北京交通大学课程设计（论文）

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）
21301062	陈艺彬	9	23	17	34	83

计算机图形学论文

基于图元分解与扩散算法的爆炸效果模拟

Explosion effect simulation based on primitive decomposition and diffusion algorithm

学　　院：软件学院

专　　业：软件工程

学生姓名：陈艺彬

学　　号：21301062

指导教师：吴雨婷

北京交通大学

2024 年 6 月

中文摘要

摘要:本次实验提出了一种基于图元分解和扩散模拟的爆炸效果实现方法，并使用PyOpenGL 进行了详细的实现和实验验证。

通过 Voronoi 算法进行图元分解，生成符合自然破裂形态的多边形碎片，同时结合高度判断保留部分三角面片的原有形态，增强了爆炸效果的细节表现。利用力的计算对图元进行位移、旋转和缩放模拟，实现了物体在爆炸过程中的破碎和扩散。

实验结果显示，所提出的方法在合理的硬件配置下能够实现高效的实时渲染，并具有较高的动态真实感。

关键词：图元分解；扩展模拟；实时渲染；Voronoi 算法；PyOpenGL

目　　录

中文摘要 II
目　　录 III
1 引言 1
2 相关工作介绍 2
2.1 图元管理算法 2
2.1.1 Octree 算法 2
2.2 图元分解算法 3
2.2.1 Voronoi 算法 3
2.2.2 Delaunay 三角剖分 3
2.2.3 结合 3
2.3 性能优化策略 4
2.3.1 多线程并发处理 4
2.3.2 数据压缩和实时数据处理 5
2.3.3 优化渲染技术 6
2.4 总结 6
3 实现原理 7
3.1 算法总体设计 7
3.2 算法具体实现 7
3.2.1 图元分解技术 7
3.2.2 高度判定 8
3.2.3 扩散模拟 8
3.3 总结 8
4 实验设置 9
4.1 实验配置 9
4.1.1 硬件配置 9
4.1.2 软件配置 9
4.2 实验场景 9
5 实验结果与分析 10
6 结论 11
参考文献 12

1 引言

在遥远的杂烩大陆之上，豌豆射手正惬意而贪婪地汲取阳光，可恍惚间一股腐烂的气味使它回过了神，“嘭”下意识的一发射击击碎了远方的僵尸，僵尸的尸体发生了爆炸，原来杂烩宇宙的裂缝正悄然扩大，进一步吸引着其他世界的外来者降临，这片大陆终将不再平凡...

在杂烩大陆之中，豌豆射手的攻击得到了增强，可以使僵尸发生爆炸。而如何在动态的视觉效果之中，有效地生成和管理图元是图形学的一个重要研究领域，本次实验将聚焦于立体图形爆炸效果的实现，探讨如何通过增加图元数量及进行位移变换来产生逼真的爆炸效果。

论文主要包括了相关工作介绍、基本原理、实验方案设计、实验结果与分析、结论五大模块。

2 相关工作介绍

图形爆炸效果是图形学领域的一个研究热点，本小节将介绍与图形爆炸效果相关的算法及本次实验中的环境配置。

公式：

$$\phi = \frac{D_p^2}{150}\frac{\psi^3}{(1-\psi)^2} \tag{2-1}$$

$$C_2 = \frac{3.5}{D_p}\frac{(1-\psi)}{\psi^3} \tag{2-2}$$

式中 D_p—— 多孔质材料的平均粒子直径(m)；

ψ—— 孔隙度（孔隙体积占总体积的百分比）；

ϕ—— 特征渗透性或固有渗透性，与材料的结构性质有关(m^2)。

将 $\frac{1}{\sqrt{2}}$ 写成 $1/\sqrt{2}$ 或 $2^{-1/2}$

2.1 图元管理算法

在三维图形处理中，图元作为基本的构建单元，其高效管理对于实现复杂动画至关重要。

2.1.1 Octree 算法

八叉树结构是由 Hunter 博士于 1978 年首次提出的一种数据模型[1]，是一种树形数据结构，每个内部节点都正好有八个子节点。八叉树常用于分割三维空间。将其递归细分为八个卦限，从而有效地组织和索引图元，可以显著提升空间查询地效率，目前广范适用于三维图形应用中。

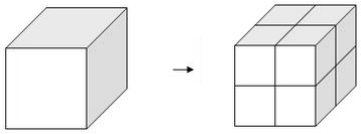

图 2-1-1 递归子切分一个立方体为多个卦限

2.2 图元分解算法

图元分解算法在图形学中被广泛运用于模拟物体的破坏过程，如爆炸效果。为了实现更真实的模拟效果，常结合使用 Voronoi 算法和 Delaunay 三角剖分。以下是对两种算法的详细介绍。

2.2.1 Voronoi 算法

Voronoi 算法是由乌克兰数学家建立的空间分割算法[2]，是一种基于空间划分的算法，通过生成 Voronoi 图来实现物体的碎片化，每个 Voronoi 区域由一个种子点和所有离该点最近的点组成，通过生成不规则的多边形碎片，来获得逼真的视觉效果。

图 2-2-1 二十个点和沃洛诺伊单元格

2.2.2 Delaunay 三角剖分

Delaunay 三角剖分时一种将点集连接成三角网的算法[2]，通过最大化最小角的方式生成高质量的三角形分解，其算法的关键是对离散数据点合理地连城三角网，即构建 Delaunay 三角形，具体建立三角形步骤如下列方法所示。

方法 Delaunay Triangulation[3]:

a. 离散点自动构建三角网
b. 寻找并记录与每个离散点相邻的所有三角形编号
c. 对与每个离散点相邻的三角形按顺时针或逆时针方向排序，以便下一步连接生成泰森多边形
d. 计算并记录每个三角形的外接圆圆形
e. 连接每个离散点的相邻三角形的外接圆圆心，得到泰森三角形

2.2.3 结合

Voronoi 分解提供了不规则的碎片形状，增强视觉效果的真实感，Delaunay 三角剖分在每个 Voronoi 区域内生成高质量的三角网络，用于进一步实现爆炸模拟。

2.3 性能优化策略

项目使用 PyOpenGL 作为图形编程工具，访问和调用 OpenGL 提供的功能，从而实现图形渲染、图像处理和创建交互式图形应用程序等任务。

在进行图形爆炸效果模拟实验中，采用了以下几种性能优化策略，进而达到提高系统的计算效率和渲染性能的目的。

2.3.1 多线程并发处理

PyOpenGl 本身是单线程的，为此采用多线程管理和利用 GPU 的并行计算能力来实现性能优化。

1) 多线程资源加载

使用 Python 的`threading`将资源加载，放入后台线程中进行，从而避免在主渲染线程中加载资源时的卡顿。

```
import threading
from OpenGL.GL import *
from OpenGL.GLUT import *

def load_texture_async(file_path):
    texture_id = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, texture_id)

    glBindTexture(GL_TEXTURE_2D, 0)
    return texture_id
threading.Thread(target=load_texture_async, args=("png",)).start()
```

2) 异步缓冲区更新

使用 OpenGL 的缓冲取映射技术，在后台线程中更新缓冲取数据。

```
import numpy as np

def update_vertex_buffer_async(vbo, data):
    glBindBuffer(GL_ARRAY_BUFFER, vbo)
    ptr = glMapBuffer(GL_ARRAY_BUFFER, GL_WRITE_ONLY)
    np_array = np.frombuffer(ptr, dtype=np.float32, count=len(data))
    np_array[:] = data
    glUnmapBuffer(GL_ARRAY_BUFFER)
    glBindBuffer(GL_ARRAY_BUFFER, 0)

vbo = glGenBuffers(1)
data = np.array([...], dtype=np.float32)
threading.Thread(target=update_vertex_buffer_async, args=(vbo,
data)).start()
```

2.3.2 数据压缩和实时数据处理

通过数据压缩和数据处理，减少存储和传输的开销，提高处理速度

1) 压缩纹理格式

```
from OpenGL.GL.EXT.texture_compression_s3tc import *

def load_compressed_texture(file_path):
    texture_id = glGenTextures(1)
    glBindTexture(GL_TEXTURE_2D, texture_id)
    # ...
    glCompressedTexImage2D(GL_TEXTURE_2D, 0,
GL_COMPRESSED_RGB_S3TC_DXT1_EXT, width, height, 0, imageSize, data)
    glBindTexture(GL_TEXTURE_2D, 0)
    return texture_id
```

使用 PyOpenGL 支持的而压缩纹理格式来来减少纹理数据的存储和传输开销。

2) 实时数据处理

利用 OpenGL 的缓冲取对象 VBO 进行高效的数据传输和处理，通过映射缓冲区技术直接在 CPU 内存中操作 GPU 数据。

```
import numpy as np

def update_buffer_object(vbo, data):
    glBindBuffer(GL_ARRAY_BUFFER, vbo)
    glBufferData(GL_ARRAY_BUFFER, data.nbytes, data,
GL_DYNAMIC_DRAW)

vbo = glGenBuffers(1)
data = np.array([...], dtype=np.float32)
update_buffer_object(vbo, data)
```

2.3.3 优化渲染技术

在优化渲染技术，在保证视觉质量的同时提高性能。

1) LOD 技术

在渲染时根据物体碎片与视点的距离动态调整渲染细节，通过减少远处物体的顶点

```
def update_lod(model, camera_position):
    distance = np.linalg.norm(model.position - camera_position)
    if distance > FAR_DISTANCE:
        model.set_lod(LOW)
    elif distance > MID_DISTANCE:
        model.set_lod(MEDIUM)
    else:
        model.set_lod(HIGH)
```

数和纹理分辨率来提高渲染性能。

2.4 总结

本小节详细介绍了一种图元管理算法和一种经典图元分解算法，并对本次实验中采取的性能优化策略进行了详细说明。

3 实现原理

本小节将详细介绍基于图元分解和扩散的爆炸效果实现原理。原理主要为将图形分解产生更多的三角面片图元，根据高度决定保留哪些三角面片的原有形态[4]，通过位移、旋转和缩放处理剩余三角面片来实现逼真的爆炸效果。

3.1 算法总体设计

本实验的算法设计总体流程图如下所示。

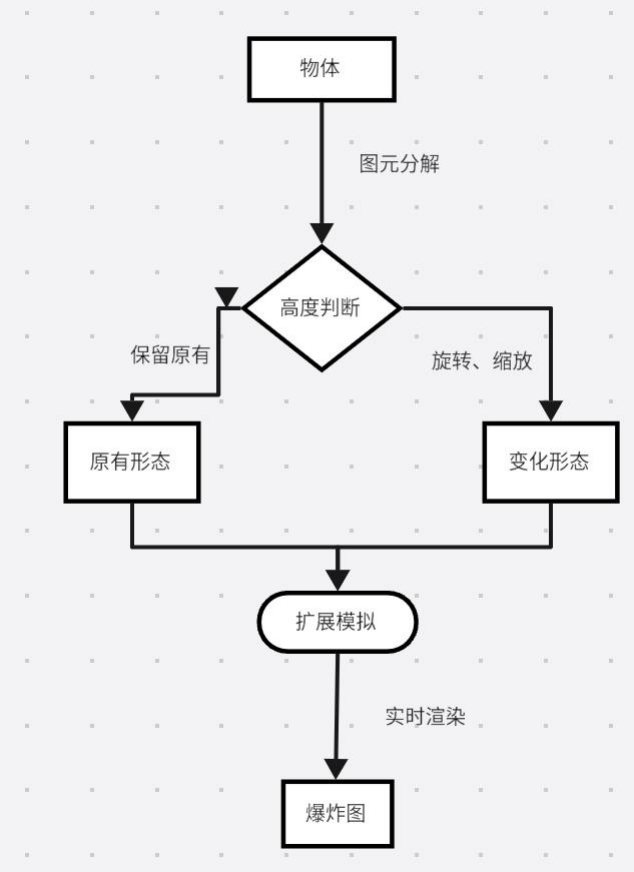

图 3.1.1 算法设计总体流程图

3.2 算法具体实现

3.2.1 图元分解技术

采用 Voronoi 算法进行图元分解，以生成不规则但符合自然破碎形态的图元。

3.2.2 高度判定

根据三角面片的高度决定保留哪些面片的原有形态，高度判断基于三角面片的质心高度。

3.2.3 扩散模拟

根据爆炸中心的位置信息，对剩余三角面片进行位移、旋转和缩放操作。

其中力的计算与图元的位移公式如下:

$$F = \frac{C - P}{|C - P|^2} \tag{3-1}$$

$$P_{new} = P + F \cdot \Delta t \tag{3-2}$$

$$P_{\text{new}} = R \cdot P \tag{3-3}$$

$$P_{new} = F \cdot \text{s} \tag{3-4}$$

式中 C —— 爆炸中心

P —— 图元质心

R —— 旋转矩阵

s —— 缩放因子

3.3 总结

本小节详细描述了基于图元分解和扩散的爆炸效果实现方法，介绍了整体算法流程以及对每个部分的算法进行了解释说明。

4 实验设置

本小节将详细介绍实验设置。

4.1 实验配置

该部分将介绍下具体的硬件和软件配置。

4.1.1 硬件配置

1) 处理器: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.8GHz (8 CPUs),~2.8GHz
2) 内存:16GB RAM
3) 图形处理器:NVIDIA GeForce GTX 1080
4) 操作系统:Windows 11

4.1.2 软件配置

1) 开发环境:Python 3.11
2) 图形库:PyOpenGL 3.1.7

4.2 实验场景

实验场景为模拟孤岛危机 3 纳米服模型[5]在受到爆炸力作用时候分解为多个三角面片，通过调整爆炸中心的位置、力的大小和方向，来实现爆炸效果。

5 实验结果与分析

从实验结果图中，可以发现通过 Voronoi 分解和扩散模拟，生成的爆炸效果具有较好的视觉观感,并且结合自身体验实时渲染对笔记本的负担也不高。

图 5-1 原始模型效果

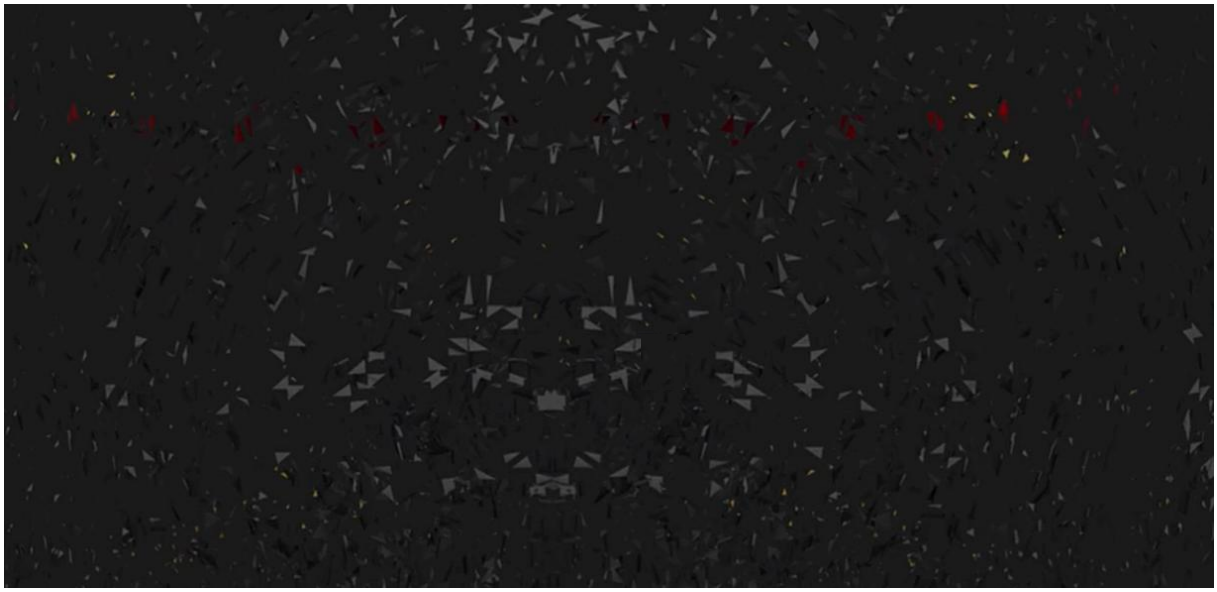

图 5-2 模型爆炸效果

6 结论

在本次实验中，采取了一种基于图元分解和扩散模拟的爆炸效果实现方法，结合 Voronoi 算法和高度判断，增强了爆炸效果的视觉真实感和细节表现，并使用 PyOpenGL 进行了详细的实验和实验验证，通过实验效果的呈现，可以证明该方案的有效性。

参考文献

[1] https://www.cnblogs.com/Glucklichste/p/11505743.html

[2] https://zh.wikipedia.org/zh-cn/%E6%B2%83%E7%BD%97%E8%AF%BA%E4%BC%8A%E5%9B%BE

[3] https://dsa.cs.tsinghua.edu.cn/~deng/cg/project/2014s/2014s-d.pd

[4] https://lab.uwa4d.com/lab/5bc54e8404617c5805d4e89b

[5] https://polytonic.github.io/Glitter/