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

一、 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。

二、 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

三、 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

四、 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

五、 VBO&场景地形

1 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

六、 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。

七、 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