math_imple 相关问题记录

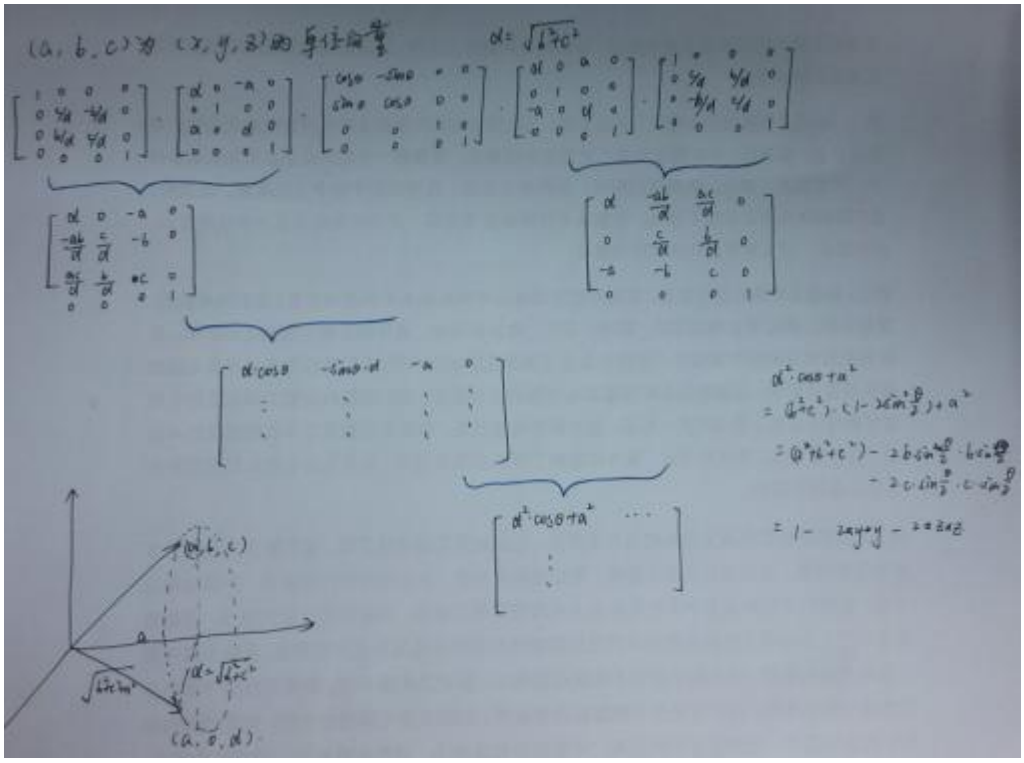
1.旋转所用公式的证明

```
void matrix_set_rotate(matrix_t *m,float x,float y,float z,float theta)
{
    //m矩阵为模型视图矩阵
    float qsin=(float)sin(theta*0.5);
    float qcos=(float)cos(theta*0.5);
    vector_t vec={x,y,z,1.0};//(0,0,0)->(x,y,z)为旋转轴
    float w=qcos;
    vector_normalize(&vec);
    x=vec.x*qsin;
    y=vec.y*qsin;
    z=vec.z*qsin;
    //因为旋转轴有点就在原点，因此无需平移了，直接旋转两次与z轴重合，旋转theta角，再反向乘上逆矩阵求回来
    m->m[0][0]=1-2*y*y-2*z*z;//只验证了该点
    m->m[1][0]=2*x*y-2*w*z;
    m->m[2][0]=2*x*z+2*w*y;
    m->m[0][1]=2*x*y+2*w*z;
    m->m[1][1]=1-2*x*x-2*z*z;
    m->m[2][1]=2*y*z-2*w*x;
    m->m[0][2]=2*x*z-2*w*y;
    m->m[1][2]=2*y*z+2*w*x;
    m->m[2][2]=1-2*x*x-2*y*y;
    m->m[0][3]=m->m[1][3]=m->m[2][3]=0.0;
    m->m[3][0]=m->m[3][1]=m->m[3][2]=0.0;
    m->m[3][3]=1.0;
}
```

其中 m 为模式视图矩阵（**<Model View Martrix>用来将物体坐标世界坐标转到当前规定的视觉坐标用的，至于之后三维视觉坐标怎么转到二维屏幕上去，则是投影矩阵<Projection Martrix>的事儿了**），(x,y,z)为通过原点的旋转轴向量。绕一般轴旋转，模式视图矩阵，求法仍是先旋转两次使旋转轴与坐标轴重合，再绕坐标轴旋转 theta，再乘上旋转逆矩阵，将旋转轴移回来。

摄像头设置初始化模式视图矩阵，随后的旋转、平移、错切等都是对模式视图矩阵进行的操作。

推导如下：



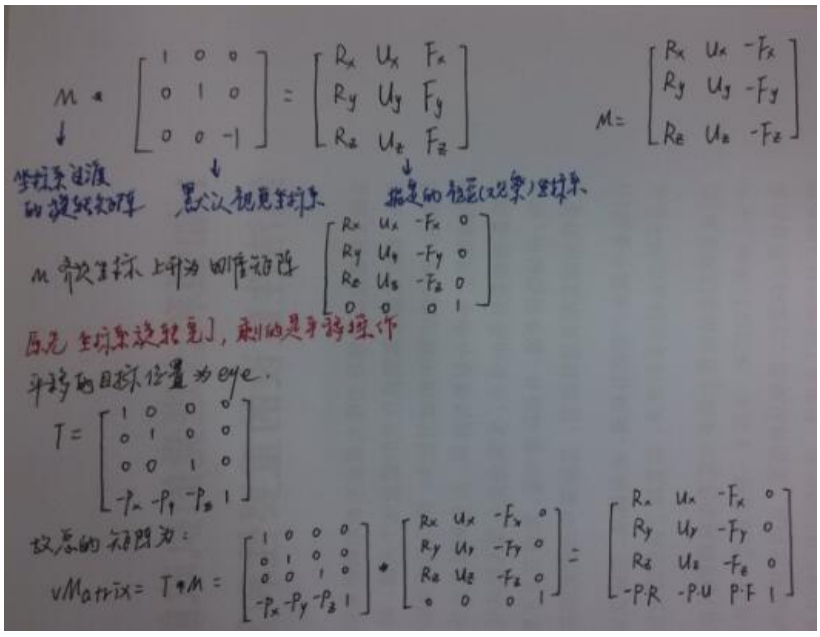
2.摄像头设置

```
void matrix_set_lookat(matrix_t *m,const vector_t *eye,const vector_t *at,const vector_t *up)
{
    //eye为相机在世界坐标的位置；
    //at为相机尽头对准的物体在世界坐标的位置；
    //up为相机向上的方向在世界坐标中的位置
    //m为模型视图矩阵，用以将物体中的物体坐标转化到世界坐标，再由世界坐标转化到眼坐标。
    //此处仿的是opengl，没有单独的模型矩阵或视图矩阵
    vector_t xaxis;
    vector_t yaxis;
    vector_t zaxis;
    //建立眼（观察）坐标，得出眼坐标的坐标轴单位向量
    vector_sub(&zaxis,at,eye);//设置眼坐标的z轴（面向物体方向为z轴，深度）
    vector_normalize(&zaxis);
    vector_crossproduct(&xaxis,up,&zaxis);//通过与规定的相机向上的向量叉乘，得出x轴向量
    vector_normalize(&xaxis);
    vector_crossproduct(&yaxis,&zaxis,&xaxis);//x轴与z轴得出向上的y轴向量

    m->m[0][0]=xaxis.x;
    m->m[1][0]=xaxis.y;
    m->m[2][0]=xaxis.z;
    m->m[3][0]=-vector_dotproduct(&xaxis, eye);

    m->m[0][1]=yaxis.x;
    m->m[1][1]=yaxis.y;
    m->m[2][1]=yaxis.z;
    m->m[3][1]=-vector_dotproduct(&yaxis, eye);

    m->m[0][2]=zaxis.x;
    m->m[1][2]=zaxis.y;
    m->m[2][2]=zaxis.z;
    m->m[3][2]=-vector_dotproduct(&zaxis, eye);

    m->m[0][3]=m->m[1][3]=m->m[2][3]=0.0;
    m->m[3][3]=1.0;
}
```

确认一个摄像头需要三个向量，第一个向量固定摄像头的位置，第二个向量规定摄像头的镜头的面向，第三个位置规定摄像头镜头的朝上向量。（相较于人头，先要明白头在哪个位置；然后明白脸朝向哪一侧，最后明白头是歪着的，还是摆正了的）。

初始化模式视图矩阵的公式推导在此处有：https://www.web-tinker.com/article/20177.html

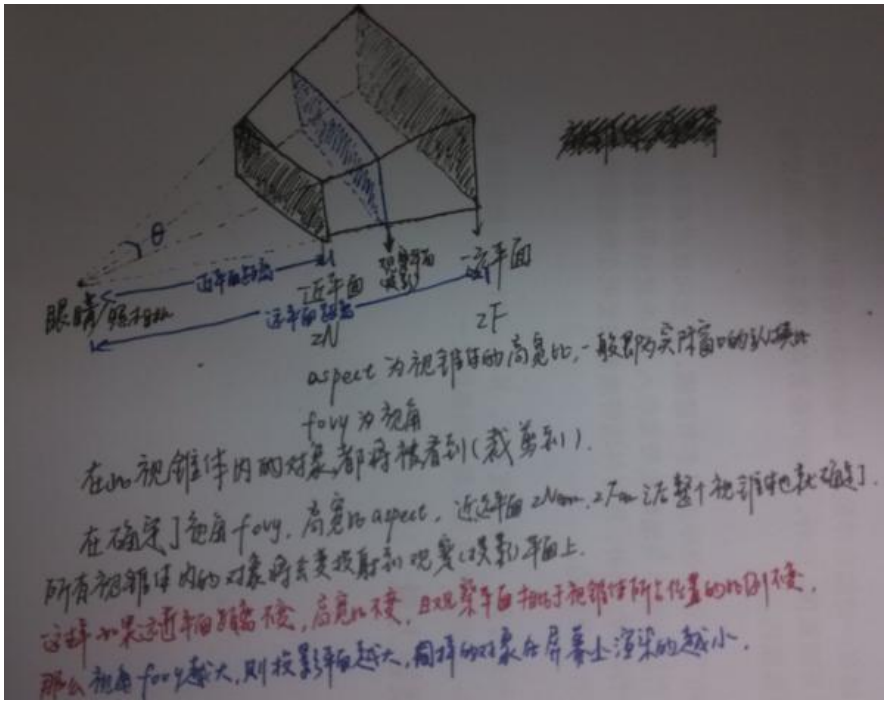
其实就是把原先默认的视图坐标系旋转平移到我们设置好的视图坐标系中。下面是用自己的语言写上一遍：



3.投射矩阵设置

```
void matrix_set_perspective(matrix_t *m,float fovy,float aspect,float zn,float zf)
{
    //
    float fax=1.0/(float)tan(fovy*0.5);
    matrix_set_zero(m);
    m->m[0][0]=(float)(fax/aspect);
    m->m[1][1]=(float)(fax);
    m->m[2][2]=zf/(zf-zn);
    m->m[3][2]=-zn*zf/(zf-zn);
    m->m[2][3]=1;
}
```

m 为投射矩阵将三维的视觉坐标中的坐标转化为二维的屏幕坐标。



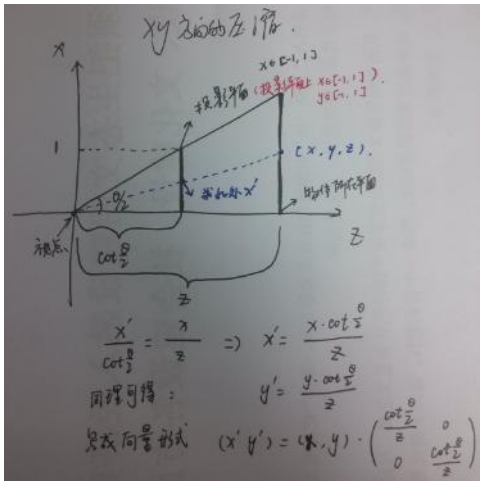
投射矩阵推导：

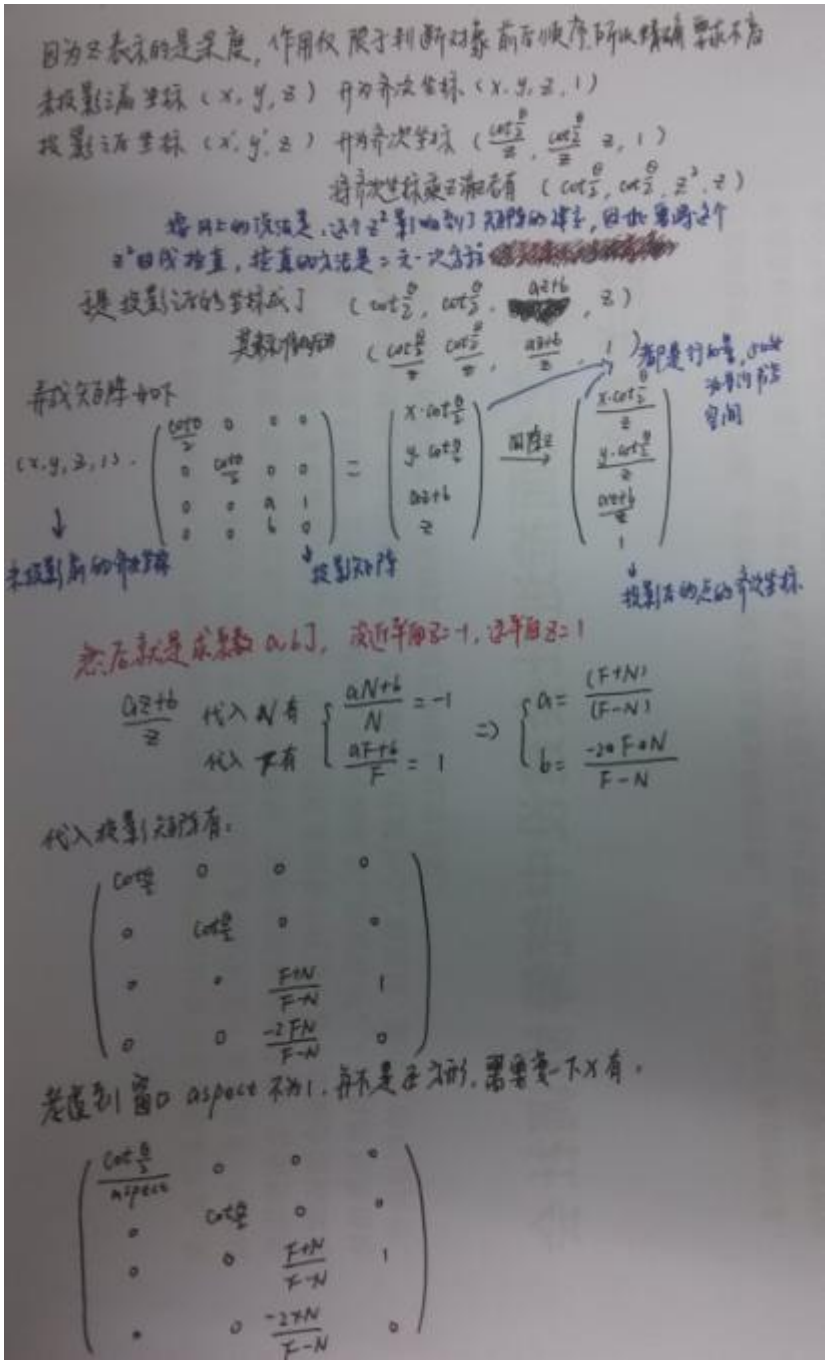
https://www.web-tinker.com/article/20157.html

http://blog.csdn.net/popy007/article/details/1797121

下面是用自己的话组织一遍：

x,y 方向的压缩





上面的程序编写中省去了+N 和*2，是因为 D3DXMatrixPerspectiveFovLH 的 CVV 中 z 的取值范围是[0 , 1]，近平面时 z 的取值是 0 的缘故。

axis_imple 相关问题记录

1.简单的线性插值

基本思想是：给一个 x 属于[a, b]，找到 y 属于[c, d]，使得 x 与 a 的距离比上 ab 长度所得到的比例，等于 y 与 c 的距离比上 cd 长度所得到的比例，用数学表达式描述很容易理解：

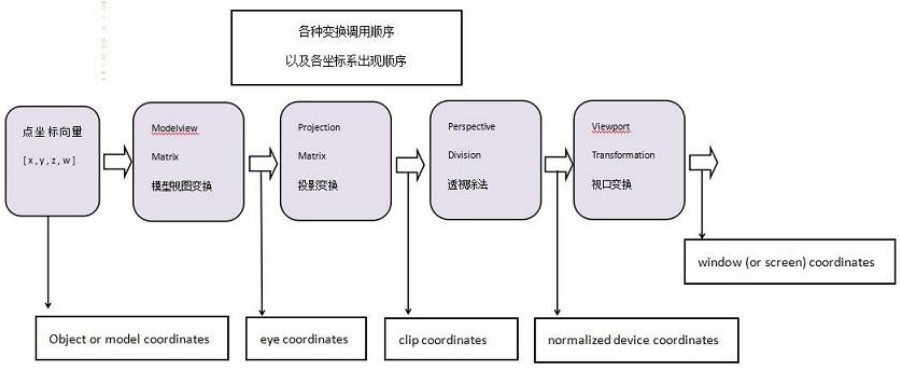
$$\frac{x-a}{b-a}=\frac{y-c}{d-c}$$

上面透视投影计算 xy 压缩后的坐标时，有用到。

2.坐标变换过程

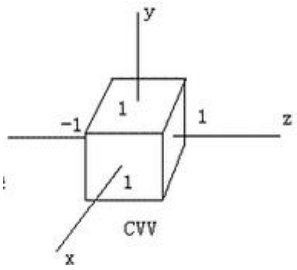
这篇 opengl 坐标变换写的很好：

http://www.360doc.com/content/14/1028/10/19175681_420522107.shtml



a.点是世界坐标系中的点，通过 word*view 矩阵，将点转变到观察坐标系中；

b.再将观察坐标系中的点，通过乘以 projection 矩阵，压缩转变到视锥体中的坐标；



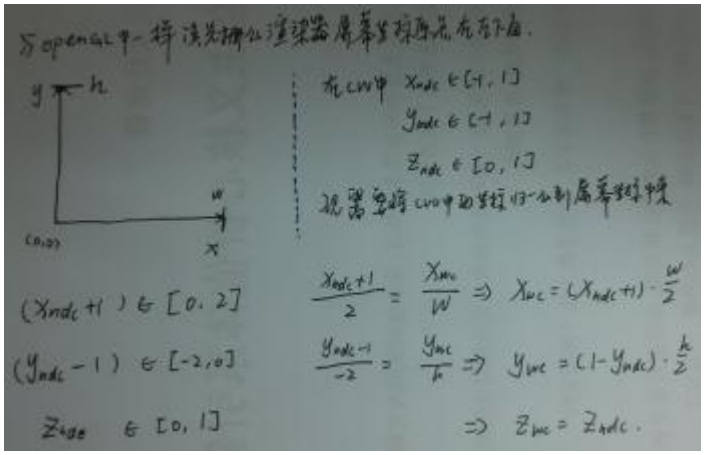
CVV：一个上下左右坐标取值在[-1 , 1]的正方体，在这个光栅化渲染器中，CVV 的 z 取值为[0,1]

c.然后用透视除法（大家都除个 w，将最后一项变为 1，就像上节中投射设置矩阵做的一样 CVV）方便裁剪（是否在视景体中）；

```
//归一化得到屏幕坐标
void transform_homogenize(const transform_t *ts,vector_t *y,const vector_t *x)
{
    float rhw=1.0/x->w;
    y->x=(x->x*rhw+1.0)*ts->w*0.5;
    y->y=(1.0-x->y*rhw)*ts->h*0.5;
    y->z=x->z*rhw;
    y->w=1.0;
}
```

在这个光栅化渲染器中，将归一化设备坐标（就是同除 w）和归一化设备坐标转屏幕坐标给拧在一块儿了

下面是归一化得到屏幕坐标的公式推导：



常用的归一化

线性函数转换：y=(x-MinValue)/(MaxValue-MinValue)。

对数函数转换：y=log10(x)，说明：以 10 为底的对数函数转换。

反正切函数转换：y=atan(x)*2/PI

d.最后，经过变换的坐标和屏幕像素之间必须建立对应关系。这个过程叫视口变换（所谓的视口变换就是规定屏幕上显示的场景的范围和尺寸）。

geometry_imple 相关问题记录

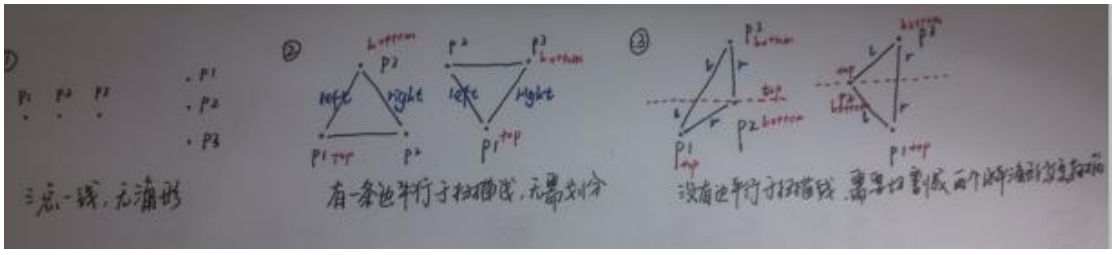
1.几何结构体成员变量注释

```
typedef struct
{
    float r;
    float g;
    float b;
} color_t;//RGB颜色
typedef struct
{
    float u;
    float v;
} texcoord_t;//纹理坐标
typedef struct
{
    point_t pos;
    texcoord_t tc;
    color_t color;
    float rhw;//rhw是做甚用？？？？？？？
} vertex_t;//顶点
typedef struct
{
    vertex_t v;//保存Y=y与left、right的交点
    vertex_t v1;
    vertex_t v2;
} edge_t;//边
typedef struct
{
    float top;//方便水平扫描线识别扫描的垂直范围
    float bottom;
    edge_t left;//左边的边
    edge_t right;//右边的边
} trapezoid_t;//梯形
typedef struct
{
    vertex_t v;//v中存放的是扫描线的浮点型，也即原来left与Y=y的交点
    vertex_t step;//存储步长
    int x;//x,y中存放的是扫描线起点的整型，因为像素是整型的
    int y;
    int w;//扫描线的宽
} scanline_t;//扫描线
```

Rwh 还是不怎么明白做什么用的？同样相关于 rwh 的一些函数也是不知道用途。此处暂且记下

2.三角形划分，方便扫描

因为扫描线是水平的，所以三角形底边什么的，也是要求水平的，以下是对三角形的重新划分种类。（默认没有点重合）



其中情况 3 的两种形态的判断公式推导：

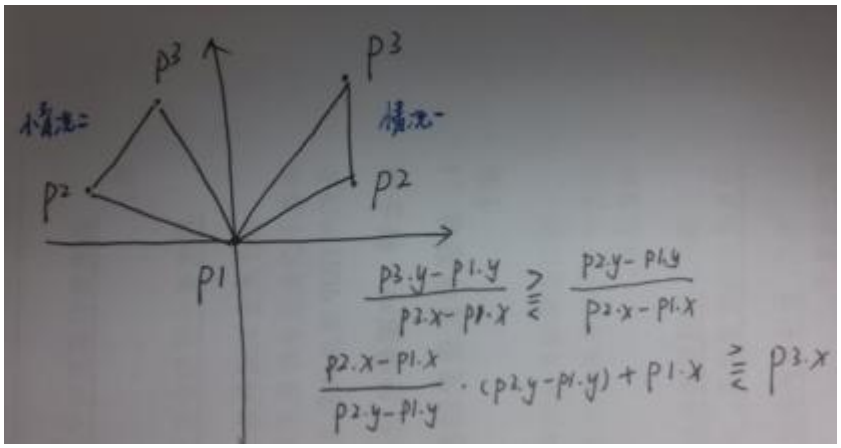
其实就是斜率之间的比较：

```
k=(p3->pos.y-p1->pos.y)/(p2->pos.y-p1->pos.y);
x=p1->pos.x+(p2->pos.x-p1->pos.x)*k;//中间点为
```

```
if(x<=p3->pos.x)
```

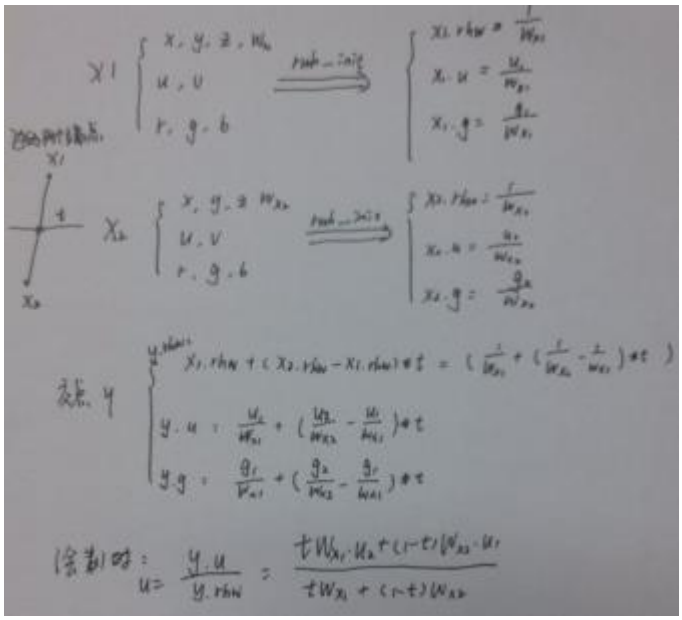
很奇怪他为什么会用这种方式计算。

P2 一定在 p3 之下的，只要比较斜率就可以知道到底是哪种形态的三角形了。



3.vertex_rhw_init

```
void vertex_rhw_init(vertex_t *v)//若点不在同一深度，颜色/纹理的渐变就需要把z方向给考虑进去
{
    float rhw=1.0f/v->pos.w;
    v->rhw=rhw;
    v->tc.u*=rhw;//纹理坐标同除点的w
    v->tc.v*=rhw;
    v->color.r*=rhw;//RGB颜色坐标同除点的w
    v->color.g*=rhw;
    v->color.b*=rhw;
}
```



如果点不在同一深度，颜色（纹理）的渐变就需要 z 方向给考虑进去了；上式中 t 是 x，y 方向的因素，w 是 z 方向的因素。

renderer_device_imple 相关问题记录

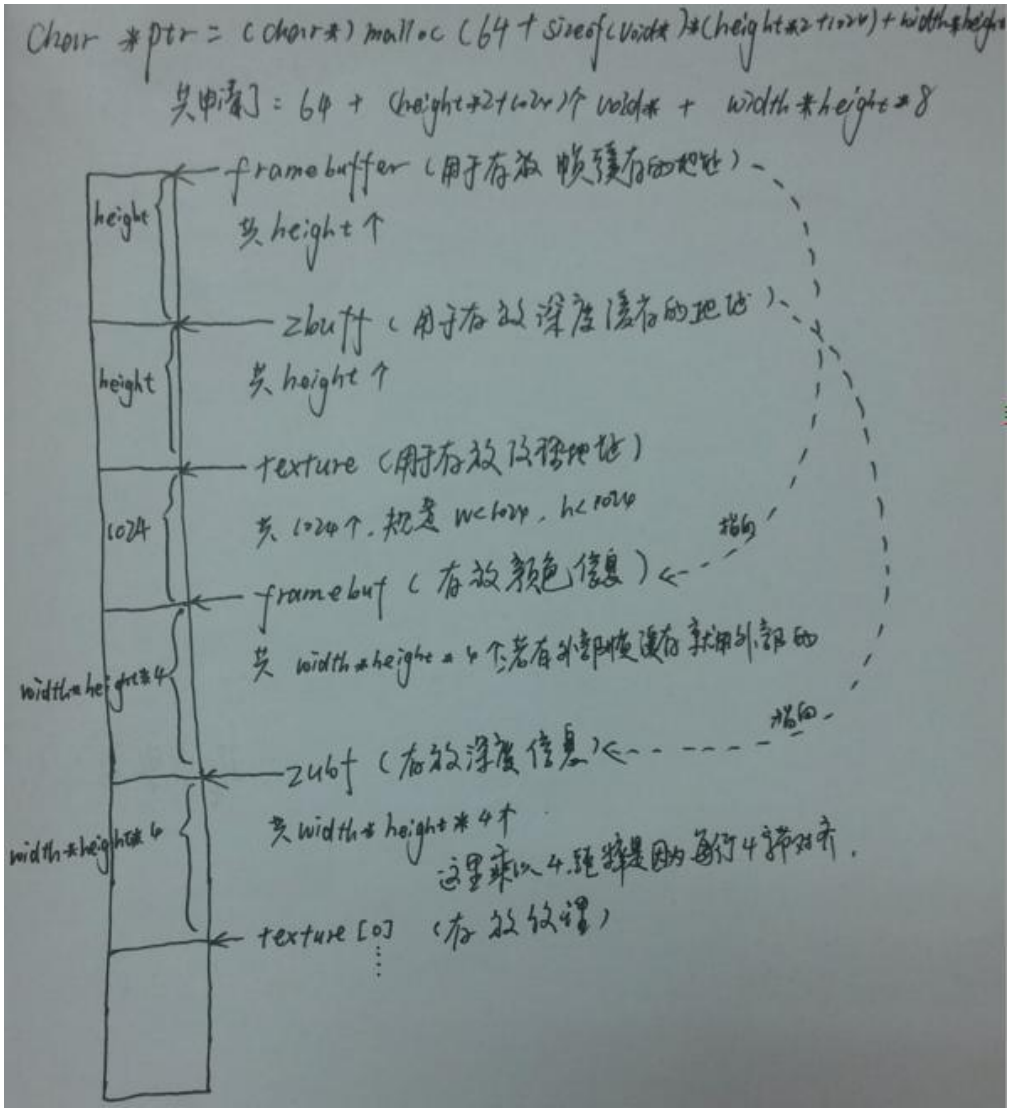
1.帧缓存

帧缓冲（framebuffer）是 Linux 为显示设备提供的一个接口，把显存抽象后的一种设备，他允许上层应用程序在图形模式下直接对显示缓冲区进行读写操作。这种操作是抽象的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由 Framebuffer 设备驱动来完成的。

在开发者看来，FrameBuffer 是一块显示缓存，往显示缓存中写入特定格式的数据就意味着向屏幕输出内容。所以说 FrameBuffer 就是一块白板。例如对于初始化为 16 位色的 FrameBuffer 来说， FrameBuffer 中的两个字节代表屏幕上一个点，从上到下，从左至右，屏幕位置与内存地址是顺序的线性关系。

帧缓存可以在系统存储器(内存)的任意位置，视频控制器通过访问帧缓存来刷新屏幕。 帧缓存也叫刷新缓存 Frame buffer 或 refresh buffer, 这里的帧(frame)是指整个屏幕范围。

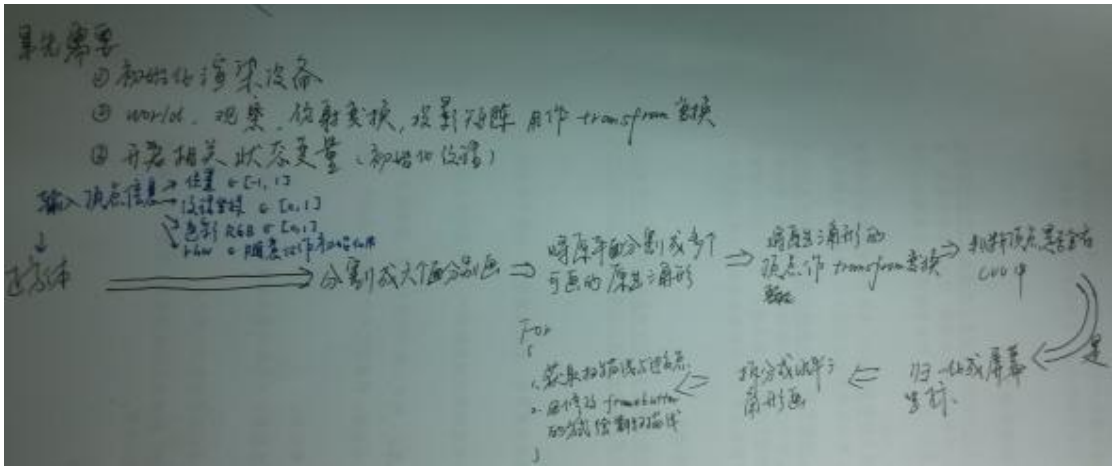
帧缓存有个地址，是在内存里。我们通过不停的向 frame buffer 中写入数据，显示控制器就自动的从 frame buffer 中取数据并显示出来。全部的图形都共享内存中同一个帧缓存。



此处是申请了一块堆内存作为帧缓存。但实际上还是使用了外部帧缓存 LPVOID ptr;在 win32 部分设置了外部帧缓存 LPVOID ptr;用以替代内部帧缓存。

总的渲染过程

1.图过程



2.各点理解

• world 矩阵，观察坐标系矩阵、仿射变换、透视投影等等最后全部按序乘起来，做成一个 transform 矩阵；这个矩阵用作将实际上的点转化到屏幕坐标（z,w 仅作深度测试用，再无其他用途）。

• 纹理、颜色、线框等设置成状态量，状态开启才会绘制相应的图形

• 绘制本质上是对帧缓存 framebuffer 起作用（直接更改 framebuffer 中对应点的值）

• 所有图形均会先分割成原生三角形，原生三角形再分割成可以绘制的水平三角形

• 绘制方法就是扫描线填充算法；先求交，然后从一个交点沿直线绘制点到另一个交点

• z 存储的是原 z 的线性值，由于只是用于深度测试，判断谁前谁后，所以不精确也没关系

• 输入的点位置，纹理坐标，色彩值均在-1 与 1 的范围内

• 用户输入点是在当前绘图坐标中，当前绘图坐标开始是与世界坐标重合的，在旋转，平移等仿射变换后，就不重合了，所以在变换过程中有乘上世界坐标矩阵这一环

• 用的是 w 来进行深度测试，这样清空深度缓存只要将 rhw 设置成 0 就行，但是 z 中的深度信息会被保存下来

• 要先做 rhw_init 是因为，如果点不是在同一深度上，那么颜色（纹理）的渐变过程，也要把 z 方向给考虑进去

• 没有尝试去理解 window 编程中创建窗口什么的

颜色

1.颜色存储

```
float r=scanline->v.color.r*w;
float g=scanline->v.color.g*w;
float b=scanline->v.color.b*w;
int R=(int)(r*255.0);//当初存的是r
int G=(int)(g*255.0);
int B=(int)(b*255.0);
R = CMID(R,0,255);//防止出界
G = CMID(G,0,255);
B = CMID(B,0,255);
framebuffer[x]=(R<<16)|(G<<8)|(B);
```

颜色的存储是：0xffffff，前 ff 是 R，中 ff 为 G，后 ff 为 B

2.BMP 图像文件

Opengl 读取 bmp 图像文件：

http://www.360doc.com/content/13/0913/23/12278894_314301815.shtml

Windows 所使用的 BMP 文件，在开始处有一个文件头，大小为 54 字节，保存了包括文件格式标识、颜色数、图像大小、压缩方式等信息。大小一项在宽：ox0012 和 ox0016

```
typedef struct tagBITMAPFILEHEADER
{
    UINT16 bfType; //2Bytes, 必须为"BM", 即0x424D 才是Windows位图文件
    DWORD bfSize; //4Bytes, 整个BMP文件的大小
    UINT16 bfReserved1; //2Bytes, 保留, 为0
    UINT16 bfReserved2; //2Bytes, 保留, 为0
    DWORD bfOffBits; //4Bytes, 文件起始位置到图像像素数据的字节偏移量
}BITMAPFILEHEADER;
//BMP信息头结构体定义如下:
typedef struct _tagBMP_INFOHEADER
{
    DWORD biSize; //4Bytes, INFOHEADER结构体大小, 存在其他版本I NFOHEADER, 用作区分
    LONG biWidth; //4Bytes, 图像宽度(以像素为单位)
    LONG biHeight; //4Bytes, 图像高度, +: 图像存储顺序为Bottom2Top, -: Top2Bottom
    WORD biPlanes; //2Bytes, 图像数据平面, BMP存储RGB数据, 因此总为1
    WORD biBitCount; //2Bytes, 图像像素位数
    DWORD biCompression; //4Bytes, 0: 不压缩, 1: RLE8, 2: RLE4
    DWORD biSizeImage; //4Bytes, 4字节对齐的图像数据大小
    LONG biXPelsPerMeter; //4 Bytes, 用象素/米表示的水平分辨率
    LONG biYPelsPerMeter; //4 Bytes, 用象素/米表示的垂直分辨率
    DWORD biClrUsed; //4 Bytes, 实际使用的调色板索引数, 0: 使用所有的调色板索引
    DWORD biClrImportant; //4 Bytes, 重要的调色板索引数, 0: 所有的调色板索引都重要
}BMP_INFOHEADER;
```

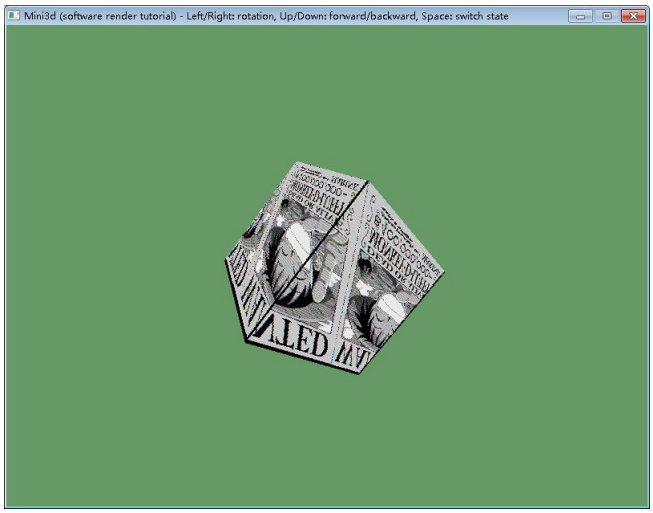
读文件大小的语句

```
int iwidth;
int iheight;
FILE *pfile=fopen(szfilename,"rb");
if(pfile==0)
{
    exit(0);
}
fseek(pfile,0x0012,SEEK_SET);
fread(&iwidth,sizeof(int),1,pfile);
fseek(pfile,0x0016,SEEK_SET);
fread(&iheight,sizeof(int),1,pfile);
```

54 字节以后，如果是 16 色或 256 色 bmp，则还有一个颜色表，但 24 位色 bmp 没有这个。24 位色 BMP 文件中，每三个字节表示一个像素颜色。BMP 文件中采用的是 BGR（和 RGB 反过来了）。像素数据量并不完全等于图像的高度乘以宽度乘以每一位像素的字节数，而是可能略大于这个值。原因是对齐，每一行像素数据的长度若不是 4 的倍数，则填充一些数据使它是 4 的倍数。所以在计算分配空间的时候要把这个算上，否则可能导致分配的内存空间长度不足，造成越界

```
int pixellength=iwidth*3;//3:BGR（bl
while(pixellength%4!=0)//读取的时候是
{
    pixellength++;
}
pixellength=pixellength*(iheight);
```

这边在编写的时候只能加载 256*256 大小的 bmp 文件




```
fseek(pfile,0x0012,SEEK_SET);
fread(&iwidth,sizeof(int),1,pfile);
fseek(pfile,0x0016,SEEK_SET);
fread(&iheight,sizeof(int),1,pfile);

int filepos=54;
for(int i=0;i<iwidth;i++)
{
    for(int j=0;j<iheight;j++)
    {
        fseek(pfile,filepos,SEEK_SET);
        fread(&B,1,1,pfile);
        fseek(pfile,filepos+1,SEEK_SET);
        fread(&G,1,1,pfile);
        fseek(pfile,filepos+2,SEEK_SET);
        fread(&R,1,1,pfile);
        filepos+=3;

        int r=(int)R;
        int g=(int)G;
        int b=(int)B;
        t[i][j]=(r<<16)|(g<<8)|b;
    }
}
```