电子科技大学信息与软件工程学院

实 验 指 导 书

（实验）课程名称 图形与动画 II

电子科技大学教务处制表

目　　录

实验三 粒子系统 2

一、实验目的.....2

二、实验原理.....2

三、实验内容.....9

四、实验步骤.....9

实验三 粒子系统

一、实验目的

1. 掌握基本粒子系统的实现方法；
2. 掌握键盘事件响应的方法；
3. 能够在基本粒子系统基础上设计新的粒子动画。

二、实验原理

（一） 粒子系统框架

粒子系统基本实现步骤大致分为三步：

1. 生成（发射）粒子
2. 模拟粒子
    a) 模拟粒子运动
    b) 模拟粒子老化
    c) 模拟粒子与环境的交互（如：碰撞）
3. 渲染粒子

生成（发射）粒子通常设置粒子初始属性，包括位置、速度、加速度、生命周期、颜色、大小等信息。发射粒子的数量以及分布状态等等信息，这些通常通过可控的随机过程来处理。

接下来需要更新粒子状态。在此阶段，检查每个粒子是否已经超出了生命周期，一旦超出就将这些粒子剔出模拟过程，否则就根据粒子受力状况，更新粒子的加速度、速度以及位置信息。另外，根据渲染相关的属性更新粒子的颜色、大小等。在考虑粒子受力状态时，除了重力、摩擦力这些常见作用力外，经常需要检查与特殊三维物体的碰撞以使粒子从障碍物弹回。

在更新完成之后，需要渲染粒子，根据粒子颜色、形状、大小绘制粒子，生成一帧动画。

（二） 基本粒子系统实现

1. 定义粒子结构

首先需要定义粒子的属性，我们定义一个结构体 particles 来描述粒子属性，如图 1 所示：

- active – 指示该粒子是否消亡；
- life – 粒子的生命值；
- fade – 粒子衰老速度；

```
typedef struct                          // Create A Structure For Particle
{
    bool    active;                     // Active (Yes/No)
    float   life;                       // Particle Life
    float   fade;                       // Fade Speed

    float   r;                          // Red Value
    float   g;                          // Green Value
    float   b;                          // Blue Value

    float   x;                          // X Position
    float   y;                          // Y Position
    float   z;                          // Z Position

    float   v_x;                        // X Velocity
    float   v_y;                        // Y Velocity
    float   v_z;                        // Z Velocity

    float   a_x;                        // X Acceleration
    float   a_y;                        // Y Acceleration
    float   a_z;                        // Z Acceleration
}
particle;                               // Particles Structure
```

粒子颜色

粒子位置

粒子速度

粒子加速度

图 1

2. 创建粒子并初始化

接下来创建粒子并初始化：

1) 定义数组 particles 存储粒子信息，数组类型为定义的结构类型 particle:

```
#define MAX_PARTICLES   1000        // Number Of Particles To Create

particle particles[MAX_PARTICLES];  // Particle Array (Room For Particle Info)
```

2) 在函数 InitGL 中初始化粒子信息，如图 2 所示，其中：
   a) fade 粒子衰老速度范围为 0.003 – 0.102;
   b) 粒子初始颜色为红色；
   c) 粒子初始速度根据根据球坐标系设置，速度大小范围 0-99;
   d) 粒子受力为重力；

```
float theta, phi, rho;
for (int i = 0; i < MAX_PARTICLES;i++)              // Initials All The Particles
{
    particles[i].active = true;                              // Make All The Particles Active
    particles[i].life = 1.0f;                                // Give All The Particles Full Life
    particles[i].fade = float(rand()%100)/1000.0f+0.003f;    // Random Fade Speed

    particles[i].r = 1.0f;  // Color Red
    particles[i].g = 0.0f;
    particles[i].b = 0.0f;

    theta = (rand()%360)*PI/180;                            // Velocity
    phi = (rand()%180)*PI/180;
    rho = rand()%100;
    particles[i].v_x = float(sinf(phi)*cosf(theta)*rho);
    particles[i].v_y = float(sinf(phi)*sin(theta)*rho);
    particles[i].v_z = float(cosf(phi)*rho);

    particles[i].a_x = 0.0f;                                 // Acceleration
    particles[i].a_y = -9.8f;
    particles[i].a_z = 0.0f;
}
```

图 2

3) 球坐标系

三维空间中一点 P 的球坐标系坐标为（ρ, φ, θ），如图 3 所示：

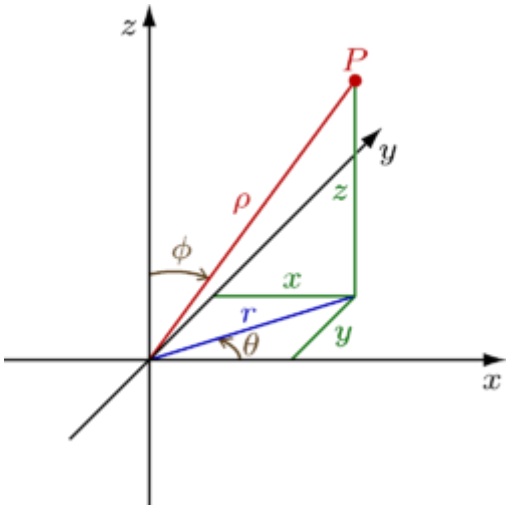

图 3

- ρ 是距离球心的距离；
- φ 是距离 z 轴的角度（称作余纬度或顶角，角度从 0 到 180°）；
- θ 是距离 x 轴的角度（与极坐标中一样）；
- 通过以下公式，可以从直角坐标变换为球坐标：

$$x=\rho\sin\phi\cos\theta$$

$$y=\rho\sin\phi\sin\theta$$

$z=\rho\cos\phi$

3. 粒子状态更新

粒子状态更新在 renderScene 函数中：

```
void renderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);     // Clear Screen And Depth Buffer
    glLoadIdentity();                                       // Reset The ModelView Matrix

    float theta, phi, rho;
    for (int i = 0; i < MAX_PARTICLES;i++)                  // Loop Through All The Particles
    {
        if (particles[i].active)                            // If The Particle Is Active
        {
            float x = particles[i].x;                       // Grab Our Particle X Position
            float y = particles[i].y;                       // Grab Our Particle Y Position
            float z = particles[i].z+zoom;                  // Particle Z Pos + Zoom

            // Draw The Particle Using Our RGB Values, Fade The Particle Based On It's Life
            glColor4f(particles[i].r,particles[i].g,particles[i].b,particles[i].life);
            glPointSize(4.0f);
            glBegin(GL_POINTS);
                glVertex3f(x,y,z);
            glEnd();

            particles[i].x+=particles[i].v_x/(slowdown*1000);    // Move On The X Axis By X Speed
            particles[i].y+=particles[i].v_y/(slowdown*1000);    // Move On The Y Axis By Y Speed
            particles[i].z+=particles[i].v_z/(slowdown*1000);    // Move On The Z Axis By Z Speed

            particles[i].v_x += particles[i].a_x;           // Take Pull On X Axis Into Account
            particles[i].v_y += particles[i].a_y;           // Take Pull On Y Axis Into Account
            particles[i].v_z += particles[i].a_z;           // Take Pull On Z Axis Into Account
            particles[i].life -= particles[i].fade;        // Reduce Particles Life By 'Fade'

            if (particles[i].life < 0.0f)                  // If Particle Is Burned Out
            {
                particles[i].life = 1.0f;                  // Give It New Life
                particles[i].fade = float(rand()%100)/1000.0f+0.003f;   // Random Fade Value
                particles[i].x = 0.0f;                     // Center On X Axis
                particles[i].y = 0.0f;                     // Center On Y Axis
                particles[i].z = 0.0f;                     // Center On Z Axis

                theta = (rand()%360)*PI/180;
                phi = (rand()%180)*PI/180;
                rho = rand()%100;

                particles[i].v_x = float(sinf(phi)*cosf(theta)*rho);    // Velocity
                particles[i].v_y = float(sinf(phi)*sin(theta)*rho);
                particles[i].v_z = float(cosf(phi)*rho);

            }
        }
    }

    glutSwapBuffers();
}
```

绘制粒子

粒子位置更新

粒子速度更新

粒子生命值更新

粒子消亡后重新生成粒子

图 4

1) 绘制粒子

采用画点方式绘制粒子，粒子大小设置：

```
glPointSize(4.0f);
```

我们利用粒子的生命值来控制粒子的透明度：

```
glColor4f(particles[i].r, particles[i].g, particles[i].b, particles[i].life);
```

2) 粒子状态更新

粒子位置更新时，我们将速度除以了（slowdown*1000），其中slowndown是个全局变量，可用来控制粒子速度大小：

```
float    slowdown=0.25f;                // Slow Down Particles
```

3) 粒子消亡

粒子消亡后，重新生成粒子。只需要给该粒子重新赋予生命值、衰老速度和运动速度。

4. 其他程序模块

- **主函数**

```
int main(int argc, char **argv) {
    // init GLUT and create window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(640,640);
    glutCreateWindow("FireWorks");

    // register callbacks
    glutDisplayFunc(renderScene);
    glutTimerFunc(50, TimerFunction, 1);
    glutReshapeFunc(changeSize);

    // initialization
    if(!InitGL()){
        printf("Initialization Failed.");
        return false;
    }

     // enter GLUT event processing loop
    glutMainLoop();

    return 0;
}
```

图 5

- **计时器**

```
void TimerFunction(int value){
    glutPostRedisplay();
    glutTimerFunc(50,TimerFunction,1);
}
```

图 6

- **初始化函数**

在函数 InitGL 中还需要加入其它的设置：

```
int InitGL(GLvoid)                                      // All Setup For OpenGL Goes Here
{
    glShadeModel(GL_SMOOTH);                            // Enable Smooth Shading
    glClearColor(0.0f,0.0f,0.0f,0.0f);                  // Black Background
    glClearDepth(1.0f);                                 // Depth Buffer Setup
    glDisable(GL_DEPTH_TEST);                           // Disable Depth Testing
    glEnable(GL_BLEND);                                 // Enable Blending
    glBlendFunc(GL_SRC_ALPHA,GL_ONE);                   // Type Of Blending To Perform
    glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);   // Really Nice Perspective Calculations
    glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);             // Really Nice Point Smoothing
```

图 7

- **窗口响应函数**

```
void changeSize(int w, int h) {

    // Prevent a divide by zero, when window is too short
    // (you cant make a window of zero width).
    if (h == 0)
        h = 1;

    float ratio =  w * 1.0 / h;

    // Use the Projection Matrix
    glMatrixMode(GL_PROJECTION);

    // Reset Matrix
    glLoadIdentity();

    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

    // Set the correct perspective.
    gluPerspective(45.0f,ratio,0.1f,200.0f);

    // Get Back to the Modelview
    glMatrixMode(GL_MODELVIEW);
}
```

图 8

（三） 添加纹理

1. 渲染原理

为了让粒子渲染效果更好，我们给粒子贴上纹理，如图 9 所示：

图 9

首先绘制正方形，然后将纹理贴在正方形上。在绘制正方形时，我们不采用传统方法，因为要绘制大量正方形速度相对较慢。OpenGL 在绘制三角形时，速度是很快的，所以我们采用绘制两个三角形的方式绘制正方形：

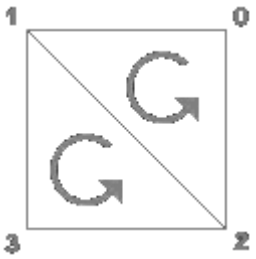

图 10

```
glBegin(GL_TRIANGLE_STRIP);          // Build Quad From A Triangle Strip
glTexCoord2d(1,1); glVertex3f(x+0.5f,y+0.5f,z); // Top Right
glTexCoord2d(0,1); glVertex3f(x-0.5f,y+0.5f,z); // Top Left
    glTexCoord2d(1,0); glVertex3f(x+0.5f,y-0.5f,z); // Bottom Right
    glTexCoord2d(0,0); glVertex3f(x-0.5f,y-0.5f,z); // Bottom Left
glEnd();
```

GL_TRIANGLE_STRIP 指定绘制三角形方式。如图 10 所示，通过指定顶点：

v0, v1, v2, v3

绘制 2 个三角形，分别由 v0, v1, v2 和 v1, v2, v3 构成。

在指定顶点序列同时，建立起顶点坐标和纹理坐标的对应关系。

2. 修改函数

1) 载入纹理

载入纹理，需要添加函数LoadGLTextures。首先定义全局变量texture：

```
GLuint  texture[1];                 // Storage For Our Particle Texture

int LoadGLTextures()                                    // Load Bitmaps And Convert To Textures
{
    /* load an image file directly as a new OpenGL texture */
    texture[0] = SOIL_load_OGL_texture
        (
        "Data/Particle.bmp",
        SOIL_LOAD_AUTO,
        SOIL_CREATE_NEW_ID,
        SOIL_FLAG_INVERT_Y
        );

    if(texture[0] == 0)
        return false;

    // Typical Texture Generation Using Data From The Bitmap
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

    return true;                                        // Return Success
}
```

图 11

2) 修改初始化函数

```
int InitGL(GLvoid)                                      // All Setup For OpenGL Goes Here
{
    if (!LoadGLTextures())                              // Jump To Texture Loading Routine ( NEW )
    {
        return false;                                   // If Texture Didn't Load Return FALSE
    }
    glShadeModel(GL_SMOOTH);                            // Enable Smooth Shading
    glClearColor(0.0f,0.0f,0.0f,0.0f);                  // Black Background
    glClearDepth(1.0f);                                 // Depth Buffer Setup
    glDisable(GL_DEPTH_TEST);                           // Disable Depth Testing
    glEnable(GL_BLEND);                                 // Enable Blending
    glBlendFunc(GL_SRC_ALPHA,GL_ONE);                   // Type Of Blending To Perform
    glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);   // Really Nice Perspective Calculations
    glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);             // Really Nice Point Smoothing
    glEnable(GL_TEXTURE_2D);                            // Enable Texture Mapping
    glBindTexture(GL_TEXTURE_2D,texture[0]);            // Select Our Texture
```

图 12

三、实验内容

1. 实现基本粒子系统；
2. 为基本粒子系统添加运动模糊效果；
3. 为基本粒子系统添加纹理贴图；
4. 在基本粒子系统基础上扩展，如生成火焰、烟花、瀑布等效果，并与环境发生交互，加入碰撞检测。

四、实验步骤

1. 明确项目需求；
2. 编写代码；
3. 编译代码；
4. 测试程序；
5. 根据测试结果对程序进行调试改进。