值
- 如果要在像素着色器中手动计算插值系数，就会出现问题 => 手动计算，需要手动进行透视校正