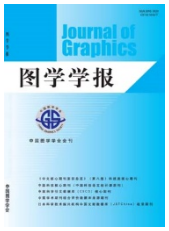


图学学报
Journal of Graphics
ISSN 2095-302X,CN 10-1034/T

《图学学报》网络首发论文

题目：基于计算着色器的并行 Delaunay 三角剖分算法
作者：陈国军，李震烁，陈昊祯
收稿日期：2024-06-29
网络首发日期：2024-10-21
引用格式：陈国军，李震烁，陈昊祯．基于计算着色器的并行 Delaunay 三角剖分算法[J/OL]．图学学报. https://link.cnki.net/urlid/10.1034.t.20241018.0916.002

网络首发时间：2024-10-21 13:35:54
网络首发地址：https://link.cnki.net/urlid/10.1034.t.20241018.0916.002

图 学 学 报
JOURNAL OF GRAPHICS

基于计算着色器的并行 Delaunay 三角剖分算法

陈国军，李震烁，陈昊祯
(中国石油大学(华东)青岛软件学院、计算机科学与技术学院，山东 青岛 266580)

摘　　要：Delaunay 三角剖分是一种经典的计算几何算法，在众多领域中有着广泛的使用，随着实际需求的不断提高，现有的 Delaunay 三角剖分算法已不能满足大规模数据的需求，为此，提出了一种基于计算着色器的并行 Delaunay 三角剖分方法，该方法通过纹理缓存将点集数据输入到计算着色器中，并利用计算着色器加速 Delaunay 三角剖分，同时在现有方法的基础上提出动态插入法解决点集在离散空间中的重映射问题。此外，为了能够让显存有限的 GPU 构建出远超其显存限制的 Delaunay 三角网，提出基于计算着色器的分区双向扫描算法，该方法将点集划分为多个子区域，然后通过扫描各个子区域的方式进行构网。实验结果表明：在相同运行环境下，基于计算着色器的方法与现有的方法相比缩短了构网时间。同时分区双向扫描算法很好的解决了 GPU 的显存瓶颈问题，能让显存有限的 GPU 构建出远超其显存容量的 Delaunay 三角网。

关　键　词：Delaunay 三角剖分；计算着色器；GPU；并行计算；Voronoi 图

Delaunay triangulation partitioning processing algorithm based on compute shaders

CHEN Guojun, LI Zhenshuo, CHEN Haozhen
(Qingdao Institute of Software, College of Computer Science and Technology, China Universtity of Petroleum (East China), Qingdao Shandong 266580, China)

Abstract: Delaunay triangulation is a classic computational geometry algorithm, which is widely used in many fields. With the increasing demand for practical applications, the existing Delaunay triangulation algorithm can no longer meet the requirements of large-scale data. Therefore, a parallel Delaunay triangulation method based on compute shaders is proposed. This method inputs point set data into the compute shader through texture buffer, and uses the compute shader to work the Delaunay triangulation algorithm. At the same time, a dynamic insertion method is proposed based on the existing method to solve the remapping problem of point sets in discrete space. In addition, in order to enable GPUs with limited video memory to construct Delaunay triangulations that far exceed their video memory limitations, a partitioned bidirectional scanning algorithm based on compute shaders is proposed. This method divides the point set into multiple sub-regions, and then constructs the network by scanning each sub-region. Experimental results show that under the same operating environment, the method based on compute shaders shortens the network construction time compared with the existing method. At the same time, the partitioned bidirectional scanning algorithm solves the GPU memory bottleneck problem well, allowing GPUs with limited video memory to construct Delaunay triangulations that far exceed their video memory capacity.

Keywords: Delaunay triangulation; compute shader; GPU; parallel computing; Voronoi diagram

收稿日期：2024-06-29；定稿日期：2024-09-25
Received：29 June, 2024；**Finalized**：25 September, 2024
第一作者：陈国军(1968–)，男，副教授，博士，硕士生导师。主要研究方向为图形图像处理、虚拟现实。E-mail：chengj@upc.edu.cn
First author: CHEN Guojun(1967−), PhD, associate professor, master's supervisor, His main research interests are graphic image processing and virtual reality. E-mail：chengj@upc.edu.cn

Delaunay 三角剖分作为一种重要的计算工具，在众多领域中都有着广泛的使用，例如地理系统中的三维模型重建[1]，地形模拟[2]，人机接口技术[3]，以及游戏领域的寻路算法，图像渲染[4]等，并且 Delaunay 三角剖分在物理[5]、生物[6]、计算机视觉[7]等领域也有着广泛的使用。

早期的 Delaunay 三角剖分的实现算法大多数为基于 CPU(中央处理器)的串行生成算法，近几年来随着对于构网速度的需求不断提升，传统的串行生成算法已不能够满足需求，因此基于 GPU(图像处理单元)的并行算法逐渐进入大众视野。GPU 的内部有着大量的流处理器单元，这些流处理单元带给了 GPU 强大的并行计算能力，因此利用 GPU 进行 Delaunay 三角剖分能够显著加快构网速度。现如今主流的 GPU 通用计算平台是 Nvidia 公司的 CUDA(Compute Unified Device Architecture)平台，CUDA 平台能够有效的调度 GPU 进行并行计算，从而提高构网速度。

然而现有的基于 GPU 的构网算法仍存在一些问题。使用 CUDA 进行构网的过程中，需要频繁地与渲染管线进行数据交换，从而影响整体运行效率，并且只有 Nvidia 的显卡才能使用 CUDA 平台进行并行计算，这意味着 CUDA 程序对于硬件有一定的需求，从而影响程序兼容性。此外，由于 GPU 的显存容量有限，当数据量过大以至于超过 GPU 的显存容量时，GPU 则无法进行构网，而目前缺少一种让显存有限的 GPU 构建远超其显存大小的构网算法。

为了解决上述问题，本文提出了一种通过计算着色器(Compute Shader)调度 GPU 进行构网的并行 Delaunay 三角剖分算法，该算法能够对大规模的数据集进行分区并行计算，从而解决 GPU 所存在的性能瓶颈问题。同时计算着色器的使用也能够有效的加快构网速度，同时还能提高程序的兼容性。通过后续的实验表明，本文提出的算法能够有效对大规模点集进行构建 Delaunay 三角网，且基于计算着色器的三角剖分的算法相较于使用其他计算平台的算法来说有着明显的性能提升。

1 相关工作

1.1 串行 Delaunay 三角剖分算法

早期的 Delaunay 三角剖分为单线程的串行算法，其中使用最为广泛的是增量插入法[8]，增量插入法的优点是原理简单，实现起来较为容易，并且对于各种数据集增量插入算法都能够得出点集的 Delaunay 三角网，因此增量插入算法在各个实际场景中的使用非常广泛。基于增量插入算法也有很多的改进算法，LIN 等[9]将增量插入算法与分治法的思想结合起来，在云环境中实现了对大规模点集的分布式处理；ZHOU 等[10]通过控制待插入点进行三角剖分的顺序，提出了 HCPO(hierarchical column-prime order)算法，该算法将插入点分成了不同层次。在进行插入时减少了检测 Delaunay 三角网合法性的次数。有效的降低了增量插入法的时间复杂度；增量插入法的缺点是时间复杂度约为 $O(n^2)$，较高的时间复杂度使得其在处理大规模点集时构网速度过慢，构网效率不高。

扫描线算法也是一种经典的串行算法，由 FORTUNE 等[11]最早提出，本质是利用 Voronoi 图与 Delaunay 三角网的对偶关系，先利用扫描线法构成点集的 Voronoi 图，随后利用对偶关系生成点集的 Delaunay 三角剖分，该算法的时间复杂度也为 $O(nlogn)$，优势在于拥有快速的构网速度。在 Fortune 提出该扫描线算法之后又有其他的扫描线算法被陆续提出，比如 ZALIK 提出的基于增量插入算法的扫描线算法[12]，以及 BINIAZ 等人提出的使用扫描圆来获取点集数据并进行三角剖分的算法[13]。

1.2 并行 Delaunay 三角剖分算法

随着多核 CPU 的普及，利用多核 CPU 对点集并行构网能够显著加快构网速度，这其中最有代表的算法是分治法，分治法最早由 D. T. LEE 等人在 1980 年提出[14]，核心思想为将大点集分割为小的子点集再进行递归处理，优点在于时间复杂度相对较低，大约为 $O(nlogn)$，此外，子点集之间的合并可以并行处理，进而进一步加快处理时间。这种分而治之的思想对于后续并行构网算法的研究有着很大的启发，SONG 等[15]使用分治法对激光雷达获取的点云数据进行三角剖分，使用 MapReduce 的 Map 阶段将点云数据进行划分，随后对划分后的数据集进行三角剖分，最后使用三角网生成算法将各个区域的 Delaunay 三角网合并。NGUYEN 等[16]将分治法和增量插入算法进行结合，提出了一种新的并行算法，首先将点集进行分区，随后利用增量插入算法在插入

时只会影响周围有限数量的三角形的特性，对不同分区的点同时使用增量插入算法，保证各个分区在构网的同时不影响其他区域的三角网，从而实现对大规模点集的并行处理。分治法的主要优化思路在于对合并过程的优化，其中包括对分区之间三角网的生成以及减少合并过程所带来的额外的性能开销。

GPU 拥有大量的计算单元，将这些计算单元用于通用计算上能够极大的提升计算效率，因此当需要对大量的点进行实时处理时使用 GPU 进行三角网的构建也是一个不错的选择。A. NAVARRO 等[17]基于边缘翻转法，设计出对已有的三角网格转化为 Delaunay 三角网的并行算法。HOFF III 等[18]利用泰森多边形(也称 Voronoi 图)与 Delaunay 三角剖分的对偶关系实现了 Delaunay 三角剖分的并行生成。RONG 等[19]在此基础了提出了 JFA(Jump Flooding Algorithm)算法来缩短离散泰森多边形的生成时间。CAO 等[20]提出的 PBA 算法则进一步加快了离散泰森多边形的生成速度，变相提升 Delaunay 三角剖分的构建效率。

1.3 计算着色器

计算着色器(Compute Shader)是现代图形 API 中用于通用计算任务的一种着色器。它能够调用 GPU 众多的计算单元，将这些计算单元用于并行计算上，从而能够计算与图像渲染无关的信息，通常用于对场景粒子的物理变化计算[21]，地形分析[22]，或者是对大数据集进行并行处理等[23]。计算着色器的优势在于它能够直接对 GPU 显存中的数据进行读写和输入输出，因此计算着色器具有非常高的运行效率，同时计算着色器运行于传统的渲染管线之外，因此计算着色器可以运行在渲染管线的任意阶段，拥有高度的使用自由性。

计算着色器通过工作组来将并行计算的任务进行分配，计算着色器内部定义有两种工作组，全局工作组与本地工作组，其中全局工作组控制本地工作组的数量，本地工作组控制用于计算的线程数量，工作组的大小可以通过 x、y、z 三个变量控制。当需要进行并行计算时，计算着色器将计算任务分配到不同的工作组中，随后工作组对计算任务分配线程，从而实现数据的并行计算。计算着色器通过纹理缓冲进行输入输出，因此一般将需要处理的数据输入到纹理缓冲中，然后输入到计算着色器中进行处理。

2 离散空间中构建 Delaunay 三角剖分

由于 Delaunay 三角剖分与 Voronoi 图存在对偶关系，而 Voronoi 图可以使用并行算法快速生成，因此本文通过构建点集的 Voronoi 图的方式来快速构建 Delaunay 三角剖分。各个步骤的效果如图 1 所示。

(a) (b) (c)

图 1 构网各步骤示意图((a)点集数据；(b)点集的 Voronoi 图；(c)点集的 Delaunay 三角剖分)

Fig. 1 Each step of Triangulation((a) Datasets; (b) Voronoi diagram of datasets; (c) Delaunay triangulation of datasets)

2.1 输入点数据转化为离散空间数据

由于构建 Voronoi 图的相关计算主要在离散空间中进行，因此需要为点集数据创建合适大小的离散空间，并将点集数据存储到离散空间对应的位置中，设 R 为合适大小的离散空间，点集数据中的点的坐标为(x,y)，点集数据在离散空间中对应的坐标为(tx,ty)，则(tx,ty)由公式(1)可得

$$\begin{aligned} tx &= n*\frac{x-x_{\min}}{x_{\max}-x_{\min}} \\ ty &= n*\frac{y-y_{\min}}{y_{\max}-y_{\min}} \end{aligned}, (tx,ty \in N) \qquad (1)$$

使用式(1)在离散空间中找到对应位置后即

可将点集数据存储到 R 中对应的位置上，而 R 中未被映射的点用默认值(-1,-1)代替，为了在下文中表述方便，将离散空间中存有点集数据的点称为种子点，存有默认值的点称为背景点。构建离散 Voronoi 图的本质就是为每个背景点寻找距离最近的种子点，并将距离最近的种子点信息存储到背景点的位置上。

2.2 构建离散 Voronoi 图

构建离散 Voronoi 图的方法采用分区条带法(Parallel Banding Algorithm, PBA)[20]，这是一种高度并行的构建离散 Voronoi 图的方法，该方法分两步构建离散 Voronoi 图，第一步构建点集的一维 Voronoi 图，第二步根据得到的一维 Voronoi 图构建二维 Voronoi 图。

首先并行访问每一个背景点，寻找与该背景点位于相同行的距离最近的种子点，并将满足条件的种子点坐标输入到该背景点中。设当前访问的背景点为 $r(x,y)$，与 r 具有相同 y 值的种子点集合为 $S_x=\{s_1,s_2,...,s_n\}$，则满足条件的 $s_a(x_a,y_a)$ 具有以下性质

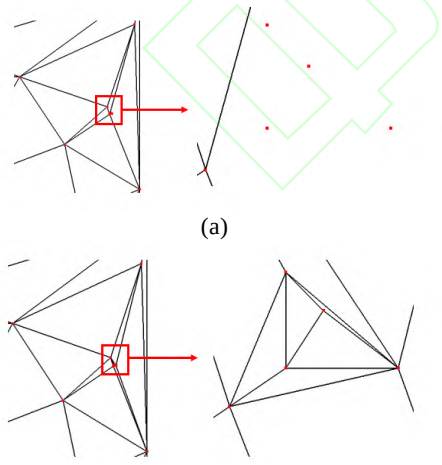
$$\|r-s_a\|\le\|r-s_n\|(s_a\in S_x,s_n\in S_x,s_a\ne s_n) \quad (2)$$

找到满足条件的 s_a 后将 s_a 的坐标信息存储到 r 对应的位置里，当离散空间中所有的背景点都被访问过后即可认为完成构建了一维 Voronoi 图。

二维 Voronoi 图则是在一维 Voronoi 图的基础上进行升维获得。首先为每一列分配一个线程，在线程中获取当前列出现过的种子点信息的坐标集合 $Q_y=\{q_1,q_2,...,q_n\}$，随后从 Q_y 中为当前列的目标点寻找满足条件的种子点信息，设当前访问的目标点为 $r(x,y)$，Q_y 中满足条件的种子点为 q_a，则 q_a 具有如下性质

$$\|r-q_a\|\le\|r-q_n\|(q_a\in Q_y,q_n\in Q_y,q_a\ne q_n) \quad (3)$$

找到 q_a 后将 q_a 的坐标信息存储到 r 中，当离散空间中所有背景点都找到了距离其最近的种子点信息时则完成了离散 Voronoi 图的构建。



(a)

(b)

图 2　重映射导致的构网缺失((a)错误的三角网；(b)正确的三角网)

Fig. 2　Missing triangulation caused by remapping ((a) Error triangulation; (b) Correct triangulation))

2.3 Delaunay 三角剖分

得到离散 Voronoi 图之后即可通过寻找 Voronoi 图的 Voronoi 顶点来构建点集的 Delaunay 三角剖分。具体做法是并行访问 R 中的所有离散点，获取 $R(x,y)$，$R(x+1,y)$，$R(x,y+1)$，$R(x+1,y+1)$ 所存储的种子点信息，当获取到 3 个或四个不同的种子点信息时，则说明 $R(x,y)$ 是一个 Voronoi 顶点，然后根据获取到的种子点信息数量构建三角形。若获取到三个不同的种子点信息则构建一个三角形，若获取到四个种子点信息则对这四个点使用最大空圆准则进行判断，选择合适的点构建三角形。

直接由 Voronoi 图得到的 Delaunay 三角网不一定是完整的，原因在于部分 Voronoi 顶点位于离散空间之外。从而导致部分三角形无法被构建，为了生成完整的 Delaunay 三角剖分需要对三角网的边界进行补边。补边的方法参考 RONG 等人提出的方式[24]，这里不再赘述，当补边完成时则成功构建了点集的 Delaunay 三角剖分。

3　动态插入法解决重映射问题

现有的构网算法仍存在一个问题，当点集存在多个距离相近的点时，若使用公式(1)进行映射，会出现多个点同时映射到同一离散点的情况，此时若继续执行三角剖分算法会因为部分点没有映射到离散空间中从而导致构网缺失，如图 2 所示，这些未映射到离散空间的点被称为缺失点。现有的解决构网缺失的方案是记录下缺失点的信息，随后当点集的 Delaunay 三角剖分构建完成后，使用增量插入法将缺失点插入到构建好的 Delaunay 三角剖分中[25]，然而由于增量插入法是基于 CPU 的串行方法，且额外调用 CPU 进行运算在数据的传输上也会产生额外的性能开销，因此该方法存在整体效率不高的问题。

缺失点出现的本质原因是由于离散空间的离散点的分布是固定的，而点集中的点有可能存在超出离散点范围之外的数据，从而导致部分数

据点无法找到对应位置的离散点，进而产生了缺失点。因此解决这一问题的关键在于在离散空间中为缺失点创建对应的离散点。

基于这一思想，本文提出了动态插入法(Dynamic Insertion)，具体做法如图 3 所示，其中蓝色点代表离散空间的背景点，红色点是成功映射到离散空间的点，绿色点是为成功映射到离散空间的缺失点，当检测到离散空间中存在缺失点时，在离散空间中根据缺失点的坐标为缺失点创建额外的离散行与离散列，随后将缺失点映射到额外创建的离散点中，这样一来就成功将缺失点映射到了离散空间中。从而能够让缺失点参与到后续的构网步骤中。

动态插入法与现有方法相比，优点在于该方法与原先的 PBA 算法进行了很好的融合，不需要使用额外的步骤进行构网，从而提高了构网速度。并且，动态插入法在离散空间中新创建的离散行能够让离散空间能够容纳的数据点数量增加，从本质上来说提高了离散空间的精度，从而使点集的 Voronoi 图的精度提高，并不会改变原先 Voronoi 图的结构，并且由于缺失点形成的三角形是由精度增加后的 Voronoi 图通过对偶化得到的，因此新增加的三角形能够满足 Delaunay 三角剖分的标准。

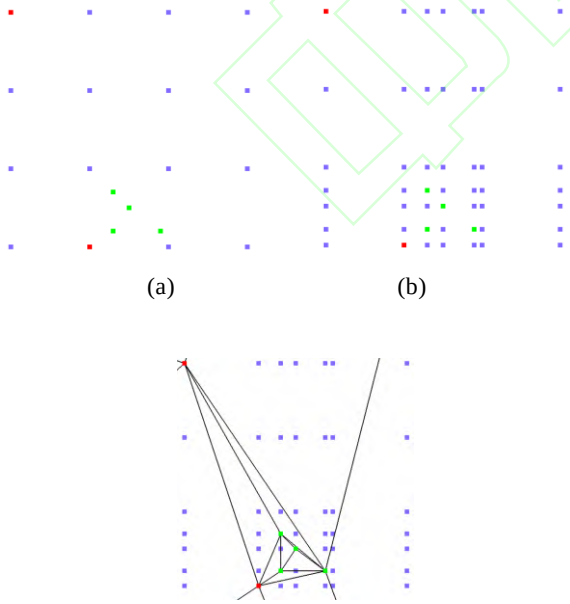


图 3　动态插入法各步骤示意图((a)确定缺失点的位置；(b)为缺失的数据点添加额外的离散行与离散列; (c)构建三角剖分))

Fig. 3　Each steps of Dynamic Insertion((a) Determine the location of missing points; (b)Add additional discrete rows and columns for missing data points; (c) Construct triangulation)

4　分区双向扫描算法

在实际应用中，当点集对应的离散空间规模超过 GPU 的显存容量时，会导致 GPU 无法对点集进行构网，造成离散空间规模过大的原因有两点，一是点集数据本身规模过大，二是动态插入法插入离散行与离散列时变相增加了离散空间的大小。为了减小 GPU 的负担，让 GPU 顺利的进行构网，需要对点集数据进行分区处理，然而目前的分区算法都是基于 CPU 的算法，尚未有基于 GPU 的分区处理算法。

为了解决上述问题，让 GPU 能够顺利的进行构网，本文提出了分区双向扫描算法(Partition Bidirectional Scanning, PBS)，该算法将分区思想与计算着色器的并行方法结合起来，核心思想是将点集数据划分到多个子区域中，随后对子区域进行横向与竖向的两次扫描，在扫描过程中将各个子区域边界处的原始信息传递到其他子区域，随后对各个子区域进行构网。其中，该算法的关键在于对各个分区之间的处理，以防止出现构网缺失，如图 4 所示。

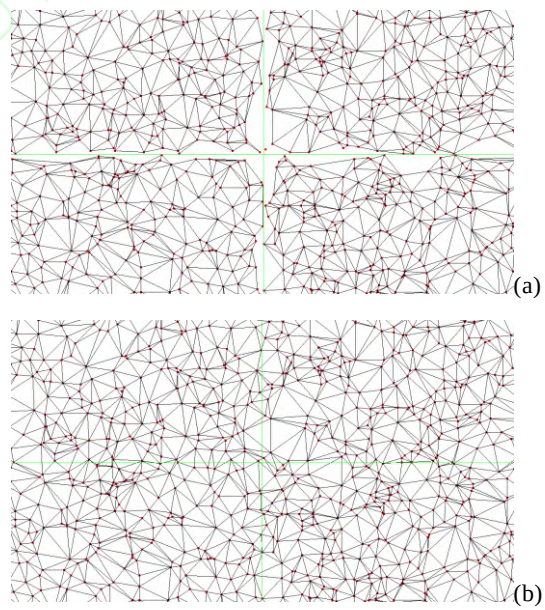


图 4　未对边界处理所造成的三角网缺失((a)分区间存在间隙的三角剖分；(b)正确的 Delaunay 三角剖分)

Fig 4.　Missing triangulation caused by no boundary treatment ((a) Triangulation with gaps between partitions; (b) Correct Delaunay Triangulation)

4.1　划分点集数据

数据进行处理前需要将原先的离散空间划分为 $m \times m$ 的区域，划分后的每个小块称为子区

域。划分离散空间的时候需要控制划分后的子区域的规模，单个子区域的规模不能超过 GPU 的处理上限。

4.2 横向扫描子区域并构建一维 Voronoi 图

划分子区域后需要遍历所有的子区域，构建各个子区域的一维 Voronoi 图，然而，若直接构建子区域的一维 Voronoi 图会导致各个子区域的一维 Voronoi 图无法正确的拼合在一起，原因在于子区域在构建 Voronoi 图之前未考虑相邻子区域对当前子区域的影响，从而导致各个子区域的一维 Voronoi 图相互独立。

因此，若想让子区域的一维 Voronoi 图能够正确的拼合在一起，需要在构建子区域一维 Voronoi 图之前将各个子区域边界处的种子点信息传递到相邻子区域的边界处，如图 5 所示，设点 a 为子区域 P 上最靠近右边界的种子点，点 b 为子区域 Q 中最靠近左边界的种子点，其中 a 与 b 位于相同行。由图中的步骤可知，当子区域 P 和 Q 接收到来自邻近子区域的种子点信息 b 和 a 后，P 和 Q 在构建 Voronoi 图时会考虑 b 和 a 两个种子点对当前子区域的影响，从而能够让两个区域的 Voronoi 图能够正确的拼合在一起。

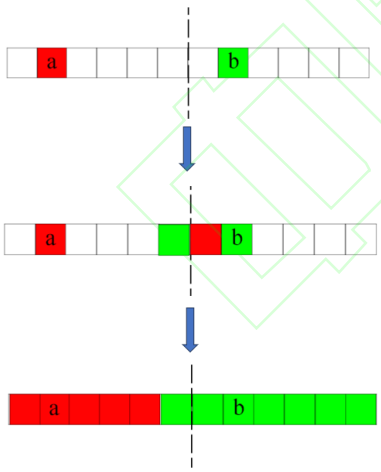


图 5　子区域在接收到来自邻近区域边界处的种子点信息后能正确地连接在一起

Fig. 5　Sub regions can be correctly connected together after receiving seed point information from neighboring region boundaries

传递种子点信息的方法则采用 “从左向右”以及“从右向左”的两次扫描步骤来进行传递，以下是”从左向右”的扫描步骤中，对其中一行子区域所执行的算法：

输入：位于同一行的子区域集合 $P=\{P_1,P_2,...,P_n\}$，集合中的子区域按照从左向右的顺序排列

输出：已得到来自左侧种子点信息的子区域集合 $P=\{P_1,P_2,...,P_n\}$

Step1：设置变量 $P_a=P_1$，$P_b=P_2$

Step2：并行访问 P_a 中的每一行，记录最靠近 P_a 右边界的种子点信息

Step3：将记录的种子点信息传输到 P_b 相同行的左边界处

Step4：P_b 接收到种子点后，判断原先左边界处是否存在种子点信息，若不存在则将接收到的种子点信息更新到左边界的背景点上

Step5：令 $P_a=P_2$，$P_b=P_3$，循环 Step2~Step4，直到 P_a 的右侧没有子区域为止。

”从左向右”的步骤结束后，再从右边子区域执行”从右向左”的扫描步骤，此时的 $P_a=P_n$，$P_b=P_{n-1}$，这里需要注意的是从右往左进行扫描时，从 P_a 到 P_b 的种子点信息不能是上一次扫描中由 P_b 传递到 P_a 的种子点信息。

当”从右向左”的扫描完成后，所有子区域都存有了来自左侧与右侧子区域边界部分的种子点信息。此时可使用 2.2 所述方法为每个子区域构建一维 Voronoi 图，由于已经获取了相邻子区域的边界处种子点信息，子区域的一维 Voronoi 图相互之间必然是紧密相连的，各步骤示意图如图 6 所示。

4.3 纵向扫描子区域并构建点集的 Voronoi 图

在子区域的一维 Voronoi 图构建完成后需要对各个子区域进行升维得到子区域的二维 Voronoi 图，同样的，在构建各个子区域的二维 Voronoi 图之前也需要对子区域进行竖向的两次扫描步骤，与 4.2 所述方法不同的地方在于竖向两次扫描的目的不一样，第一次扫描的目的是保证每个子区域能与下方的子区域正确相连，第二次扫描的目的是保证每个子区域能与上方子区域正确相连，同时在第二次扫描的过程中还会构建各个子区域的离散 Voronoi 图。

第一次扫描开始之前需要为每个子区域的每一列创建存有当前列种子点信息的种子点队列，种子点队列的作用是为后续的子区域间种子点信息传输以及构建 Voronoi 图的步骤提供载体，种子点队列创建完成后对子区域进行“从下向上”的扫描，设 P_a 为扫描到的子区域，Q_a 为 P_a 的种子点队列集合，$Q_a=\{q_{a1},q_{a2},...,q_{am}\}$，$P_b$ 是

位于 P_a 正下方的子区域，Q_b 为 P_b 的种子点队列集合，$Q_b=\{q_{b1},q_{b2},...,q_{bm}\}$，对 P_a 的所有种子点队列并行执行如下算法。

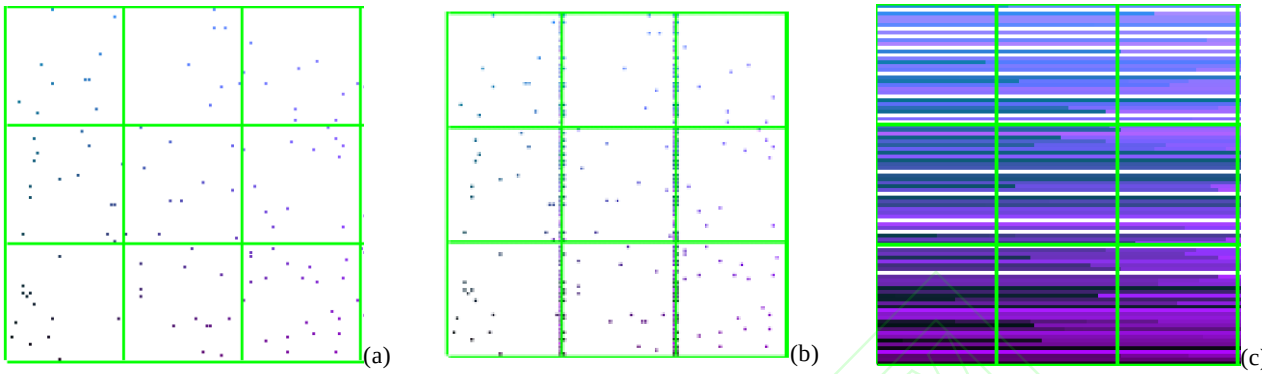


图 6　构建子区域一维 Voronoi 图各步骤示意图((a)初始状态；(b)完成两次扫描后的状态；(c)点集的一维 Voronoi 图)

Fig. 6　Schematic diagram of the steps to construct the one-dimensional Voronoi diagram of each sub-region((a) Datasets; (b) After completing two scans; (c) 1D-Voronoi of datasets)

输入：Q_a 的种子点队列 q_a，Q_b 的种子点队列 q_b，其中 q_a 与 q_b 位于相同列

输出：已获得下方相邻区域种子点信息的种子点队列 q_a

Step 1. 判断 q_a 中的种子点队列是否为空，若是则将 q_b 中的所有种子点信息复制到 q_a，随后结束算法；若不是则进行 Step2

Step 2. 将 q_b 的首个种子点信息转移到 q_a 的末尾处，随后判断 q_a 倒数第二个种子点的 Voronoi 区域与当前列的相交情况，若不与当前列相交则从 q_a 中删除该种子点信息

Step 3. 判断 q_a 末尾两个种子点信息是否来自 q_b，若是则结束算法，若不是则返回 Step2

其中 Step2 所提到的不与当前列相交的情况如图 7 所示，*a*, *b*, *c* 为 3 个种子点，*m* 为选定的一列，可以看出，*m* 列只穿过了种子点 *a* 和种子点 *b* 的 Voronoi 区域，未穿过种子点 *c* 的 Voronoi 区域，而创建 *m* 的种子点队列时由于一维 Voronoi 图的缘故会导致 *m* 的种子点队列会存有种子点 *c* 的信息，因此需要在 Step2 中识别出不与当前处理列所相交的种子点信息并删除该种子点信息。该步骤的目的是去除不相关的点，减少构网所需时间。

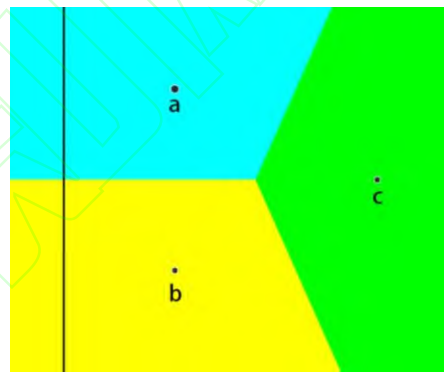


图 7　需要删除的原始点信息

Fig. 7　The original point that needs to be deleted

对于 Step3 所提到的判断条件的详细说明如图 8 所示，设种子点 *a* 和种子点 *b* 是下方子区域 *m* 的种子点，观察列 *y*（红框标注的那一列）可知，*a* 和 *b* 的 Voronoi 区域都进入到了上方子区域 *n* 中，而 *a* 和 *b* 在种子点队列中是连续的两个种子点，因此为确保子区域 Voronoi 图的正确性，需要同时将 *a* 和 *b* 的种子点信息传输到子区域 *n* 的种子点队列中才能正确的将子区域 *m* 和子区域 *n* 的 Voronoi 图拼合在一起。

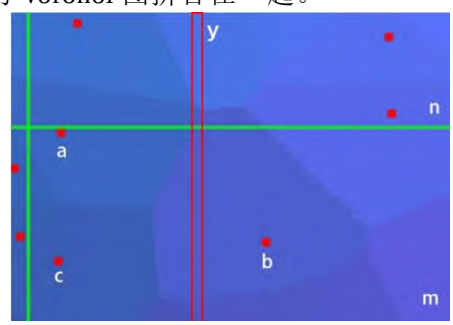


图 8　判断条件说明图

Fig. 8　Explanation diagram of judgment conditions

当所有子区域都被扫描过后，此时若直接根据每个子区域的种子点队列对子区域构建Voronoi图可发现仍有部分子区域的Voronoi图仍无法正确相连，这是由于各个子区域的种子点队列没有来自各自上方的子区域边界处的种子点信息，因此，为了让所有子区域的 Voronoi 图能够正确相连，在第二遍扫描中需要将上方子区域的边界种子点信息传递到下方子区域中，对所有子区域进行"从上向下"的扫描。以下是对其中一列子区域所执行的算法。

输入：已获得下方相邻区域种子点信息的子区域集合 $P=\{P_1,P_2,...P_n\}$，P 中的子区域按照从上到下的顺序排列。

输出：完成 Voronoi 图构建的子区域集合 $P=\{P_1,P_2,...P_n\}$

Step 1. 根据 P_1 的种子点队列构建 P_1 的 Voronoi 图，随后创建指针队列 $p=\{p_1,p_2,...,p_m\}$，将 P_1 每一列的种子点队列最后访问的元素位置记录到 p 中。

Step 2. 设置变量 $P_a=P_2$。

Step 3. 根据 p 中记录的指针位置从上方子区域的种子点队列中选取合适的种子点构建 P_a 的 Voronoi 图，直到相应的种子点队列所有的元素都被选取过作为离散点的种子点。

Step 4. 根据 P_a 本身的种子点队列继续构建 P_a 的 Voronoi 图，并记录每一列的种子点队列最后访问的元素位置。

Step 5. 将 P_a 每一列的种子点队列最后访问的元素位置更新到 p 中。

Step 6. 设置变量 $P_a=P_3$，重复步骤 Step3~Step5，直到 P 中所有的子区域都建立了 Voronoi 图。

其中 Step3 是链接两个子区域 Voronoi 图的关键步骤，当下方区域进行构建 Voronoi 图时先从上方种子点队列的末尾处选择合适的种子点，从而保证了种子点选取的连续性，并在 Step4 完成构建子区域的 Voronoi 图之后记录最后访问的位置，随后在 Step5 中更新指针信息，从而让下方的子区域能够继续访问本区域的种子点。

子区域 Voronoi 图的构建在 Step4 进行，这是由于当进行到 Step4 时，子区域已经接收到了来自四邻域边界处的种子点信息，因此可直接构建 Voronoi 图，将子区域的构建操作整合到扫描步骤中可以减少遍历次数。完成以上步骤后，所有子区域的 Voronoi 图均已构建完成，并且由于所有子区域的四邻域关系均已确认，各个子区域的 Voronoi 图能够正确的拼合在一起。

4.4 构建子区域的 Delaunay 三角剖分

得到子区域的 Voronoi 图后即可通过 Voronoi 图生成子区域的 Delaunay 三角剖分，并行访问每个子区域的 Voronoi 顶点，通过 Voronoi 顶点构造每个子区域的 Delaunay 三角剖分，当所有子区域的 Delaunay 三角剖分构建完成后，子区域的 Delaunay 三角剖分能够组合成完整的 Delaunay 三角剖分，如图 9 所示。

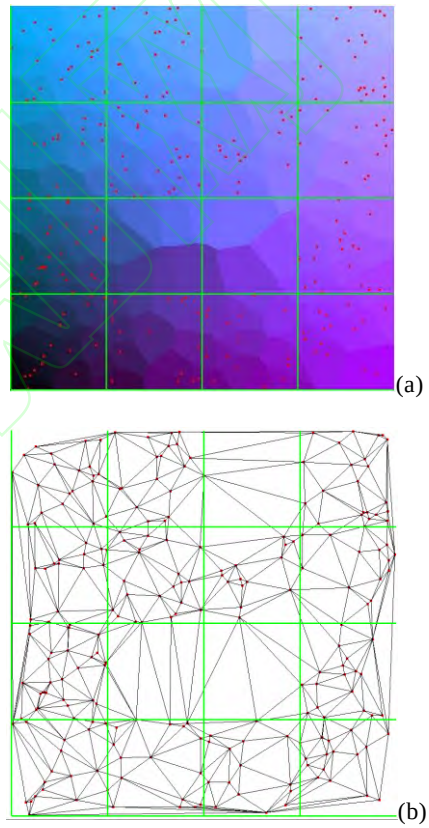


图 9 由子区域 Voronoi 图转化为 Delaunay 三角剖分示意图((a)子区域组合而成的完整 Voronoi 图；(b)由各个子区域的 Delaunay 三角剖分组合的完整 Delaunay 三角剖分)

Fig. 9 Transformation from sub-region Voronoi diagram to Delaunay triangulation((a) A complete Voronoi diagram composed of sub regions; (b) The complete Delaunay triangulation composed of Delaunay triangulations from each subregion)

5 计算着色器实现构网算法

使用计算着色器实现 Delaunay 构网算法分为以下几个步骤。

1) 在顶点缓存中创建一块区域用于存储点集的离散空间，随后将点集数据映射到该离散空间中，若存在重映射的点，则在离散空间中添加额外的离散行与离散列。

2) 为离散空间的每一个离散点分配一个工作组，使用第二节所述方法构建点集的一维 Voronoi 图。

3) 为一维 Voronoi 图的每一列分配一个工作组，对一维 Voronoi 图进行升维来计算出点集的离散 Voronoi 图。

4) 为离散 Voronoi 图的每一个离散点分配一个工作组，使用 2.3 所述方法由点集的 Voronoi 图构建点集的 Delaunay 三角网。

5) 在计算着色器中分配四个工作组，用于对构建好的 Delaunay 三角网进行补边。

其中 2)~5)步需要使用计算着色器进行并行计算，因此需要 glDispatchCompute 命令来调用计算着色器，为了保证各步骤之间能够按照顺序依次执行，还需要使用 glMemoryBarrier 来控制各个计算着色器之间的调度顺序。

对于第四节分区双向扫描算法的计算着色器实现，除了在构建 Voronoi 图与 Delaunay 三角剖分需要计算着色器进行并行计算外，在不同子区域间传递种子点信息的任务也需要使用计算着色器来完成。在构建子区域一维 Voronoi 图的步骤中，需要为子区域的每一行分配一个工作组，并行执行种子点传输与确认的任务；在构建二维 Voronoi 图的阶段，需要为子区域的每一列分配一个工作组，并行将边界处种子点信息发送到相邻的子区域中。

6 实验结果与分析

本文的实验环境为 Intel(R) Core(TM) i7-10870H CPU，内存为 16GB DDR4，显卡为 Nvidia GeForce RTX 2070 with Max-Q，操作系统为 Windows11，开发平台为 Microsoft Visual Studio 2022，开发语言采用 C++。

6.1 不同实现方法的对比测试

将使用计算着色器的方法与增量插入法，CGAL 的标准库方法以及使用 CUDA 的方法进行对比，其中 CUDA 和计算着色器使用相同的构网方法，且点集中不存在重映射点，结果见表 1。由表可知，当点集规模不大时，基于 CPU 的增量插入算法以及 CGAL 库的算法的时间效率要高于基于 GPU 的方法，原因在于使用 GPU 计算前需要对 GPU 进行初始化等操作，从而影响了运行时间。当点集数量增大时，凭借 GPU 强大的并行计算能力，基于 GPU 的方式的运行效率远大于基于 CPU 的方式。并且基于计算着色器的构网算法在时间效率上明显高于基于 CUDA 的构网算法。

表 1　不同方法所用时间对比(ms)

Table 1　Performance of different methods (ms)

点集数量(10^x)	增量插入法	CGAL	CUDA	计算着色器
1	0.3	**0.0434**	15	8
2	3	**0.1489**	19	11
3	139	**0.92**	23	17
4	11213	**10**	45	25
5	1778960	140	54	**33**
6	>1 day	1557	229	**108**
7	>1 day	22857	1101	**501**

注：加粗数据为最优值。

表 2 展示了 CUDA 和计算着色器处理相同点集时各阶段的时间消耗，从表中信息可知在数据的传输方面计算着色器有着明显的优势，且当数据集达到一定规模后，计算着色器有着更短的计算时间，这是由于计算着色器可以直接从渲染管线中读取数据，而 CUDA 读取数据需要进行频繁的数据传输，因此计算着色器相较于 CUDA 来说减少了数据传输的过程，从而节省了大量时间。

表 2　CUDA 和计算着色器各阶段用时对比(ms)

Table 2　Comparison of CUDA and Compute Shader (ms)

	CUDA			计算着色器		
点数(10^x)	导入时间	计算时间	导出时间	导入时间	计算时间	导出时间
4	20.8	3.4	21.9	6.5	6.65	12.5
5	25.5	3.8	27.4	6.8	7.13	20.2
6	102.3	10.6	126.9	38.5	8.67	64.9
7	425.8	35.1	688.9	196.2	10.7	295.04

6.2 动态插入法性能分析

将本文提出的动态插入法与现有的解决重映射的算法[25]做对比测试，为大小为 10000 的点集添加不同数量的随机重映射点，记录两种方法构网所需时间，结果见表 3。可以看出动态插入法有着更短的运行时间，这是由于动态插入法在时间复杂度上有着明显的效率优势，设离散空间中重映射点的数量为 n，则动态插入法则需要为

重映射点创建的离散行与离散列的数量为 $2n$，因此动态插入法的总时间复杂度为 O(n)，而现有的方法是使用增量插入法处理重映射点，增量插入法的时间复杂度是 O(n^2)，因此动态插入法与之相比有着明显的性能优势。同时动态插入法全程在 GPU 内完成计算，相对于现有方法减少了数据传输，从而进一步提高构网效率。

表 3　动态插入法与现有方法对比(ms)

Table 3　Comparison between Dynamic Insertion and Existing Methods (ms)

重映射点数量	100	300	500	700	900
现有方法	255	825	1149	1595	2056
动态插入法	38.2	43.8	49.2	56.5	63.6

6.3　计算着色器构网各阶段性能分析

表 4 展示了当点集数量发生变化时，构网各阶段用时的变化。从表中可看出，随着点集规模的提升，构网各阶段的用时都有所上升，其中二维 Voronoi 的生成与构建三角网这两个步骤的上升幅度最大，这是由于当点集规模增大时，容纳点集的离散空间也在相应增大，需要访问的离散点数量也随之增加，而这两个步骤需要访问的离散点数量相对较多，因此当点集数量增加时，这两个步骤相对于其他步骤来说提升的比例最大。

表 4　点集数量变化对构网时间的影响(ms)

Table 4　The effect of the change in the number of datasets(ms)

100	1k	10k	100k	1m	10m
0.5716	0.714	0.7193	0.8966	0.9918	1.024
1.6568	1.676	1.9092	2.4387	3.2204	4.337
1.8372	1.85	2.1182	2.3203	2.6523	5.82
1.9263	2.081	2.0088	2.2599	2.7129	3.564
5.9919	6.321	6.7555	7.9155	9.5774	14.74

6.4　分区双向扫描算法性能分析

表 5 展示的是当分区数量不同时，使用分区双向扫描算法对 3000 万数量的点集进行构网所需要的时间。可以看出增加分区数量会提高整体构网时间，这是因为分区数增加时，需要处理的边界也随之增加，同时更多的分区意味着需要频繁的使用计算着色器对分区进行构网，而每一次使用计算着色器都会产生额外的性能开销，因此在实际使用中，需要尽可能的减少分区的数量，从而减少构网所需要的时间。

表 5　分区数量变化对构网时间的影响(ms)

Table 5　The effect of the change in the number of partitions (ms)

分区数	4	16	64	256
横向扫描各个子区域	8.4803	11.298	34.067	131.816
构建一维 Voronoi 图	1.8149	6.1011	28.302	102.074
纵向扫描各个子区域	6.3424	12.878	49.902	179.477
构建二维 Voronoi 图	8.0842	25.822	135.4	448.633
构建三角网	542.13	716.95	922.73	1130.42
修补三角网	142.65	140.16	143.45	142.321
耗时总和	709.49	913.2	1313.8	2134.74

6.5　分区双向扫描算法时间复杂度分析

从时间复杂度的角度对分区双向扫描算法进行分析，假设将离散空间大小为 $a*a$ 的点集划分为了 $m*m$ 的分区，在对子区域进行横向扫描的阶段，由于每一个分区都需要进行向左向右的两次扫描，所需要进行的运算次数为 $2*m(m-1)$，在分区内部，每一次确认边界种子点所需要进行的遍历次数为 a/m，因此横向扫描阶段总计需要 $2*a(m-1)$次运算。在纵向扫描阶段，由于将自上而下的扫描整合到了二维 Voronoi 图构建步骤中，因此纵向扫描只需要 $m*(m-1)$次运算，而每个子区域进行构网的时间复杂度为 O(a/m)，由上述信息可知分区双向扫描算法的时间复杂度为 O(am)，而现有的以分区为主要思想的算法的时间复杂度约为 O($nlogn$)，其中 n 为点集数量。因此从时间复杂度的对比可以得出，分区双向扫描算法的时间复杂度与子区域的大小与分区数量有关，而与点集数量无关，因此当数据集规模较大时，本文提出的分区双向扫描算法会体现出明显的性能优势。

当出现少量的点集数据出现在大规模的离散空间的情况会出现运行效率下降的情况。针对这种情况，可以对离散空间中的点进行再次映射，将大离散空间中的点映射到小规模的离散空间中，对小规模的离散空间进行计算，从而加快构网速度。

7　结语

本文提出了一种通过计算着色器来实现 Delaunay 三角剖分的算法，该算法与传统串行算法以及使用 CUDA 的并行三角剖分算法相比在

运行效率上有明显的性能提升，同时计算着色器脱离 CUDA 进行 GPU 通用计算的特性也变相提高程序的跨平台能力，为 Delaunay 三角剖分算法的实现提供了一个新的方向，并且本文提出的动态插入法也能很好的解决重映射问题。此外，本文所提出的分区双向扫描算法能够很好的对数据量超出 GPU 的运行上限的数据进行构网，从而避免了因 GPU 的瓶颈导致的构网错误。但是对于各个分区的扫描仍是通过串行的方式进行扫描，未来工作可以通过使用多 GPU 的工作站对各个分区进行并行扫描，从而进一步提升构网速度。

参考文献(References):

[1] 袁清洌, 吴学群. 结合 Delaunay 三角面分离法与搜索球策略的三维曲面重建算法[J]. 图学学报, 2018, 39(2): 278-286.
YUAN Q L, WU X Q. 3D surfaces reconstruction algorithm via detaching Delaunay triangular mesh and search-ball approach[J]. Journal of Graphics, 2018, 39(2): 278-286 (in Chinese).

[2] MULCHRONE K F. Application of Delaunay triangulation to the nearest neighbour method of strain analysis[J]. Journal of Structural Geology, 2003, 25(5): 689-702.

[3] 童立靖, 李嘉伟. 一种基于改进 PointNet++网络的三维手姿估计方法[J]. 图学学报, 2022, 43(5): 892-900.
TONG L J, LI J W. A 3D hand pose estimation method based on improved PointNet++[J]. Journal of Graphics, 2022, 43(5): 892-900 (in Chinese).

[4] KANG D S, KIM Y J, SHIN B S. Efficient large-scale terrain rendering method for real-world game simulation[C]//The 1st International Conference on E-Learning and Games. Cham: Springer, 2006: 597-605.

[5] MAVRIPLIS D J. Adaptive mesh generation for viscous flows using triangulation[J]. Journal of Computational Physics, 1990, 90(2): 271-291.

[6] NANDURI J R, PINO-ROMAINVILLE F A, CELIK I. CFD mesh generation for biological flows: geometry reconstruction using diagnostic images[J]. Computers & Fluids, 2009, 38(5): 1026-1032.

[7] DE FLORIANI L, FALCIDIENO B, PIENOVI C. Delaunay-based representation of surfaces defined over arbitrarily shaped domains[J]. Computer Vision, Graphics, and Image Processing, 1985, 32(1): 127-140.

[8] BOWYER A. Computing Dirichlet tessellations[J]. The Computer Journal, 1981, 24(2): 162-166.

[9] LIN J X, CHEN R Q, YANG C C, et al. Distributed and parallel Delaunay triangulation construction with balanced binary-tree model in cloud[C]//The 2016 15th International Symposium on Parallel and Distributed Computing. New York: IEEE Press, 2016: 107-113.

[10] ZHOU S, JONES C B. HCPO: an efficient insertion order for incremental Delaunay triangulation[J]. Information Processing Letters, 2005, 93(1): 37-42.

[11] FORTUNE S. A sweepline algorithm for Voronoi diagrams[C]//The 2nd Annual Symposium on Computational Geometry. New York: ACM, 1986: 313-322.

[12] ŽALIK B. An efficient sweep-line Delaunay triangulation algorithm[J]. Computer-Aided Design, 2005, 37(10): 1027-1038.

[13] BINIAZ A, DASTGHAIBYFARD G. A faster circle-sweep Delaunay triangulation algorithm[J]. Advances in Engineering Software, 2012, 43(1): 1-13.

[14] LEE D T, SCHACHTER B J. Two algorithms for constructing a Delaunay triangulation[J]. International Journal of Computer & Information Sciences, 1980, 9(3): 219-242.

[15] SONG Y, LI M, LIU X. A paralleled Delaunay triangulation algorithm for processing large LIDAR points[J]. ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, 2022, 10: 141-146.

[16] NGUYEN C, RHODES P J. TIPP: parallel Delaunay triangulation for large-scale datasets[C]//The 30th International Conference on Scientific and Statistical Database Management. New York: ACM, 2018: 8.

[17] NAVARRO C, HITSCHFELD-KAHLER N, SCHEIHING E. A parallel GPU-based algorithm for Delaunay edge-flips[EB/OL]. [2024-04-28]. https://scholar.google.com/scholar?hl=zh-CN&as_sdt=0%2C5&q=A+parallel+gpu-based+algorithm+for+delaunay+edge-flips&btnG=#d=gs_cit&t=1728983123697&u=%2Fscholar%3Fq%3Dinfo%3AtcrG4K4tgnsJ%3Ascholar.google.com%2F%26output%3Dcite%26scirp%3D0%26hl%3Dzh-CN.

[18] HOFF III K E, KEYSER J, LIN M, et al. Fast computation of generalized Voronoi diagrams using graphics hardware[C]//The 26th Annual Conference on Computer Graphics and Interactive Techniques. New York: ACM, 1999: 277-286.

[19] RONG G D, TAN T S. Jump flooding in GPU with applications to Voronoi diagram and distance transform[C]//2006 Symposium on Interactive 3D Graphics and Games. New York: ACM, 2006: 109-116.

[20] CAO T T, TANG K, MOHAMED A, et al. Parallel banding algorithm to compute exact distance transform with the GPU[C]//2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. New York: ACM, 2010: 83-90.

[21] VA H, CHOI M H, HONG M. Real-time cloth simulation using compute shader in Unity3D for AR/VR contents[J]. Applied Sciences, 2021, 11(17): 8255.

[22] MOWER J E. Real-time drainage basin modeling using concurrent compute shaders[EB/OL]. [2024-04-28]. https://www.albany.edu/~Jmower/Publications/MowerAC2016_PaperSub.pdf.

[23] MIHAI C C, LUPU C. Using graphics processing units and compute shaders in real time multimodel adaptive robust control[J]. Electronics, 2021, 10(20): 2462.

[24] RONG G D, TAN T S, CAO T T, et al. Computing two-dimensional Delaunay triangulation using graphics hardware[C]//2008 Symposium on Interactive 3D Graphics and Games. New York: ACM, 2008: 89-97.

[25] QI M, CAO T T, TAN T S. Computing 2D constrained Delaunay triangulation using the GPU[C]//ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. New York: ACM, 2012: 39-46.