粒子系统漫游文档

——张子扬 515030910038

1. 场景建模：

**整个场景设定在一个长方体内**，skySizeX，skySizeY，skySizeZ决定了三个方向的长度。**场景使用了天空盒**，绘制六个表面并加载一套完整的太空盒贴图，每个表面如下：

glPushMatrix();

glTranslatef(0.0f, 1.0f\*skySizeY / 2.0f, 0.0f);

glRotatef(270, 1, 0, 0);

glScalef(skySizeX, skySizeZ, 1);

drawRect(texture[5]);

glPopMatrix();

**在场景中心建造了一个房子，由屋顶，表面和烟囱构成，整个房子模型由两个Obj文件导入**，分别是房子的Obj模型和烟囱的Obj模型。**房子的表面添加窗和门作为装饰。**

为方便导入模型，创建了Obj类和Material类，Obj类持有一个Obj所有必须的信息，Material则保存一个obj的所有材质信息。

模型导入的方法是调用ReadObj函数，输入一个Obj文件的路径，并输出一个保存有obj名字和对应Obj结构体的map和所有obj名字的set，以及obj名字与其Material类的map。**读取时每行读取，然后根据每行第一个单词决定操作**，如v,vt,vn,o(g),f进行条件判断，这部分基本与上次相同。不同的是这次额外添加了对mtl文件的支持，实现了保存材质名字，与读取mtl文件的ReadMtl函数，实现方式与ReadObj类似，仍然是读取每行第一个单词，进行对应操作，结果保存在材质map中。他还负责载入纹理贴图，并保存它的句柄。

载入纹理贴图的方式参考了网上的方法，fread读取图片文件，并将结果保存在一个GLuByte数组中：

if ((pFile = fopen(file\_name, "rb")) == NULL)

if (fread(pixels, total\_bytes, 1, pFile) <= 0)

调用glTexImage2D形成纹理，并使用glGenTextures glBindTexture绑定并生成句柄：

glGenTextures(1, &texture\_ID);

glBindTexture(GL\_TEXTURE\_2D, texture\_ID);

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, width, height, 0, GL\_BGR\_EXT, GL\_UNSIGNED\_BYTE, pixels);

调用时，形如：

texture[0] = readTexture("door.bmp");

需要加载时：

glBindTexture(GL\_TEXTURE\_2D, texture[0]);

模型渲染的方法是每次渲染都调用LoadObj函数，这个函数**输入所有obj的信息，并遍历所有obj，循环所有face**，用glNormal3f，glTexCoord2f，glVertex3f三个函数绘制所有顶点，法向量和纹理，如：

while (iter != temp.faces.end()){

glNormal3f(temp.normals[\*(iter + 2) - 1].x, temp.normals[\*(iter + 2) - 1].y, temp.normals[\*(iter + 2) - 1].z);

glTexCoord2f(temp.texcoords[\*(iter + 1) - 1].first, temp.texcoords[\*(iter + 1) - 1].second);

glVertex3f(temp.vertexs[\*iter - 1].x, temp.vertexs[\*iter - 1].y, temp.vertexs[\*iter - 1].z);

iter += 3;

}

这部分与上次相似，这次多提供了纹理绘制。

加载房子时，**使用了显示列表**即

glNewList(lid, GL\_COMPILE);

glPushMatrix();

glTranslatef(0, -1.0f\*skySizeY / 2.0f - 2.0f, -1.0f\*skySizeZ / 2.0f + 60.0f);

glScalef(3.0f, 3.0f, 3.0f);

glRotatef(90, 0, 1, 0);

loadObj(housename, housemap, house\_matname);

glPopMatrix();

二．粒子系统

创建了粒子发射器类Emitter和粒子类Partical。

1.Emitter类

Emitter类持有一个Partical类指针的数组指针，管理所有它发射出的粒子，发射速率speed决定了这个数组的大小。发射器的位置由六个float变量决定：x1, y1, x2, y2, z1, z2，使粒子初始坐标在(x1,x2),(y1,y2),(z1,z2)的范围内随机选择，例如：

p[i]->place.x = 1.0f\*place\_x / speed\*(x2 - x1) + x1;

**使用线性插值的方法决定了粒子的初始位置**。

为了使粒子系统更一般化，Emitter类持有两个**函数指针**，分别是：

Particle\* (\*initFunc)();

bool(\*deadFunc)(Particle\*);

它们分别指向初始化粒子各参数（运动速度，方向，加速度）的函数，和判定粒子死亡条件的函数。**不同的粒子系统，向其传入不同函数即可构造出不同的粒子物理运动效果**。

Emitter的构造函数接收参数，包括发射器位置，以及发射速率。之后需要调用emit函数，传入上述两个函数指针，决定粒子运动形式。Emitter还有update函数和show函数，这两个函数循环遍历所有它们管理的粒子，调用每个粒子自己的update和show函数。Emitter的update函数还会检查粒子是否已经死亡，条件判断为：

if (p[i]->life < 0 && isInfinite || deadFunc(p[i]))

**如果死亡，会调用设定好的initFunc并初始化粒子位置**，实现**无限个粒子发射**的效果。当然还支持**有限发射**的情况，只要在死亡后将管理粒子的数组对应位置设为NULL。**这两种更新方式可以在执行时切换。**

**发射器位置允许在运行时改变**，调用：

changeLoc(float \_x1, float \_x2, float \_y1, float \_y2, float \_z1, float \_z2);

在改变光源粒子效果时会用到。

2.Partical类

粒子类的属性有：

bool has\_tex;//纹理或颜色

vec place;//位置

vec size;//大小

vec speed;// 速度

vec acc;//加速度

vec angle;//角度

vec color;//颜色

unsigned int texture;//纹理

float life;//生命

**它们几乎全部在Partical类的构造函数初始化**。其中vec是为方便定义的一个三维向量类。has\_tex表明例子是否有贴图，没有会使用一种颜色绘制（实际上，所有粒子都分配了贴图）；place是

粒子的当前位置，每次update会更新位置；texture是粒子贴图纹理的句柄；life是粒子的生命周期，每一次update，life会递减一个值，**这个值只是保证粒子最终会消失，减轻系统负荷，一般情况下，判断粒子死亡的都是设定好的deadFunc。**

Partical类的show，update和draw函数分别显示，更新与绘制粒子。show函数如下：

void Particle::show(){

glDepthMask(GL\_FALSE);

glPushMatrix();

glTranslatef(place.x, place.y, place.z);

glRotatef(angle.x, 1, 0, 0);

glRotatef(angle.y, 0, 1, 0);

glRotatef(angle.z, 0, 0, 1);

glScalef(size.x, size.y, size.z);

if (has\_tex)

draw(texture);

else

draw();

glPopMatrix();

glDepthMask(GL\_TRUE);

}

首先设置glDepthMask(GL\_FALSE)**使得深度缓冲区只读，这样的结果是粒子最后会半透明，粒子加载上方形贴图后，只会显示不透明的形状部分，视觉效果很好；**接下来按照粒子位置，角度，尺寸**绘制粒子方片**，调用draw函数：

glEnable(GL\_TEXTURE\_2D);

glBindTexture(GL\_TEXTURE\_2D, texture);

const GLfloat x1 = -0.5, x2 = 0.5;

const GLfloat y1 = -0.5, y2 = 0.5;

const GLfloat point[4][2] = { { x1,y1 },{ x2,y1 },{ x2,y2 },{ x1,y2 } };

int dir[4][2] = { { 0,0 },{ 1,0 },{ 1,1 },{ 0,1 } };

glBegin(GL\_QUADS);

for (int i = 0; i < 4; i++) {

glTexCoord2iv(dir[i]);

glVertex2fv(point[i]);

}

glEnd();

glDisable(GL\_TEXTURE\_2D);

**因此所有的粒子实际上都是方片，然后贴上方形贴图，由于之前设置半透明，就好像按照贴图上的形状绘制了粒子。这种方法避免了给粒子建模（创建Obj文件）或复杂的绘制，性能非常好，适用于大量的粒子。**Update函数更新了粒子位置和速度，并减少粒子寿命：

place.y += speed.y;

speed.y += acc.y;

life -= 0.1f;

**可见一般的物理模型是匀加速运动的模型。**

允许在运行时更换粒子外观，本质上就是替换贴图文件，只要在初始化时加载所有可用贴图，生成纹理句柄，然后调用：

setTexture(GLuint texture)

即可，之后draw函数自然使用新的纹理。

3.粒子系统实例

实现了三个粒子系统：**烟雾，雪花，光照。**

烟雾选用点贴图，在烟囱位置设置发射器，并向上发射，有左右扰动，即在initFunc中设置速度和加速度为：

float speed[] = { float(rand() % 10 - 4) / 1600, float(rand() % 10 - 4) / 800, float(rand() % 10 - 4) / 1600 };

float acc[] = { 1.0f\*(rand() % 3 - 1) / 5000, 0.2f \*((rand() % 5)+1)/ 2000+4.9f/2000 ,1.0f\*(rand() % 3 - 1) / 5000 };

死亡函数是：

bool smoke\_dead(Particle\* p){

const vec& place = p->getPlace();

if (place.y > 1.0f\*skySizeY) {

return true;

}

return false;

}

雪花选用雪花贴图，在立方体上表面（天空），向下发射，和烟雾类似。运行时，**允许交互操作雪花受风的效果（即左右飘落）**。实现方法是运行时适当调节加速度。运行时，**允许改变为下雨状态**（**改变粒子模型**）。

光照粒子在光源位置球型发射粒子，规定死亡半径，实现类似灯光源的效果。运行时，**允许随光源位置改变发射器位置。**

三．光照

创建光源GL\_LIGHT1并且设置它的位置，各项反射参数。为了显示出粒子照射效果，**光源是锥形发射的，并设置了衰减函数的各项系数**。

为了运行时光源交互，所有glLightf函数在每次渲染一开始都会设置一次，方便改变后的参数生效。**光照的交互效果有：**

**改变光源位置**

**改变光照强度**

**（以上交互都可通过键盘控制）**

为了更好地体现光照，对物体设置了材质，在绘制前，调用如：

glMaterialfv(GL\_FRONT, GL\_AMBIENT, ambient);

设置材质的参数，并允许：

glLightModeli(GL\_LIGHT\_MODEL\_TWO\_SIDE, GL\_TRUE);

四．交互

交互分为两部分：粒子系统交互和粒子系统漫游。

粒子系统交互包括：改变粒子系统参数（**如下雪与下雨转化，风的影响，切换粒子无限生成或有限生成，发射器位置**），**光照交互。**都已经介绍。

粒子系统漫游包括：在场景中移动，移动视角，碰撞检测。

移动与视角移动的核心是每次渲染时调用：

gluLookAt(eye[0], eye[1], eye[2], center[0], center[1], center[2], 0, 1, 0);

其中eye数组是眼睛所在位置，center是眼睛看到的点。通过对这两个数组的调整，实现移动与视角转动。

使用GlUT的键盘响应函数，键盘控制移动，如：

case 'a': {

move(eye, center, direction::left);

break;

}

Move函数的功能形如：

x1 = eye[0], y1 = eye[2], x2 = center[0], y2 = center[2];

case front: {

eye[0] = d\*(x2 - x1) / len + x1;

eye[2] = d\*(y2 - y1) / len + y1;

center[0] = eye[0] + x2 - x1;

center[2] = eye[2] + y2 - y1;

break;

}

Len是eye和center的几何距离，将对应方向的投影距离除以集合距离，得到一个比值，用它去乘d（设为1.0f），可计算视点的最终位置。可见当没有移动视角时，在世界坐标系的那个方向移动的就是d；移动了视角，移动的距离是d在世界坐标系的那个方向的投影。这样保证了移动视角后，**始终在观察坐标系下那个方向保持相同距离的移动**。**我们要求观察点的位置和眼睛保持一个固定的距离**，因此很方便地计算出观察点位置。

视角移动也是键盘控制的，如下：

case 'j': {

rotate(eye, center, direction::left);

break;

但这是左右移动的情况，由于眼睛移动不涉及垂直方向，因此上下的视角移动是很简单的：

case 'i': {

center[1] += 2.0f;

break;

}

Rotate函数如下：

case left: {

center[0] = x1 + (x2 - x1)\*cos(alpha) + (y2 - y1)\*sin(alpha);

center[2] = y1 + (y2 - y1)\*cos(alpha) - (x2 - x1)\*sin(alpha);

break;

}

完全按照矩阵旋转变换的方法，这里alpha设为1/10pi。

最后是实现了**碰撞检测**，只实现了**移动出天空盒场景的检测**，在move或rotate调用后都会检测一次。碰撞检测只检测**AABB型包围盒**，首先将视点移动到包围盒内，如：

if (center[0] < x1) {

center[0] = x1;

}

计算眼睛和视点的距离并防止两者距离太小（调整）：

float distance = sqrt((eye[0] - center[0])\*(eye[0] - center[0]) +

(eye[2] - center[2])\*(eye[2] - center[2]));

if (distance <= 2.0f) {

eye[0] = 2.0f\*(eye[0] - center[0]) / distance + center[0];

eye[2] = 2.0f\*(eye[2] - center[2]) / distance + center[2];

}

调整眼睛位置到包围盒内：

if (eye[0] < x1) {

flag = true;

eye[0] = x1;

}

眼睛位置一旦调整，需要重新计算眼睛和视点的距离防止过近。

五．项目结构

Main函数设置（主要函数）：

glutDisplayFunc(renderScene);

glutTimerFunc(FRAMETIME \* 1000, timeFunc, 1);

glutMouseFunc(myMouse);

glutKeyboardFunc(myKeyboard);

glutCreateMenu(ProcessMenu);

setupRC();

glutReshapeFunc设置了渲染函数reshape，执行绘图工作；**glutTimerFunc设置了每FRAMETIME \* 1000（30fps）调用一次timeFunc**，它的具体功能是：

light\_emitter->infinite\_update();

glutPostRedisplay();

glutTimerFunc(FRAMETIME \* 1000, timeFunc, 1);

即：更新粒子系统的粒子属性，重新绘制整个场景，然后等FRAMETIME \* 1000(30fps)调用它自己，实现了粒子系统的动画效果；glutKeyboardFunc设置键盘交互，具体交互方法是：

**WASD 控制（眼睛）移动**

**IJKL 控制视角转动**

**TFGH 控制光源移动**

**，。 控制风向（雪花飘落方向）**

**-= 控制光源强度**

glutCreateMenu设置了右键菜单，功能有：

glutAddMenuEntry("停止烟雾效果", 0);

glutAddMenuEntry("开始烟雾效果", 1);

glutAddMenuEntry("停止下雪", 2);

glutAddMenuEntry("开始下雪", 3);

glutAddMenuEntry("下雨", 4);

glutAddMenuEntry("下雪", 5);

setupRC是初始化函数，主要包括：

设置随机数种子：srand(unsigned(time(NULL)));

加载模型：readObj(house, "obj\\house.obj", housemap, housename, house\_matname);

生成显示列表：houseList = GenTableList();

加载纹理贴图：texture[0] = readTexture("door.bmp");

初始化粒子发射器：light\_emitter = new Emitter(LIGHT\_NUM, lpos[0],lpos[0], lpos[1], lpos[1], lpos[2], lpos[2]);

设置粒子初始化函数和死亡函数：light\_emitter->emit(init\_light\_partical, light\_dead);