北京交通大學

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）
21301112	燕宇菲	6	22	17	34	79

缺少参考文献；实验结果分析太简单

计算机图形学大作业实验报告

学　　院：软件学院

专　　业：软件工程

学生姓名：燕宇菲

学　　号：21301112

北京交通大学

2024 年 9 月

目录

1 摘要

本实验通过 OpenGL 实现了火焰粒子系统的绘制。

2 引言

2.1 实验背景与意义

在本学期的计算机图形学课程中，我们学习了多种模拟三维物体的方法，如网格细分、交并补法、粒子系统。网格细分以三角形为基本图元，将三维模型表面切分为更小的元素以增添模型细节，交并补法通过对多个几何体进行布尔运算创建复杂的几何结构，但它们都难以对复杂的三维物体进行逼真的描述，如水、火焰、烟雾、云。因为它们没有具体结构，没有具体、清晰且准确的表面；组成这些物体的物理结构往往是粒子，并且这些粒子的运动是随机的、表面是模糊的、不确定的。另外，它们也不是刚性物体，在描述下落、扩散、扰动（例如风吹散云）的时候，不能用常规的物理规律来描述他们的运动。考虑实际情况，它们还有生成的起点和消散的边界，这些起点、边界的位置通常也难以确定。粒子系统采用形状微小的粒子作为基本元素来表示不规则的模糊物体，每个粒子的位置随机，各自具有生命周期，能够很好地解决上述问题，为动态景物的描绘提供了更多可能。

2.2 国内外研究现状

1983 年，Reeves.V.T 在其发论文《Particle Systems A Technique for Modeling a Class of Fuzzy Objects》中首次提出了粒子系统的概念。从此，粒子系统就开始广泛运用于计算机中各种模糊景物的模拟。当前，粒子系统已被广泛运用于游戏开发、电影特效、科学可视化和虚拟现实等众多领域。在游戏开发领域，知名游戏引擎开发商 Epic Games 在《堡垒之夜》（Fortnite）等游戏中广泛使用了粒子系统来实现各种动态效果，而《魔兽时间》等大型游戏的制作公司 Blizzard Entertainment 也使用粒子系统来提升游戏画面的质感及真实度。在电影特效领域，ILM 在《星球大战》系列、《复仇者联盟》、《指环王》、《阿凡达》等大片中广泛使用粒子系统来实现复杂的特效和场景模拟。在科学可视化和虚拟现实领域，NASA 在科学研究和宇航员培训中使用粒子系统来模拟太空环境中的物理现象和动态效果。在军事虚拟现实系统中，技术人员借助粒子系统尽可能营造真实的战场环境，包括爆炸产生的烟雾、沙砾、火焰、碎片，模拟真实场景以提升军队的作战能力和心理素质。

3 实验环境

实验环境如表 4-1 所示。

分类	名称	规格
软件	Visual Studio	2019
硬件	操作系统	Windows11
	内存	8GB
	处理器	Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz

4 实验原理

4.1 粒子系统

粒子系统将不规则的模糊物体看作由众多粒子构成的粒子团。

4.2 粒子

（1）定义

粒子（Particle）是实数域上的一个 n 维向量。

（2）基本属性

在粒子系统中，粒子之间的相互作用被忽略，任一粒子均具有一些基本属性用以区分其他粒子，如：

1）初始位置（Position）：

粒子的物理坐标和初始运动方向。粒子在三维空间内具有确定的起源位置。粒子系统有一个生成范围区域，在这个区域内，生成的新粒子将会被随机排布。

2）速度(Velocity)：

包括速率(speed)和方向(direction)。位置的改变依赖于速度，速度是一个矢量，表明系统运动快慢和粒子的运动方向，在经过每个时间间隔后，粒子位置被改变。

3）加速度(Acceleration)：

和速度作用于位置一样，加速度作用于速度。粒子的加速度通常适用于外力作用。外力经常是重力，或者是粒子间的引力或斥力。

4）生命周期(Life)：

每个粒子均具有自己的生命周期，由帧数衡量。随着时间的推移，粒子的生命值不断减小，直到粒子死亡（生命值为 0）。一个生命周期结束时，另一个生命周期随即开始，有时为了使粒子能够源源不断地涌出，必须使一部分粒子在初始后立即死亡。

5）初始颜色

一个粒子系统被赋予一个平均颜色，并给出一个介于 0 到 1 之间的随机数乘该颜色的最大偏差。粒子透明度和粒子大小也由平均值和最大变化决定。

5.2 粒子系统实现画面流程

粒子系统实现画面的基本步骤如下：

（1）产生新的粒子：

粒子系统通过定义的发射源（如点、线、面、体）生成新的粒子，可具有不同的初始速度、方向和其他属性。

（2）赋予每个新粒子一定的属性：

每个新粒子生成时，粒子系统会为其分配特定的属性，如位置、速度、寿命、颜色、大小等。这些属性可以根据预设值、随机生成或根据特定规则计算而来。

（3）删去已经超过生存期的粒子：

系统会监测每个粒子的存在时间，一旦粒子达到其生命周期的终点或者满足某种条件（如碰撞到障碍物），系统会将其从粒子系统中移除，以释放资源。

（4）根据粒子的动态属性对粒子进行移动和变换：

在粒子存在期间，粒子系统会根据粒子的当前状态（如位置和速度）以及外部影响（如重力、风力）对粒子进行更新。这些更新会改变粒子的位置和运动状态，模拟出各种动态效果。

（5）显示由活跃粒子组成的图像：

粒子系统通过渲染技术将活跃的粒子渲染为点、线、纹理或其他几何体，最终进行可视化。

4.3 火焰物理模型

假设粒子产生区域为三维空间中 y=0 的 XOZ 平面，并在平面上随机产生粒子，由下向上发射，模拟火焰向上燃烧的效果。

火焰燃烧时，通常是焰心部分火焰非常旺盛，向外逐渐减少。用粒子来描述为：通常粒子初始化区域的中心附近粒子较为密集，向外逐渐变得稀疏，呈正态分布。通过下式可以对粒子的初始位置进行赋值，使之符合正态分布的特征。其中 Rand1()、Rand2()为[-1,1]内的随机数，Adj_value 为人

为设置的权重，保证粒子生成的随机性。

$$Pos(x) = \sum_{i=0}^{n} Rand1() * Adj\_value$$

$$Pos(y) = 0$$

$$Pos(z) = \sum_{i=0}^{n} Rand2() * Adj\_value$$

为保证火焰的局部随机性，同时维持火焰形状，需要将粒子的初始速度限定在合适范围内。

$$InitVeloc = (MaxVeloc - MinVeloc) * rand\_Num + MinVeloc$$

InitVeloc 为初始速度，MaxVeloc 为最大速度，MinVeloc 为最小速度，rand_Num 为[0,1]的随机数。

5 实验步骤

运用粒子系统的相关知识，本次实验实现了火堆型火焰的绘制。

首先新建 Flame.h,定义火焰的相关属性和方法。其中的两个宏表示粒子的两种类型，设置了粒子的一些初始属性，火焰半径和中心。

```
namespace Flame {
    #define PARTICLE_TYPE_LAUNCHER 0.0f
    #define PARTICLE_TYPE_SHELL 1.0f
        //最大速度
    #define MAX_VELOC glm::vec3(0.0,5.0,0.0)
        //最小速度
    #define MIN_VELOC glm::vec3(0.0,3.0,0.0)
        //最大最小速度差距
    #define DEL_VELOC glm::vec3(0.0,2.0,0.0)
        //最长生命周期
    #define MAX_LIFE 2.0f*1000
        //最短生命周期
    #define MIN_LIFE 1.0f*1000
        //初始点大小
    #define INIT_SIZE 30.0f;

    const int MAX_PARTICLES = 18000;//定义粒子发射系统最大的粒子数
    //初始发射器例子数量
    const int INIT_PARTICLES = 10000;
    //火焰中心
    const glm::vec3 center(0.0f);
    const float r = 0.3f;
```

之后定义粒子。type 表示粒子类型，一类为发射器，只发射粒子而不运动，当年龄到达设置时

间后重置发射器的年龄。第二类粒子为普通粒子，负责运动。alpha 表示粒子的透明度，透明度越大表示越不透明。在火焰初期，火焰粒子透明度很低，然后逐渐升高，最后又逐渐降低，造成火焰中心火焰粒子最亮，上升到一定的高度逐渐消失的现象。

```
struct FlameParticle
{
    float type;
    glm::vec3 position;
    glm::vec3 velocity;
    float lifetimeMills;//年龄
    float alpha;//alpha通道
    float size;//粒子大小
    float life;//寿命
};
```

在 Flame 类中定义粒子初始化、初始化火焰、粒子渲染等函数。

```
class Flame
{
public:
    Flame();
    ~Flame();
    void Render(float frametimeMills, glm::mat4& worldMatrix, glm::mat4 viewMatrix, glm::mat4& projectMatrix);
private:
    bool InitFlame(glm::vec3& pos);
    void UpdateParticles(float frametimeMills);//更新粒子的位置等
    void InitRandomTexture(unsigned int size);//生成1维随机纹理
    void RenderParticles(glm::mat4& worldMatrix, glm::mat4& viewMatrix, glm::mat4& projectMatrix);
    void GenInitLocation(FlameParticle partciles[], int nums);//生成初始粒子

    unsigned int mCurVBOIndex, mCurTransformFeedbackIndex;
    GLuint mParticleBuffers[2]; //粒子发射系统的两个顶点缓存区
    GLuint mParticleArrays[2];
    GLuint mTransformFeedbacks[2];//粒子发射系统对应的TransformFeedback
    GLuint mRandomTexture;//随机一维纹理
    CTexture mSparkTexture;//Alpha纹理
    CTexture mStartTexture;
    float mTimer;//粒子发射器已经发射的时间
    bool mFirst;
    Shader* mUpdateShader;//更新粒子的GPUProgram
    Shader* mRenderShader;//渲染粒子的GPUProgram
};
```

下面是上述函数的实现。

InitFlame 函数主要用于初始化 Flame 类中的粒子系统，包括设置初始粒子属性、生成和绑定 GPU 端所需的 Transform Feedback 对象和缓存对象，以及将随机纹理绑定到更新着色器中。这些准备工作为后续粒子系统的更新和渲染提供了必要的数据和资源。

```
bool Flame::InitFlame(glm::vec3 & pos)
{
    FlameParticle particles[MAX_PARTICLES];
    memset(particles, 0, sizeof(particles));
    particles[0].type = PARTICLE_TYPE_LAUNCHER;//设置第一个粒子的类型为发射器
    particles[0].position = pos;
    particles[0].lifetimeMills = 0.0f;
    particles[0].velocity = glm::vec3(0.0f, 0.1f, 0.0f);
    GenInitLocation(particles, INIT_PARTICLES);
    glGenTransformFeedbacks(2, mTransformFeedbacks);
    glGenBuffers(2, mParticleBuffers);
    glGenVertexArrays(2, mParticleArrays);
    for (int i = 0; i < 2; i++)
    {
        glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, mTransformFeedbacks[i]);
        glBindBuffer(GL_ARRAY_BUFFER, mParticleBuffers[i]);
        glBindVertexArray(mParticleArrays[i]);
        glBufferData(GL_ARRAY_BUFFER, sizeof(particles), particles, GL_DYNAMIC_DRAW);
        glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, mParticleBuffers[i]);
    }
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, 0);
    glBindVertexArray(0);
    //绑定纹理
    mUpdateShader->use();
    glBindTexture(GL_TEXTURE_1D, mRandomTexture);
    mUpdateShader->setInt("gRandomTexture", 0);
    return true;
}
```

Render 函数通过累加计时器、更新粒子状态、渲染粒子以及更新索引，实现了每帧粒子系统的更新和绘制操作。

```
void Flame::Render(float frametimeMills, glm::mat4& worldMatrix,
    glm::mat4 viewMatrix, glm::mat4& projectMatrix)
{
    mTimer += frametimeMills*1000.0f;
    UpdateParticles(frametimeMills*1000.0f);
    RenderParticles(worldMatrix, viewMatrix, projectMatrix);
    mCurVBOIndex = mCurTransformFeedbackIndex;
    mCurTransformFeedbackIndex = (mCurTransformFeedbackIndex + 1) & 0x1;
}
```

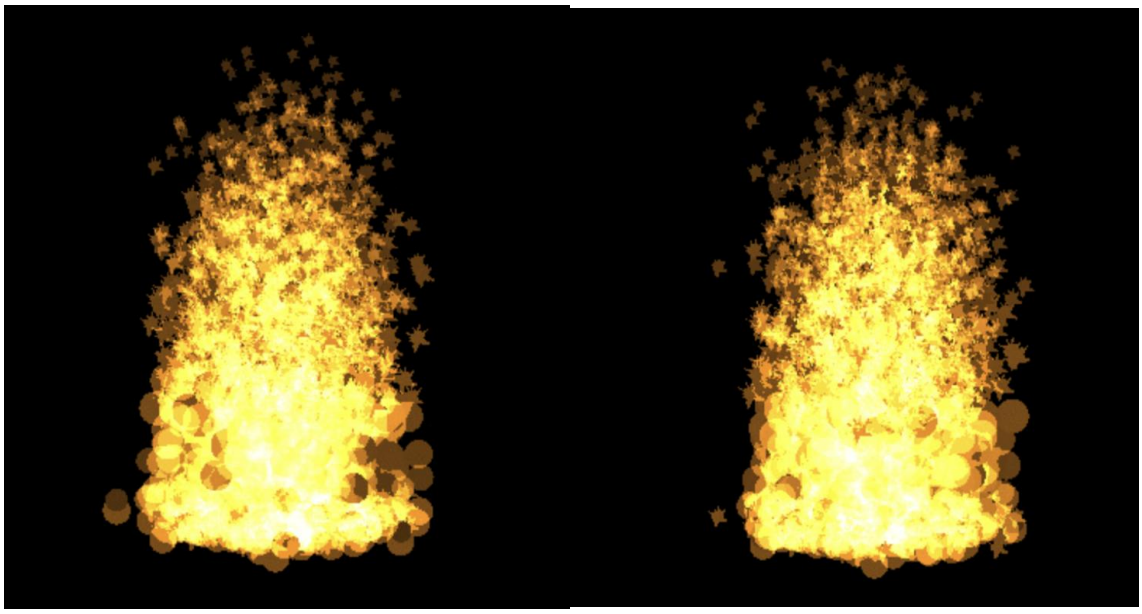
UpdateParticles 方法用于更新粒子的状态。它使用更新着色器设置粒子的时间间隔、计时器值以及生命周期范围等参数。然后绑定一个随机纹理，以增加粒子运动的随机性。通过启用光栅化丢弃功能，直接将渲染结果写入变换反馈缓存，而不实际渲染到屏幕上。之后，设置粒子数组和缓存对象的顶点属性指针，根据是否首次更新选择不同的方式绘制粒子。最后，完成变换反馈的结束并关闭相关的顶点属性，解绑定粒子数组和缓存对象，完成粒子状态的更新操作。

```
void Flame::UpdateParticles(float frametimeMills)
{
    mUpdateShader->use();
    mUpdateShader->setFloat("gDeltaTimeMillis", frametimeMills);
    mUpdateShader->setFloat("gTime", mTimer);
    mUpdateShader->setFloat("MAX_LIFE", MAX_LIFE);
    mUpdateShader->setFloat("MIN_LIFE", MIN_LIFE);
    mUpdateShader->setVec3("MAX_VELOC", MAX_VELOC);
    mUpdateShader->setVec3("MIN_VELOC", MIN_VELOC);

    //绑定纹理
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_1D, mRandomTexture);
    //mUpdateShader->setInt("gRandomTexture",0);

    glEnable(GL_RASTERIZER_DISCARD);//我们渲染到TransformFeedback缓存中去，并不需要光栅化
    glBindVertexArray(mParticleArrays[mCurVBOIndex]);
    glBindBuffer(GL_ARRAY_BUFFER, mParticleBuffers[mCurVBOIndex]);
    glBindTransformFeedback(GL_TRANSFORM_FEEDBACK, mTransformFeedbacks[mCurTransformFeedbackIndex]);
```

```
glEnableVertexAttribArray(0);//type
glEnableVertexAttribArray(1);//position
glEnableVertexAttribArray(2);//velocity
glEnableVertexAttribArray(3);//lifetime
glEnableVertexAttribArray(4);//alpha
glEnableVertexAttribArray(5);//size
glEnableVertexAttribArray(6);//life
glVertexAttribPointer(0, 1, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, type));
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, position));
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, velocity));
glVertexAttribPointer(3, 1, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, lifetimeMills));
glVertexAttribPointer(4, 1, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, alpha));
glVertexAttribPointer(5, 1, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, size));
glVertexAttribPointer(6, 1, GL_FLOAT, GL_FALSE, sizeof(FlameParticle), (void*)offsetof(FlameParticle, life));
glBeginTransformFeedback(GL_POINTS);
if (mFirst)
{
    glDrawArrays(GL_POINTS, 0, INIT_PARTICLES);
    mFirst = false;
}
else {
    glDrawTransformFeedback(GL_POINTS, mTransformFeedbacks[mCurVBOIndex]);
}
```

```
glEndTransformFeedback();
    #define glEndTransformFeedback GLEW_G
    扩展到: __glewEndTransformFeedback
    联机搜索
glDisableVertexAttribArray(4);
glDisableVertexAttribArray(5);
glDisableVertexAttribArray(6);
glDisable(GL_RASTERIZER_DISCARD);
glBindVertexArray(0);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

6 实验结果



7 心得体会

通过本次实验，我结合课上所学，查阅相关资料，认真学习了计算机图形学中粒子系统的相关理论知识，并进行实际应用，对 OpenGL 的 Transformer Back 特性有了初步的认识和体会。这个将理论转化为实际的过程很难，但是最后实现后令人受益匪浅。