魁地奇桌球 设计报告

5130379012 张云翔

使用外部库：glut.h, glew.h, FreeImage.h,glfw.h

使用语言：C++

编译环境：VS2015, windows7

主要完成内容：粒子效果，场景光照与聚光灯，perlin噪声，三维纹理的读取与映射，VBO&场景地形，场景包围盒，球面映射，镜头追踪

1. 粒子效果：

1 元素结构：

particle为单一粒子属性，有其是否激活（是否可见），生命（可见度），衰减速度，颜色，位置，速度方向等信息

particleEngine为例子引擎，其中包含颜色渐变，粒子减速，粒子数组（每个引擎包含1000个粒子）

#define MAX\_PARTICLES 1000 // 定义最大的粒子数

struct particles

{

bool active; // 是否激活

float life; // 粒子生命

float fade; // 衰减速度

GLint r; // 红色值

GLint g; // 绿色值

GLint b; // 蓝色值

float x; // X 位置

float y; // Y 位置

float z; // Z 位置

float xi; // X 方向

float yi; // Y 方向

float zi; // Z 方向

float xg; // X 方向重力加速度

float yg; // Y 方向重力加速度

float zg; // Z 方向重力加速度

};

struct particleEngine

{

int particlenumber;

bool rainbow = true; // 是否为彩虹模式

float slowdown = 2.0f; // 减速粒子

float xspeed; // X方向的速度

float yspeed; // Y方向的速度

float zoom = -40.0f; // 沿Z轴缩放

GLuint loop; // 循环变量

GLuint col; // 当前的颜色

GLuint delay; // 彩虹效果延迟

GLuint vertex\_id; // 粒子系统纹理

particles particle[MAX\_PARTICLES]; // 保存1000个粒子的数组

GLint colors[12][3]; // 彩虹颜色

};

2 粒子效果实现：

void init\_particle\_ball(particleEngine &pe, ball &b)

void idle\_particle\_ball(particleEngine &pe, ball &b,bool produce)

int DrawGLScene(particleEngine &pe) // 绘制粒子

对于每个小球创建一个其独有的粒子系统，粒子的产生位置为小球位置正下方，速度是一个随机值，衰减率是一个随机值

粒子产生后运动即与小球无关，每一帧依照粒子数据更新粒子的位置与生命。到生命周期结束之后，重新产生新的粒子（与之前相同）

关于颜色变化，有一个粒子的颜色数组以及标记粒子现在颜色的变量，以及延迟值。每次更新延迟值，如果延迟值到达了一定量，则更新粒子当前颜色。

关于粒子绘制，采用三角带加速绘制，但是三角带是一个平面，随着视角转动会发现其在某些方向呈现片状。所以采用在xy，yz，xz三个平面都渲染一次以达到360度旋转看起来都是粒子而非片状。（当然这也加大了渲染时间）

粒子效果本身绘制时应关闭光照，开启混合模式（混合纹理与颜色），纹理采用中间白色，周围逐渐透明的样式。

关于深度测试，由于粒子数量庞大，如果不关闭深度测试，那么前面的粒子会遮挡后面的粒子造成奇怪的效果。同时，大量的粒子会遮挡住小球从而无法看见球体。所以需要开启深度测试只读。

1. 场景包围盒

1 元素结构：

场景包围盒采用六面立方体绘制，以2\*length作为其边长，用vertex\_id[6]存储6个面的纹理

struct skybox

{

int length;

GLuint vertex\_id[6];

};

2 实现：

绘制时应考虑场景包围盒是否能包围住整个场景与摄像机，以及最远处是否能够绘制（不够远的话需要改变场景渲染的最深深度）

由于采用的是天空盒，所以在绘制时要对每个面进行贴图，所以不能直接用glutSolidCube进行绘制，而是需要每个面单独用顶点绘制，并贴上合适的纹理与法线。（纹理需要贴对方向才能做到无缝）

需要注意的是，读取纹理时需要使用glTexParameterf(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_CLAMP\_TO\_EDGE);来使得边缘能够对齐无缝，二维纹理时s与t方向，而三维纹理需要额外加一个r方向。

天空盒可以选择是否使用光照，这里没有使用光照

1. Perlin噪声与小球纹理

1 原理：

Perlin噪声的基本原理是创建数组，数组内容代表整数点的位置。整数点的值采用随机函数生成，而整数点之间的浮点数采用线性插值的方式得到。

一维perlin噪声采用一维数组与一维线性插值

二维perlin噪声采用二维数组与二维线性插值

三维perlin噪声采用三维数组与三维线性插值

2 实现与改进：

以0.01\*x与0.01\*y作为输入，对得到的结果映射到0—255上，这样得出的perlin噪声的纹理较为无规则，这个纹理贴在了旗帜与小球表面上

如果对得出的结果乘10取余，再映射到0—255上，即可得到年轮的效果。这个纹理贴在了地形上

3 小球纹理：

使用了自动贴纹理的函数，使得纹理可以自动在小球上设置纹理映射点

1. 场景光照与聚光灯

1 原理：

glMaterialf对物体材质进行设置

gllightf对光源进行设置

GL\_LIGHT\_MODEL\_AMBIENT：整体环境光照

GL\_AMBIENT：环境光照

GL\_DIFFUSE：漫反射

GL\_SPECULAR：镜面反射

GL\_POSITION：光源位置

GL\_SPOT\_CUTOFF：聚光灯切角

GL\_SPOT\_EXPONENT：聚光灯散射程度

GL\_SPOT\_DIRECTION：聚光灯照明方向

GL\_SHININESS：镜面反射指数

GL\_EMISSION：自发光

2 聚光灯追随：

在母球上方开一盏聚光灯，方向竖直向下，每一帧更新聚光灯的xy为母球的xy

1. VBO&场景地形
2. 场景地形：

1.1结构元素：

#define TERRAINHEIGHT 1080

#define TERRAINWIDTH 1080

struct Terrain

{

double x, y, z;

float s, t, h;//纹理坐标

float norx, nory, norz;

double rx, ry, rz;

float rnorx, rnory, rnorz;

};

//地形使用TERRAINHEIGHT\*TERRAINWIDTH的点阵

extern Terrain terrain[TERRAINHEIGHT][TERRAINWIDTH];

场景地形使用一个1080\*1080的数组保存，每个元素中x,y,z代表其位置，s,t,r代表纹理坐标，norx，nory，norz代表其法向

1.2 场景实现：

场景采用二维perlin噪声生成高度，采用累加的方式造成一种陡峭的风格。

绘制时使用三角带绘制，共需要绘制TERRAINHEIGHT \* TERRAINHEIGHT\*2个面片，TERRAINHEIGHT \* TERRAINHEIGHT\*3\*2个顶点

1.3 问题：

上述的绘制方法适合在面片数量较少的情况下实现，而我采用的力度为1\*1/2，非常小，所以绘制整个地形需要1080\*1080\*6，约为233万个面片，700万个顶点，实际测试中，上述的绘制方法最多能在绘制450\*300的地形时不卡顿，在900\*600的地形上就已经非常的卡了。因此不得不采取优化措施以加速渲染

2 VBO：

2.1 VBO概述：

VBO为定点索引缓存，重复的顶点（位置、颜色、纹理、法向等都相同）不会重复计算，而只存储一次，这些信息会直接使用GPU进行渲染从而大大提升渲染速度。

当然，VBO这一特性是opengl3以后才支持的特性，需要使用到glew库

2.2 VBO使用：

在使用VBO时，我们可以知道，需要存储的顶点变成了1080\*1080，116万个，相较于700万个大大降低了，而通过GPU绘制也可以使得整个渲染不卡顿。

用GL\_ARRAY\_BUFFER来代表各个缓存数组（位置，纹理，光照）

使用glVertexAttribPointer来指定绘制信息，DrawElements进行绘制

使用VBO时需要注意的是，纹理的绑定与之前不同，需要用glActiveTexture, glBindBuffer, glClientActiveTexture, glEnableClientState, glTexCoordPointer来指定当前绘制的纹理数组与绑定中的纹理标号，单使用glEnable(GL\_TEXTURE\_3D)是没用的

1. 三维纹理的读取与映射

1 三维纹理的创建与使用

在自己创建纹理时需要注意的是数组需要4对齐，（这是读取图片时为了加速采用的改进措施），二维纹理为LENGTH\*HEIGHT\*3，3代表RGB，三维纹理为LENGTH\*HEIGHT\*DEPTH\*3，三维纹理的创建于二维纹理类似，但是在设置纹理映射时为s，t，r三个坐标，s代表x，t代表y，r代表z。

三维纹理的优势是可以直接在三维中实现空间映射，而不是用二维纹理扭曲到三维中。但是实际上直接使用二维纹理的情况更多，因为纹理大多使用图片，而图片本身只有二维信息。

本项目中的三维纹理时采用三维柏林噪声实现的，效果为年轮状。

1. 球面映射与镜头追踪

1 球面映射：

将x映射为θ，范围从0—2π

将y映射为φ，范围从0—π

由于φ的映射范围是0—π，这样cos0和cosπ之间会产生突变。所以需要采取特殊的判别方法，当φ由0变成π或者由π变成0时，需要y方向的速度反向，同时x的位置旋转π

2 镜头追踪：

三种追踪方法：一种视点在小球正上方，指向圆心，一种视点在小球后方固定位置，指向小球，一种视点在小球后方，位置随小球速度变化而变化，指向小球。

由于之前提到的cos0和cosπ之间会产生突变，所以不得不对这些镜头的up方向和location做一些处理，如第一种，需要改变在变换前的up z的位置，第二种和第三种需要改变在镜头变换前的location y位置