Homework 3 Report

刁义嘉 518030910146

提交文件说明

```
518030910146_刁义嘉_hw3
|  report.pdf  #本报告
|
├─CUDA   #CUDA版本vonoroi图生成（包含绘制Delauny三角剖分）
|       1080test.png         #课程服务器 算法运行时间比较
|       2080test.png         #其他服务器 算法运行时间比较
|       jumpflood-1080.bmp   #课程服务器 jump flood 算法运行结果
|       jumpflood-2080.bmp   #其他服务器 jump flood 算法运行结果
|       makefile             #编译运行CUDA版本程序
|       naive-1080.bmp       #课程服务器 暴力算法运行结果（绘制Delauny三角剖分）
|       naive-2080.bmp       #其他服务器 暴力算法运行结果（绘制Delauny三角剖分）
|       vonoroi.cu           #CUDA源代码
|
└─OpenGL #OpenGL版本vonoroi图生成
        main.cpp                    #OpenGL主程序
        OpenGL_naive_output.png     #OpenGL 暴力算法运行结果
        OpenGL_naive_time.png       #OpenGL 暴力算法运行事件
        shader_s.h                  #Shader 类定义文件
        voronoiBasicShader.fs       #GLSL fragment shader(暴力算法实现)
        voronoiBasicShader.vs       #GLSL vertex shader
```

对于CUDA版本程序，命令行输入`make run`即可编译运行。

OpenGL环境依赖：GLAD, GLFW3（与课堂所授配置方法相同）。

OpenGL

代码实现

OpenGL部分主要实现了vonoroi图生成的暴力算法。代码实现有如下关键点：

- 显示整个窗口屏幕：

  在LearnOpenGL教程的三角形绘制基础上进行更改。使用索引缓冲对象，在屏幕上绘制两个三角形，铺满$x, y \in [-1.0, 1.0]$的区域。相关的关键代码：

```
float vertices[] = {
    -1.0f, -1.0f, 0.0f,
    1.0f, -1.0f, 0.0f,
    -1.0f, 1.0f, 0.0f,
    1.0f, 1.0f, 0.0f,
};
unsigned int indices[] = {
    0, 1, 2,  // first Triangle
    1,2,3   // second Triangle
};
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
GL_STATIC_DRAW);
```

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices,
GL_STATIC_DRAW);
// in render loop:
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
```

- 逐像素计算颜色：

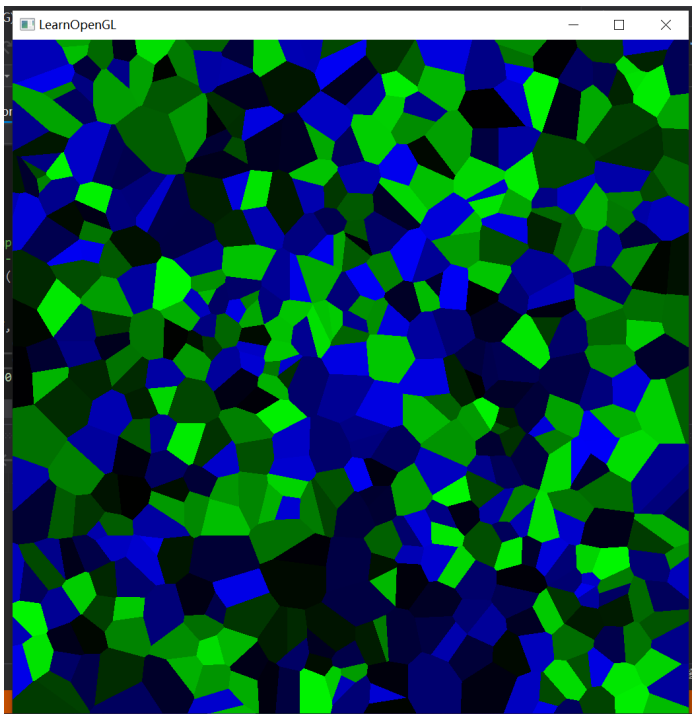
  利用fragment shader对每个像素渲染的特性，以及暴力算法中每个像素的颜色计算没有数据依赖的良好并行性，采用直接在fragment shader中实现暴力算法的方法进行像素颜色的计算。相关的fragment shader关键代码：

```
void main()
{
    int minindex = 0;
    for(int i = 0;i < 500;++i){
        if(dist(i)<dist(minindex)){
            minindex = i;
        }
    }
    if(minindex < 250)
        gl_FragColor =
vec4(0,float(minindex+1)/256.0,0,float(minindex)/500.0);
    else
        gl_FragColor = vec4(0,0,float(minindex-
249)/256.0,float(minindex)/500.0);
}
```

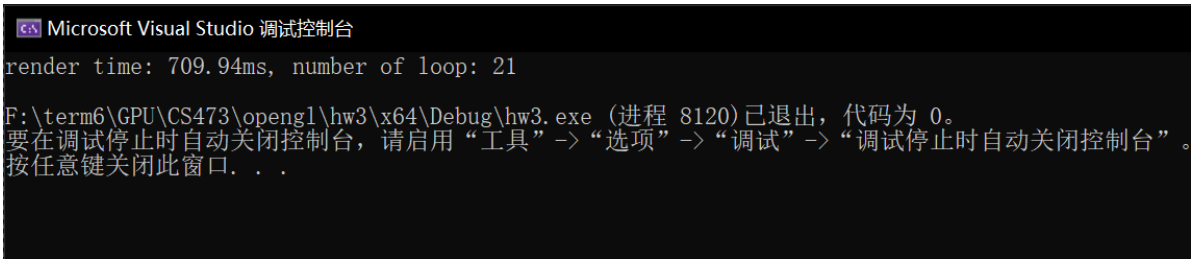
- 500个输入点采用随机生成方式生成。为方便fragment shader的每个线程读取，直接写入`voronoiBasicShader.fs`文件中。

实验结果

vonoroi图：



渲染时间测试（主要测试渲染循环的执行时间）：



CUDA

代码实现

CUDA部分主要实现了暴力算法和Jump Flood算法。

说明未采用OpenGL实现Jump Flood算法的原因。首先，fragment shader中不适合实现jump flood算法：jump flood算法中像素点之间有数据依赖关系，但fragment shader无法对线程进行管理；如果在其中仿照暴力算法实现，相当于对每一个像素点运行一次jump flood算法，会导致大量的冗余计算。其次，CUDA的线程管理较为方便，能够实现真正意义上并行执行jump flood算法。

代码实现有如下关键点

1. 暴力算法

   线程的管理方式类似于作业二。block size等于1024，共有1024个block。

   ```
   draw<<<Size, Size>>>(cudaimg, pointX, pointY, numPoints, Size);
   ```

   每一个线程负责当前像素的颜色计算，暴力计算距当前像素最近的点即可。

2. Jump Flood 算法

   - Kernel：每个kernel遍历周边八个像素的当前最近点，更新八个像素点的最近输入点的index。需要对kernel输入步长。

     ```
     KerneljumpFlood<<<GridDim, BlockDim>>>(Size, Size, SiteArray, Ping, Pong, i, Mutex);
     ```

     需要特别注意的是，由于本kernel更新的并非本线程完全管理的数据，线程间数据写可能会发生冲突，所以在遍历并写周边八个像素点时，**需要进行原子操作**。这里采用mutex锁来进行原子操作。

     ```
     // lock
     while (atomicCAS(Mutex, -1, nextpixelIdx) == nextpixelIdx){}
     // unlock
     atomicExch(Mutex, -1);
     ```

   - Kernel 调用总次数：jump步长每减半一次，重新返回host调用一次kernel。

3. 图像生成

   - 主要框架沿用作业2的`.bmp`图像生成代码
   - 着色：

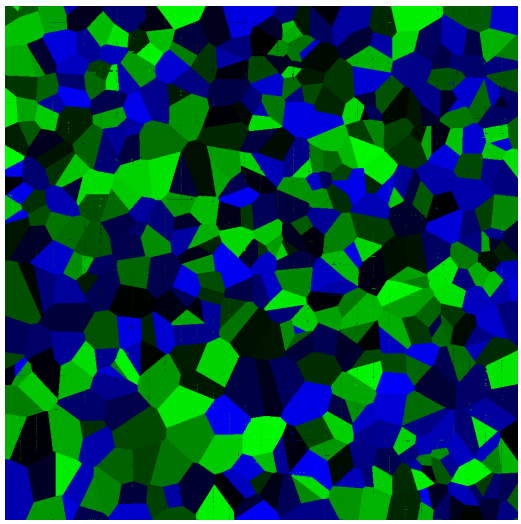
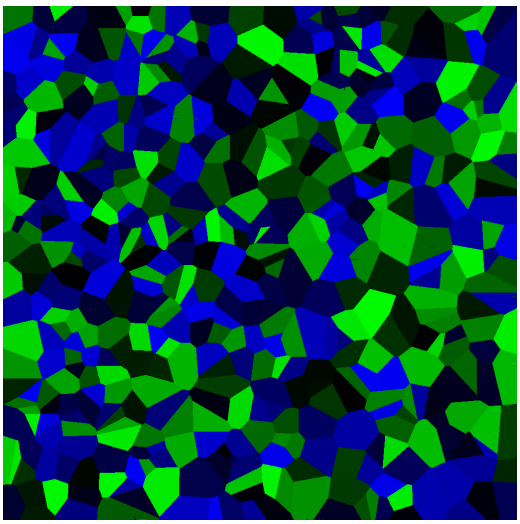
     由于本题目有500个不同点，所以采用`r=0,g=1,2,...251,b=0`以及`r=0,g=1,b=1,2,...251`500种颜色进行着色。原思路用四色定理着四种颜色，但由于实现较复杂进行了这个简化。
   - 绘制Delauny Triangulation

实现了比较简单版本的按像素绘制直线的函数。缺点是，当斜率较大时，直线会出现断线的情况。

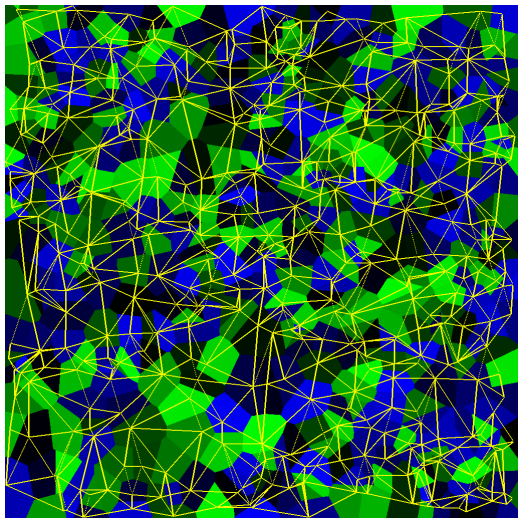
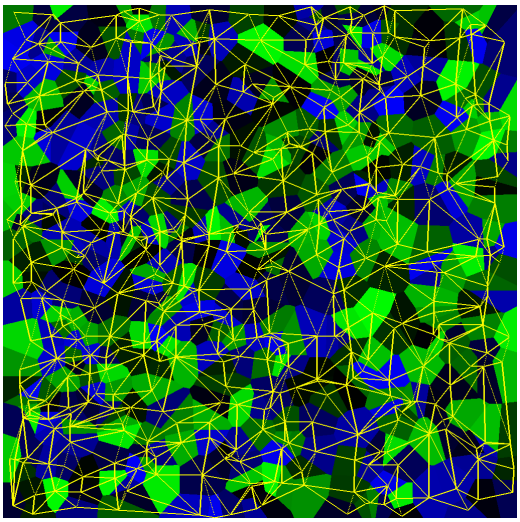
主要实现在`drawline()` `drawTriangle()`两个函数中。

实验结果

Jump Flood算法生成的vonoroi图（左图为2080ti机器，右图为课程服务器）：

暴力算法生成的vonoroi图，与Delauny Triangulation（左图为2080ti机器，右图为课程服务器）：

计算时间测试：

```
yjdiao@sjtuuser:~/CS473/cuda/hw3$ make run
nvcc vonoroi.cu -o vonoroi.out
./vonoroi.out
JumpFlood cal cost:58.5064070ms
BMP file loaded successfully!
Naive cal cost:1.7683680ms
BMP file loaded successfully!
```

```
dlgpu@ArchLab102:~/yijia/hw3$ ./vonoroi.out
JumpFlood cal cost:49.5915650ms
BMP file loaded successfully!
Naive cal cost:5.1479350ms
BMP file loaded successfully!
```

需要说明的是，jump flood生成的图像有一定的噪声。因此并未在此基础上绘制delauny triangulation。

实验结果分析

由时间测试可知，jump flood算法整体运行时间高于暴力算法。可能的原因是：

暴力算法中，线程之间没有依赖关系，能够全程使用GPU完成并发计算。但jump flood算法中，一是kernel计算中混入原子操作，二是步长变换需要在GPU-CPU之间切换多次，切换与访存开销巨大。