学号	姓名	论文规范性（10）	问题分析与调研（30）	方案创新性（20）	实验结果分析与讨论（40）	结课论文总成绩（100）
21301104	宿嘉璐	6	21	17	35	79

缺少对现有工作的梳理，缺少参考文献；实验结果分析太简单

实验报告

图形学实验报告

学　院：　软件学院

专　业：　软件工程

学生姓名：　宿嘉璐

北京交通大学

2024 年 6 月

实验报告

1. 引言

1.1 实验背景

粒子系统是一种在计算机图形学中广泛应用的技术，用于模拟和渲染复杂且随机的现象，如烟雾、火焰、爆炸和水流等。在传统的渲染方法难以精确模拟这些现象的情况下，粒子系统提供了一种高效且直观的解决方案。通过生成大量的微小粒子并对其进行独立的运动和渲染处理，可以逼真地再现这些自然现象。

1.2 实验目的

本实验的目的是通过实现和优化一个模拟水喷泉效果的粒子系统，掌握粒子系统的基本原理及其在计算机图形学中的应用，包括粒子的生成、运动、渲染和生命周期管理。

2. 理论基础

2.1 粒子系统概述

在计算机图形学中，粒子系统是一组用于模拟各种“模糊”现象（如烟雾、液体喷雾、火焰、爆炸等）的对象。每个粒子被视为一个具有位置但没有大小的点对象。通常，它们作为点精灵（使用 GL_POINTS 基本模式）进行渲染。每个粒子都有一个生命周期：它诞生，按照一套规则动画化，然后死亡。粒子可以复活并再次经历整个过程。通常，粒子不会与其他粒子相互作用或反射光线。粒子通常被渲染为一个单一的、带有透明度的、面向相机的纹理化四边形。

在本实验中，我们将实现一个相对简单的粒子系统，它具有水喷泉的外观。实验中的粒子不会被“回收”。一旦它们的生命周期结束，我们将其绘制为完全透明，使它们实际上不可见。这使得喷泉具有有限的生命周期，仿佛它只有有限的材料供应。

2.2 实验原理

2.2.1 粒子系统的基础

粒子系统是一组用于模拟如烟雾、液体喷雾、火焰、爆炸等“模糊”现象的对象。每个粒子被认为是一个没有大小的点对象。粒子有以下基本属性：

1. **位置 (Position)**：粒子的当前位置。
2. **速度 (Velocity)**：粒子的当前速度，决定了粒子的位置随时间的变化。
3. **加速度 (Acceleration)**：粒子受到的加速度，通常包括重力等恒定加速度。
4. **出生时间 (Birth Time)**：粒子生成的时间，用于计算粒子的年龄。
5. **生命周期 (Lifetime)**：粒子存在的时间，超过这个时间后粒子会消失或被回收。

2.2.2 运动学方程

粒子的运动可以通过标准的运动学方程来描述，假设粒子受到恒定加速度（如重力）的作用。粒子的运动学方程如下：

$$P(t) = P_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}t^2$$

其中：

1. P(t)是时间 t 时粒子的位置。
2. P0 是粒子的初始位置。
3. v0 是粒子的初始速度。
4. a 是粒子的加速度（如重力加速度）。
5. t 是粒子的存活时间。

2.1.3 实验具体设计

1. 初始条件

(1) 粒子的初始位置 P0 设为原点 (0,0,0)。
(2) 粒子的初始速度 v0 在一定范围内随机生成，模拟喷泉水流的散射效果。
(3) 粒子的加速度 a 设为重力加速度 (0,−9.81,0)，模拟重力的作用。

2. 动画和渲染

(1) 每个粒子的生成时间略有不同，使得粒子系统中的粒子在不同时间生成。

(2) 使用上述运动学方程计算每个粒子在其生命周期内的当前位置。

(3) 粒子在其生命周期结束时变得完全透明，以实现有限材料供应的效果。

(4) 将粒子渲染为纹理化的点精灵（使用 GL_POINTS）。利用 OpenGL 的 gl_PointCoord 变量，自动生成纹理坐标并提供给片段着色器。

3. 透明度处理

随着粒子年龄的增加，逐渐减少粒子的 alpha 值（透明度），使得粒子在其生命周期内逐渐消失。

3.实验内容

3.1 场景粒子效果相关的初始化

首先，我们需要初始化一个场景粒子效果，包括以下步骤：

1. 编译和链接粒子系统所需的顶点着色器和片元着色器。
2. 设置背景清空颜色。
3. 启用点精灵和混合功能。
4. 加载粒子的贴图纹理文件。
5. 设置粒子的生命周期和重力方向。

```
void SceneParticles::initScene()
{
    compileAndLinkShader();

    glClearColor(0.1f,0.1f,0.1f,1.0f);

    glEnable(GL_POINT_SPRITE);
    glDisable(GL_DEPTH_TEST);

    // Enable blending
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // Set the point size
    glPointSize(10.0f);

    projection = mat4(1.0f);

    angle = (float)(PI / 2.0f);

    // Generate our vertex buffers
    initBuffers();

    vector<unsigned char> image;
    unsigned iWidth, iHeight;
    GLuint textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);
    lodepng::decode(image, iWidth, iHeight, "bluewater.png");
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, iWidth, iHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, &image[0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

    prog.setUniform("ParticleTex", 0);
    prog.setUniform("ParticleLifetime", 3.5f);
    prog.setUniform("Gravity", vec3(0.0f,-0.2f,0.0f));
}
```

3.2 创建两个缓存区

我们将创建两个缓存区，一个用于存储粒子的初始速度，另一个用于存储每个粒子的出生时间。

```
void SceneParticles::initBuffers()
{
    nParticles = 8000;

    // Generate the buffers
    glGenBuffers(1, &initVel);   // Initial velocity buffer
    glGenBuffers(1, &startTime); // Start time buffer

    // Allocate space for all buffers
    int size = nParticles * 3 * sizeof(float);
    glBindBuffer(GL_ARRAY_BUFFER, initVel);
    glBufferData(GL_ARRAY_BUFFER, size, NULL, GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, startTime);
    glBufferData(GL_ARRAY_BUFFER, nParticles * sizeof(float), NULL, GL_STATIC_DRAW);
```

3.3 创建初始速度缓冲区

为了存储传递给顶点着色器的初始速度，我们将生成一个缓冲区，并随机生成每个粒子的初始速度向量。每个粒子的初始速度将从垂直锥形范围内随机选取，速度的大小在 1.25 到 1.5 之间。

```
vec3 v(0.0f);
float velocity, theta, phi;
GLfloat *data = new GLfloat[nParticles * 3];
for (unsigned int i = 0; i < nParticles; i++ ) {

    theta = glm::mix(0.0f, (float)PI / 6.0f, randFloat());
    phi = glm::mix(0.0f, (float)TWOPI_F, randFloat());

    v.x = sinf(theta) * cosf(phi);
    v.y = cosf(theta);
    v.z = sinf(theta) * sinf(phi);

    velocity = glm::mix(1.25f,1.5f,randFloat());
    v = glm::normalize(v) * velocity;

    data[3*i]   = v.x;
    data[3*i+1] = v.y;
    data[3*i+2] = v.z;
}
glBindBuffer(GL_ARRAY_BUFFER,initVel);
glBufferSubData(GL_ARRAY_BUFFER, 0, size, data);
```

3.4 创建开始时间缓冲区

为了记录每个粒子的生成时间，我们将创建另一个缓冲区，并按固定速率为每个粒子分配一个生成时间。

```
delete [] data;
data = new GLfloat[nParticles];
float time = 0.0f;
float rate = 0.00075f;
for( unsigned int i = 0; i < nParticles; i++ ) {
    data[i] = time;
    time += rate;
}
glBindBuffer(GL_ARRAY_BUFFER,startTime);
glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), data);
```

3.5 绑定顶点属性

配置和启用用于粒子系统的顶点属性，包括初始速度和开始时间，以便后续在顶点着色器中使用这些数据进行粒子模拟和渲染

```
glBindBuffer(GL_ARRAY_BUFFER,startTime);
glBufferSubData(GL_ARRAY_BUFFER, 0, nParticles * sizeof(float), data);

glBindBuffer(GL_ARRAY_BUFFER,0);
delete [] data;

glGenVertexArrays(1, &particles);
glBindVertexArray(particles);
glBindBuffer(GL_ARRAY_BUFFER, initVel);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(0);

glBindBuffer(GL_ARRAY_BUFFER, startTime);
glVertexAttribPointer(1, 1, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(1);

glBindVertexArray(0);
```

3.6 设置顶点着色器

顶点着色器用于模拟和渲染粒子系统中每个粒子的运动和透明度变化。它根据粒子的初始速度、出生时间以及外部设置的重力和生命周期参数，计算粒子在当前时间点的位置，并根据其生命周期内的时间变化调整透明度。最终，使用 MVP 矩阵将计算出的位置变换到裁剪空间，用于渲染粒子的视觉效果。

```
#version 400

layout (location = 0) in vec3 VertexInitVel; // Particle initial velocity
layout (location = 1) in float StartTime;    // Particle "birth" time

out float Transp;  // Transparency of the particle

uniform float Time;  // Animation time
uniform vec3 Gravity = vec3(0.0,-0.05,0.0);  // world coords
uniform float ParticleLifetime;  // Max particle lifetime

uniform mat4 MVP;

void main()
{
    // Assume the initial position is (0,0,0).
    vec3 pos = vec3(0.0);
    Transp = 0.0;

    // Particle dosen't exist until the start time
    if( Time > StartTime ) {
        float t = Time - StartTime;

        if( t < ParticleLifetime ) {
            pos = VertexInitVel * t + Gravity * t * t;
            Transp = 1.0 - t / ParticleLifetime;
        }
    }

    // Draw at the current position
    gl_Position = MVP * vec4(pos, 1.0);
}
```

3.7 设置片元着色器

片元着色器负责根据输入的纹理和透明度参数，渲染每个粒子的最终颜色输出。

```
in float Transp;
uniform sampler2D ParticleTex;

layout ( location = 0 ) out vec4 FragColor;

void main()
{
    FragColor = texture(ParticleTex, gl_PointCoord);
    FragColor.a *= Transp;
}
```

3.8 创建场景

3.8.1 初始化 OpenGL 和场景

初始化 OpenGL 环境并设置基础场景

```
int main(int argc, char* argv[])
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL);

    glutInitWindowSize(800, 600);
    glutInitWindowPosition(300, 200);
    glutCreateWindow(argv[0]);

    glewInit();

    initializeGL();
    glutDisplayFunc(paintGL);
    glutReshapeFunc(resizeGL);

    glutMainLoop();
    return 0;
}
```

3.8.2 渲染和更新循环

paintGL() 函数是 GLUT 的显示回调，它负责渲染场景和更新粒子系统。它首先计算当前时间，然后调用场景的 update() 函数来更新粒子系统，接着调用 render() 函数来渲染场景。渲染完成后，它交换缓冲区以显示图像，并请求 GLUT 重新显示窗口以实现动画。

```
void paintGL() {
    static double clockFac = 1.0 / CLOCKS_PER_SEC;
    float time = clock() * clockFac;
    scene->update(time);
    scene->render();
    glutSwapBuffers();
    glutPostRedisplay();

    gltGrabScreenTGA("Particles.tga");
}
```

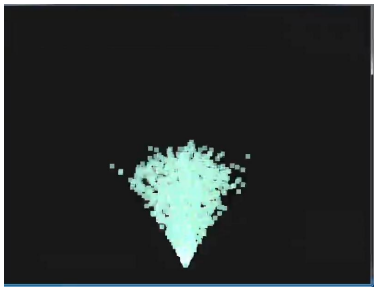
3.8.3 保存函数

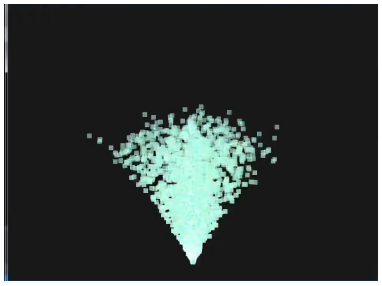
实现将当前帧保存为 TGA 图像的函数 gltGrabScreenTGA

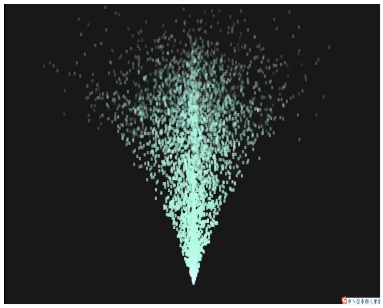
```
GLint gltGrabScreenTGA(const char* szFileName)
{
    FILE* pFile;                // File pointer
    TGAHEADER tgaHeader;        // TGA file header
    unsigned long lImageSize;   // Size in bytes of image
    GLbyte* pBits = NULL;       // Pointer to bits
    GLint iViewport[4];         // Viewport in pixels
    GLenum lastBuffer;          // Storage for the current read buffer setting

    // Get the viewport dimensions
    glGetIntegerv(GL_VIEWPORT, iViewport);

    // How big is the image going to be (targas are tightly packed)
    lImageSize = iViewport[2] * 3 * iViewport[3];

    // Allocate block. If this doesn't work, go home
    pBits = (GLbyte*)malloc(lImageSize);
    if (pBits == NULL)
        return 0;

    // Read bits from color buffer
    glPixelStorei(GL_PACK_ALIGNMENT, 1);
    glPixelStorei(GL_PACK_ROW_LENGTH, 0);
    glPixelStorei(GL_PACK_SKIP_ROWS, 0);
    glPixelStorei(GL_PACK_SKIP_PIXELS, 0);

    // Get the current read buffer setting and save it. Switch to
    // the front buffer and do the read operation. Finally, restore
    // the read buffer state
    glGetIntegerv(GL_READ_BUFFER, (GLint*)&lastBuffer);
    glReadBuffer(GL_FRONT);
    glReadPixels(0, 0, iViewport[2], iViewport[3], GL_BGR_EXT, GL_UNSIGNE
    glReadBuffer(lastBuffer);

    // Initialize the Targa header
    tgaHeader.identsize = 0;
    tgaHeader.colorMapType = 0;
    tgaHeader.imageType = 2;
    tgaHeader.colorMapStart = 0;
    tgaHeader.colorMapLength = 0;
    tgaHeader.colorMapBits = 0;
    tgaHeader.xstart = 0;
    tgaHeader.ystart = 0;
```

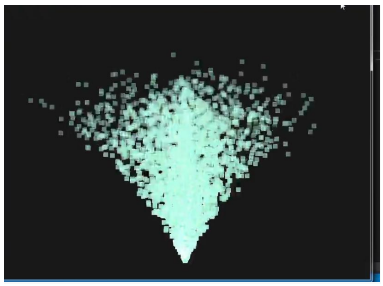

4. 实验结果截图

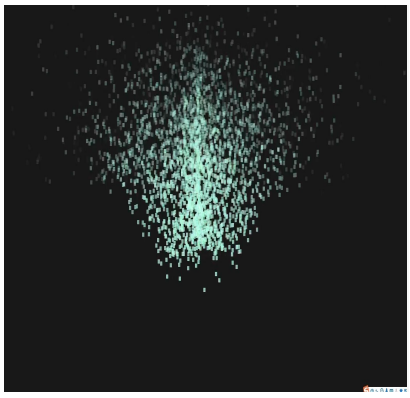
实验结果如下所示，我们生成了一个粒子喷泉，并且会随着生命周期而结束

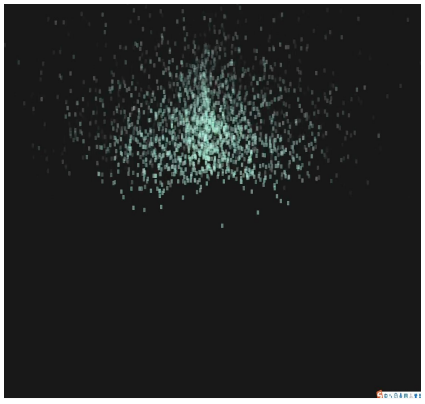


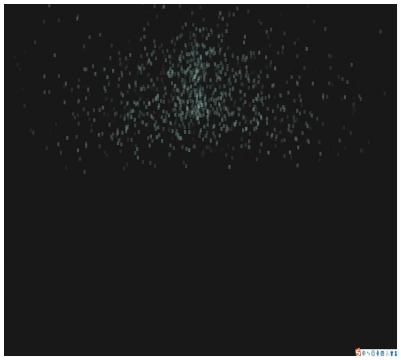












5. 实验环境

类目	内容
编程语言	C++
图形编程接口	OpenGL
开发环境	Visual Studio 2019
工具库	GLFW GLAD GLM GLUT

6. 实验总结

在完成本次实验中，我深入学习和实现了一个简单的粒子系统，模拟了水喷泉的效果。通过这个过程，我获得了以下几点主要经验和心得：

首先，粒子系统作为计算机图形学中常用的技术之一，能够高效地模拟和渲染复杂的自然现象，如烟雾、火焰和水流。相比传统的渲染方法，粒子系统以其独立运动和生命周期管理的特性，为这些模糊现象的再现提供了直观且高效的解决方案。

其次，本实验的目的在于通过实现水喷泉效果的粒子系统，掌握粒子系统的基本原理及其在计算机图形学中的应用。在实现过程中，我特别关注了粒子

的初始速度、加速度（如重力）、生命周期和透明度变化，这些因素共同作用使得喷泉效果看起来更加逼真。

在技术细节方面，我学习了如何在 OpenGL 环境中编译链接顶点和片元着色器，配置点精灵的渲染方式，并启用混合功能以处理粒子的透明度变化。通过粒子的初始速度和出生时间的缓冲区管理，我能够有效地模拟每个粒子在时间轴上的运动状态，并在渲染过程中实现动态的透明度效果，增强了视觉上的真实感和流畅度。

在解决技术挑战的过程中，我特别注重粒子生命周期的管理，确保粒子在生命周期结束后能够正确地消失，避免资源的浪费和渲染效率的降低。这要求我在编程中细致入微地处理每个粒子的状态变化，以实现预期的动画效果。

通过本次实验，我不仅加深了对粒子系统工作原理的理解，还掌握了在实际项目中应用这些技术的基本技能。这些知识和经验不仅对我当前的学习有帮助，也为我未来在计算机图形学和游戏开发领域的探索和发展奠定了坚实的基础。我期待能继续在这个领域深入学习，不断提升自己的技术能力和创造力。