Aegisub-3D 函数使用说明

原理简介

原理简介（线代部分）

首先该函数暂时只适用于凸多面体的建模跟渲染，原理为利用点集矩阵的矩阵变换来模拟多面体在空间中的行为（移动/旋转/变形），最后向视频所在平面进行投影，绘制出最终显示出的部分，以此在 aeg 中制作立体效果。

首先作为一个凸多面体，是由复数的顶点组成表面，再由表面拼凑起来，而最终显示的也正是这些表面，对于凸多面体的每一个表面，始终是由指定的顶点按照指定的顺序连接而成，因此凸多面体再空间中的行为本质可以看作是顶点在空间中位置的变换。

假设一个多面体共有 n 个顶点，在三维空间中，每个点由 x,y,z 三个坐标确定，将这 n 个点组合成一个 n*3 矩阵 $\begin{bmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ \cdots & \cdots & \cdots \\ x_n & y_n & z_n \end{bmatrix}$，之后多面体在空间中的行为都可以通过矩阵变换来实现，举例如下

1.位移，假设位移向量 d=（dx,dy,dz），那么任意一个顶点的变换可以表示为：

$$\{x_i,y_i,z_i\} \rightarrow \{x_i+dx,y_i+dy,z_i+dy\}$$

最终矩阵变为 $\begin{bmatrix} x_1+dx & y_1+dy & z_1+dz \\ x_2+dx & y_2+dy & z_2+dz \\ \cdots & \cdots & \cdots \\ x_n+dx & y_n+dy & z_n+dz \end{bmatrix}$

2．变形，假设以(0,0,0)为中心 x 方向缩放为 xn 倍，y 方向缩放为 yn 倍，z 方向缩放为 zn 倍，那么就等于需要乘上变换矩阵 $\begin{bmatrix} xn & 0 & 0 \\ 0 & yn & 0 \\ 0 & 0 & zn \end{bmatrix}$

3．旋转，假设绕 x 轴（y=0,z=0）旋转θ，那么只需要乘上一个变换矩阵

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}$$

（具体证明为线性代数相关内容，在此不做详细阐述）

原理简介（代码部分）

首先是矩阵的构建，使用两层 table，点集构建为

$$\mathbf{matrix}_{\mathbf{tbl}} = \{\{x_1,y_1,z_1\},\{x_2,y_2,z_2\},\dots,\{x_n,y_n,z_n\}\}$$

之后是关于矩阵的变换，由于 table 类型不能直接使用*+等符号进行操作，因此通过添加元表来解决这一问题：

对于 add 操作（即+），准备了两种方式：

如果加上一个长度为 3 的数组{a,b,c}，则会将 matrix_tbl 中每一项（即每一个点）的三个值（对应 x,y,z 坐标）分别加上数组中的三个元素，该操作等于让点集在空间中位移，位移向量为(a,b,c)
如果加上一个点集表格 matrix_tbl2，则会将两个点集合并
对于 mul（即*）准备了三种方式

如果直接乘上 num，则点集里所有项都会乘上 num，等于整个矩阵乘上变换矩阵 $\begin{bmatrix} num & 0 & 0 \\ 0 & num & 0 \\ 0 & 0 & num \end{bmatrix}$，即对应点集以(0,0,0)为中心缩放 num 倍

如果乘上一个长度为 3 的数组，等于整个矩阵乘上变换矩阵 $\begin{bmatrix} xn & 0 & 0 \\ 0 & yn & 0 \\ 0 & 0 & zn \end{bmatrix}$，即以(0,0,0)为中心分别在 x,y,z 方向上缩放 xn,yn,zn 倍

如果直接乘上一个 3*n 的矩阵 matrix，则直接乘上该矩阵

如果点集矩阵是自定义编写的，则需要通过 mat.set 添加上述元表

原理简介（绘图部分）

在 Aeg 中，最终的显示平面为 x-y(z=0)，因此最终绘图时候直接去掉 z 坐标，保留 x,y 部分绘图即可

表面信息（即每一个面由哪些点以怎样的顺序连接而成）通过 surface_tbl 储存，最后调用该表，以此再调用相应的点的坐标，最后连接而成

亮度计算则是使用平面法向量跟光源方向向量的夹角计算而得，如果是照明模式（即光照颜色不为空），则会根据亮度将表面颜色跟光照颜色进行合成；如果是阴影模式（即关照颜色为空字符串），则会根据亮度跟&H000000&（黑色）进行合成。自然光为基础亮度，根据两个向量夹角计算出的亮度加上基础亮度后即为最终的亮度。

如果透明度 alpha 不为 0，则原本位于背侧不直接显示的面也会进行计算，光强在上述计算方式的基础上乘以透明度（注意这个透明度 alpha 仅用于亮度计算，最终 fx 行的透明度标签需要自己设置）

配置部分

数据精度	code once	accuracy=1
光源方向	code once	light_vec={0,0,1}
光照颜色	code once	light_c="&HFFFFFF&"
默认底色	code once	default_color="&HFFFFFF&"
自然光	code once	n_light=-1

数据精度

控制最后输出绘图时小数位数（默认一位）

光源方向

控制光源地方向，光源属于平行光，准确来说这个向量跟光照过来地方向正好相反，比如{0,1,0}表示光源在正下方，若使用{0,0,0}则无光照

光源颜色

光照的颜色

如果设置成空字符串，则会切换成阴影模式，表面颜色会根据角度变暗

默认底色

多面体默认的表面颜色

自然光

环境亮度，最终光强是光源向量在表面法向量上取投影后加上自然光

取值范围[-1,1]

函数简介

渲染	code once	get_shapes=function(matrixs_tbl,surface,alpha) alpha=alpha or 0 local
动态	code once	move_3d=function(st,et,dt,matrix,surface,alpha,matrix_fun) if matrix_fu
添加表面图案	code once	set_surface_shape=function(matrix,surface,surface_shape) local o={0,0
立方体建模	code once	set_cube=function(x,y,z,color_tbl) cube_points=mat.set({{0,0,0},{x,0,0},
钻石建模	code once	set_diamond=function(l1,l2,d1,d2,n,c) c=c or default_color diamond_p

Matrix 点集矩阵

可以通过 set_cube/set_diamond 获得（暂时只准备了这两种模型的函数），也可以自定义，格式为{{x1,y1,z1},{x2,y2,z2},{x3,y3,z3}…}，注意自定义的点集矩阵需要使用 mat.set 添加元表（用于后续矩阵变换），如果某一个点来自表面纹理图案，则该点的 table 中还会有一个索引为 s 的值，对应表面图案的标识

Surface 表面信息

格式为{{p1,p2,p3…,color=‴,shapes={shapes1,shapes2…}}…}

p1,p2,p3…表示该表面的端点在点阵中的序数，顺序则表示端点连接顺序

color 表示颜色（默认为配置中的默认底色）

shapes 为可选项，用于向表面添加图案

格式为{si=,ei=,text=‴,color=‴,layer=,s=,intersected_or_not=}

si 表示纹理图案的第一个点在点集中的序数

ei 表示最后一个点在点集中的序数

text 为纹理图案的绘图代码

color 表示颜色（默认为配置中的默认底色）

layer 为层数

s 为标识符，用于标记纹理图案（默认 true）

intersected_or_not 若为 1 则最后会输出所在面的形状作为遮罩（详见函数 get_shapes），默认 1

*图形取交集部分需要 A3shape

矩阵变换

多面体在空间中的行为通过点集矩阵的变换来计算，经过 set_cube/set_diamond 得到的点集矩阵或者自定义的点集经过 mat.set 后（即添加上用于矩阵操作的 metatable 后）可以直接进行以下操作

matrix+{dx,d,y,dz}　平移 dx,dy,dz

matrix+{{x1,y1,z1},{x2,y2,z2},{x3,y3,z3}…}　连接两个点集

matrix*num 所有点乘 num ，等于乘变换矩阵 $\begin{bmatrix} num & 0 & 0 \\ 0 & num & 0 \\ 0 & 0 & num \end{bmatrix}$ （等比缩放）

matrix*{scx,scy,scz} 等于乘变换矩阵 $\begin{bmatrix} scx & 0 & 0 \\ 0 & scy & 0 \\ 0 & 0 & scz \end{bmatrix}$ （不等比缩放）

matrix*matrix2 乘变换矩阵 matrix2

建模部分

set_cube(x,y,z[,color_tbl])立方体建模

x/y/z 立方体尺寸

color_tbl 立方体颜色

{color1,color2,color3,color4,color5,color6}（注意只有输入六个颜色时才会生效，否则全是默认底色），按顺序依次对应 x，y，z，-x，-y，-z 六个面的颜色

输出 cube_matrix,cube_surface

set_prism(a,h,n)棱柱建模

a 底面边长

h 棱柱的高

n 底面边数

输出 prism_matrix,prism_surface

set_pyramid(a,h,n[,mode])棱锥建模

a 底面边长

h 棱柱的高

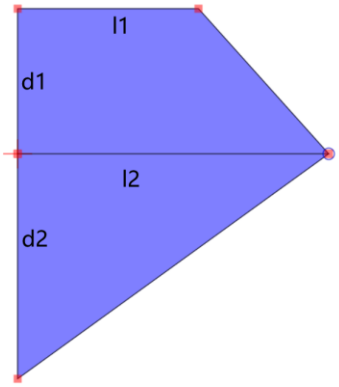
n 底面边数

mode（可选）正棱锥

输出 pyramid_matrix,pyramid_surface

set_diamond(l1,l2,d1,d2,n[,c])钻石建模

l1/l2/d1/d2 钻石尺寸



n 钻石边数

c 钻石颜色

输出 diamond_matrix,diamond_surface

表面处理

set_surface_shape 添加表面纹理图案

matrix 点集矩阵
surface 表面
surface_shape 表面图案
格式为{{si,intersected_or_not,text="",color="",layer=,name=}…}

si 为所在表面在 surface 中的序数
intersected_or_not 是否输出所在面图形（用于添加遮罩使图案限定在所在面的范围），可选，默认 1
text 绘图
color 颜色（默认为配置中的默认底色）
layer 层
name 标识（可选）

矩阵变换

mat.set 设置矩阵

将输入的点集转换为矩阵（添加元表 matrix_metatable）

mat.rotate 获取一个旋转变换矩阵

注意，这里的 x，y，z 均为定轴，不随着物体的旋转而改变
axis 旋转轴
可选"x"/"y"/"z"
theta 旋转角度
角度值，与相应的旋转轴正方向呈右手关系（右手四指沿旋转方向弯曲，拇指所指方向与旋转轴正向同向）
输出旋转变换用 3*3 矩阵

mat.s_rotate 获取一个旋转变换矩阵

该函数用于根据平面的法向量获取旋转矩阵，例如 mat.s_rotate({0,0,1},{0,1,0}) 即表示将原本法向量为(0,0,1)的面（即朝向正前方）旋转至法向量为(0,1,0)（即转至朝向正上方）
值得注意的是这个变换的具体过程是类比球坐标，先进行 y 轴矫正，将法向量 x 方向调整为 0，再进行 x 轴矫正，将法向量 y，z 方向上调整成与 vec2 一致，再进行一步 y 轴调整，完成整个旋转变换，即全过程包含绕 y 轴，绕 x 轴，绕 y 轴三步旋转
vec1 起始法向量
{x1,y1,z1}
vec2 目标法向量
{x2,y2,z2}
输出旋转变换用 3*3 矩阵

mat.l_rotate 获取绕指定旋转轴方向旋转的旋转变换矩阵

vec 旋转轴的方向向量
theta 旋转角度
角度值，跟方向向量呈右手关系（右手四指沿旋转方向弯曲，拇指所指方向与方向向量同向）

关于旋转的补充说明

函数中提供了三种旋转方式，但值得注意的是最基本的旋转变换函数，即 mat.rotate 是定轴的，即 x，y，z 三个旋转轴是始终固定的，这样的旋转变换时不满足平行四边形法则的，换而言之，矩阵在相乘的时候是不满足交换律的，通俗一点说就是这样的旋转是有顺序的，【先绕 x 转 90 再绕 y 转 90】跟【先绕 y 转 90 再绕 x 转 90】这两种操作的结果是不一样的

```
code once   cube_matrix,cube_surface=set_cube(400,400,400,color_tbl)
code once   cube_matrix=cube_matrix*mat.rotate("x",90)
code once   cube_matrix=cube_matrix*mat.rotate("y",90)
code once   cube_matrix2,cube_surface2=set_cube(400,400,400,color_tbl)
code once   cube_matrix2=cube_matrix2*mat.rotate("y",90)
code once   cube_matrix2=cube_matrix2*mat.rotate("x",90)
```

```
!maxloop(8)!!math.round(cube_matrix[j][1])!&!math.round(cube_matrix2[j][1])!
!math.round(cube_matrix[j][2])!&!math.round(cube_matrix2[j][2])!
!math.round(cube_matrix[j][3])!&!math.round(cube_matrix2[j][3])!
```

```
200&-200  -200&200  200&200
200&-200  -200&-200  -200&200
-200&-200  -200&200  200&-200
200&200  200&200  200&200
-200&-200  -200&-200  -200&-200
200&200  200&-200  -200&200
-200&200  200&200  200&-200
-200&200  200&-200  -200&-200
```

可以看到旋转顺序是有影响的，同样，在 mat.s_rotate 中，正如所描述的那样，是经历了三步这样的定轴旋转得到的，顺序分别是 y-x-y，之所以这样做一是比较方便计算，每一步的旋转轴都是固定的，不需要额外做旋转轴的变换，二是这个函数的本意是为了方便对表面的图案进行调整，将表面的图案对齐到{0,0,1}方向（即正视），这样就可以忽略 z 坐标直接对 x，y 进行修改来实现表面图案的变形，最后再调整回去即可，这样一来可以保证整个变形过程是易于控制的

```
m_fun=function(m,s,ti,tn)
    for si=1,#s do
        if s[si].shapes then
            n_vec=get_nvec({m[s[si][1]],m[s[si][2]],m[s[si][3]]},{0,0,0})
            for ssi=1,#s[si].shapes do
                for pi=s[si].shapes[ssi].si,s[si].shapes[ssi].ei do
                    local pi_m=mat.set({m[pi]})*mat.s_rotate(n_vec,{0,0,1})*mat.rotate("z",20*ti)*{(3+2*math.sin(ti/5)),(3+2*math.sin(ti/5)),1}*mat.s_rotate({0,0,1},n_vec)
                    m[pi]=pi_m[1]
                end
            end
        end
    end
    return m*mat.rotate("y",10*ti)*mat.rotate("x",-30)*(0.7+0.3*math.sin(ti/5))
end
```

比如在示例中，只需要先通过 get_nvec 确定出所在平面的法向量，之后先通过 mat.s_ratate 将图案矫正到{0,0,1}方向，就可以当作普通的二维图形进行缩放旋转及其他操作，最后再使用 mat.s_ratate 调整回去即可，因为尽管定轴旋转正常来说是收顺序影响，但是分成 x-y-x 三步就可以避免这个问题，使这一旋转变换实现可逆。

当然，如果想要让 x,y,z 轴跟物体绑定，可以在矩阵中添加{1,0,0},{0,1,0},{0,0,1}三个点，分别代表 x,y,z 轴正向的方向向量，之后使用 mat.l_rotate 进行旋转即可

```
code once   cube_matrix,cube_surface=set_cube(400,400,400,color_tbl)
code once   cube_matrix[#cube_matrix+1] = {1,0,0} cube_matrix[#cube_matrix+1] = {0,1,0} cube_matrix[#cube_matrix+1] = {0,0,1}
code once   cube_matrix=cube_matrix*mat.l_rotate(cube_matrix[#cube_matrix-1],90)
code once   cube_matrix=cube_matrix*mat.l_rotate(cube_matrix[#cube_matrix-2],90)
```

由于 x,y,z 轴正向的方向向量都被加入到矩阵中，会跟随矩阵中其他点一同做变换，这样也就实现了旋转轴的绑定

图形渲染

get_shapes 静态渲染

matrixs_tbl 点集矩阵

surface 表面

alpha 透明度

取值范围 0-1，若不为 0，则背后的面也会输出，亮度计算时会乘上 alpha

输出

{{text="",color="",clip="",layer="",s=}…}

text 为绘图

color 为颜色

clip 为遮罩，仅用于将表面图案限定在所在面显示范围内，立方体表面以及如果 intersected_or_not 设置为 0 则会返回空串（注意这个不能像其他元素那样直接使用，因为会返回空串，而且遮罩的位置也需要调整，具体使用方法请参考 示例.ass 中 cube3 部分）

layer 为层数，多面体表面在前面的 layer=1，后方的 layer=0，表面纹理绘图是在所在表面的 layer 基础上加上设置的层数（注意若所在表面位于后侧，则是减去设置的层数）

s 为标识用，多面体表面该项为 nil

move_3d 动态渲染

经过 matrix_fun 操作后的矩阵 matrix，表面 surface 和透明度 alpha 会传递给 get_shapes

st 开始时间

et 结束时间

dt 间隔时间

matrix 点集矩阵

surface 表面

alpha 透明度

matrix_fun 矩阵变换函数

function(matrix,surface,ti,tn) [code] return m end

输入矩阵 matrix，表面 surface,ti 为时间循环计数，tn 为总的时间切片数,输出变换后的矩阵 m

输出

输出为空，retime,relayer 与 maxloop 已内置，该行中可调用的变量如下：

ti 时间循环计数

tn 时间切片总数

loop_i 同一时间内图形序数

m3d={text="",color="",s=,clip="",layer=}该表详细介绍见 get_shapes