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）
21301157	林楷	8	25	17	35	85

北京交通大学

本科《计算机图形学》课程论文

探索在多边形光栅化中基于块的遍历采样算法

Exploring Block-Based Traversal Sampling Algorithms in Polygon Rasterization

学　　院：软件学院

专　　业：软件工程

学生姓名：林楷

学　　号：21301157

指导教师：吴雨婷

北京交通大学

2024 年 6 月

中文摘要

摘要：光栅化是计算机图形学领域的一个关键过程，将矢量描述转换为光栅格式。传统方法通常涉及评估边界框内的每个单独像素，以确定它是否位于目标形状内，或通过使用遍历算法对其进行优化。然而，由于对非重叠区域进行不必要的计算，这种方法可能会导致性能损失。为了解决这个问题，本文提出了一种创新的基于块的遍历采样算法用于光栅化。该技术将边界框划分为多个块，根据分离轴定理计算每个块与目标二维多边形之间的重叠。只有重叠的块通过遍历算法进行评估，然后进行采样，有效地省略了非重叠区域的计算。本文中的算法结合了这两种方法的优点，并通过类似于块稀疏算法的方法进行了进一步优化。通过这种方式，提出的方法显著降低了计算开销并提高了性能，从而为计算机图形学中的高效光栅化提供了一种有前景的替代方案。

关键词：光栅化；遍历采样；重叠块

ABSTRACT

ABSTRACT: Rasterization is a key process in the field of computer graphics, which converts vector descriptions into raster formats. Traditional methods typically involve evaluating each individual pixel within the bounding box to determine whether it is within the target shape, or optimizing it using traversal algorithms. However, due to unnecessary calculations on non overlapping regions, this method may result in performance loss. To address this issue, this paper proposes an innovative block based traversal sampling algorithm for rasterization. This technique divides the bounding box into multiple blocks and calculates the overlap between each block and the target 2D polygon based on the separation axis theorem. Only overlapping blocks are evaluated through traversal algorithms and then sampled, effectively omitting the calculation of non overlapping regions. The algorithm in this article combines the advantages of these two methods and is further optimized through a method similar to block sparsity algorithm. Through this approach, the proposed method significantly reduces computational overhead and improves performance, providing a promising alternative to efficient rasterization in computer graphics.

KEYWORDS： rasterization; traversal sampling; overlapping blocks

目　　录

1 引言

三维 Z 缓冲线性插值多边形的快速绘制是计算机图形学中的一个基础问题。这个问题通常由两个部分组成：

（1）顶点的三维变换、投影和照明计算。

（2）将多边形光栅化到帧缓冲区上。

本文专门讨论了第二部分的一个方面，即多边形边界的计算。

叉积运算

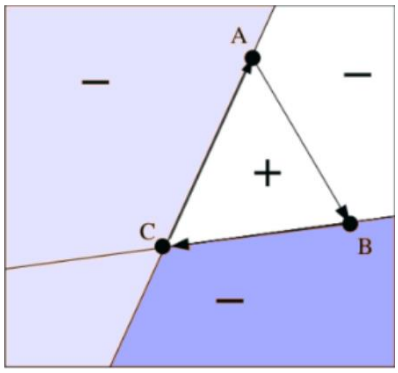

图 1-1 三角形可以由边的组合形成

图 1-1 显示了如何通过边函数指定的三条边的并集来定义三角。通过使用三条以上边的布尔组合，可以定义更复杂的多边形。传统上，每个像素都被视为一个点，以评估其与多边形的位置关系。这是通过对多边形的边进行叉积运算来实现的，从而确定点所在的每条边的边。这最终有助于我们了解像素是否位于三角形内。[1]

包围盒算法

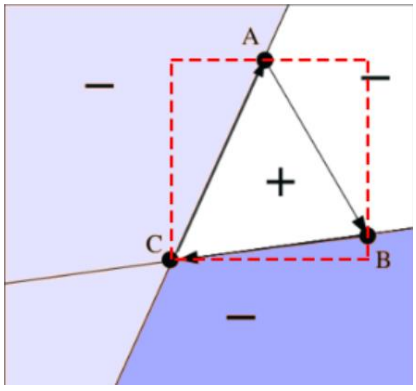

图 1-2 包围盒算法示意图

图 1-2 展示了包围盒算法，简单地说，它是一种将多边形封装在一个足够大的矩形中的方法。[2]通过创建包围盒，可以减少不必要的计算，因为它消除了对框外像素进行计算的需要。通过单独定位包围盒，可以显著节省性能。

遍历算法

随着遍历算法的发展，通过修改遍历顺序实现了进一步的优化。该算法保证了多边形中所有像素的覆盖率。图 1-3 展示了两种基本算法。最简单的策略是遍历边界框；然而，它通常不是最有效的。一个更智能的算法会在下一条线超过三角形边缘后移动到下一条线上。

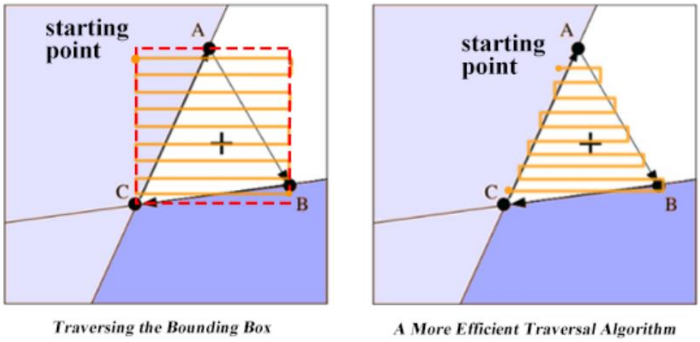

图 1-3 简单的遍历算法

这种智能算法有一个重要的复杂性，当它前进到下一条线时，它可能会进入三角形。在这种情况下，算法必须在启动下一条扫描线之前首先寻找边缘的外部。[1]这个问题的一个例子在图 1-4 中三角形的右上角有所展示。

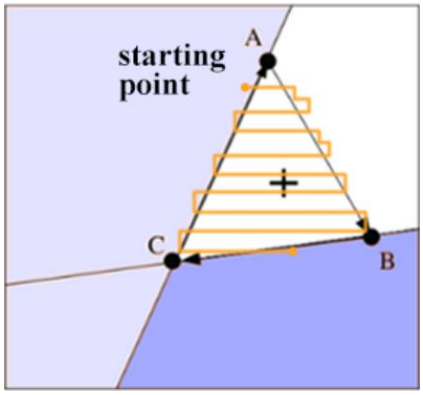

图 1-4 遍历算法可能必须搜索边缘

一个更智能的算法如图 1-5 所示。该算法从起点开始下降，从中心线开始逐渐扩展。与更简单的算法相比，这种算法的优势在于，它从不需要寻找边，然后回溯。无论如何，折衷的办法是，在遍历外部点时，必须保留中心线的插值器状态，因为插值器必须在中心线处重新启动。需要注意的是，在底部，如果“中心”线最终位于三角形的外部，则会发生偏移。

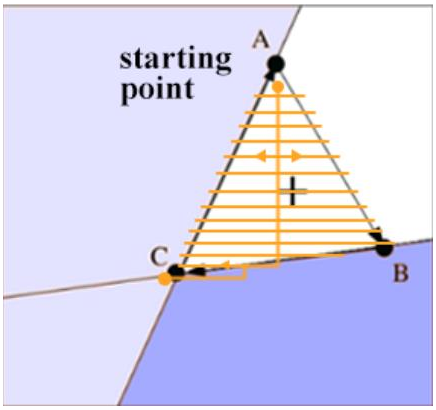

图 1-5 更智能的算法从中心线向外推进

块稀疏算法

尽管遍历算法也考虑了切片进行并行优化，但它没有详细说明如何选择相应的切片。因此，本文在借鉴 OpenAI 的块稀疏算法的基础上，引入了基于块的遍历算法来优化这一过程。

图 1-6 显示了块稀疏算法的基本思想，它通过将计算集中在输入矩阵的高幅度块上来提高计算效率。它将矩阵划分为几个块，然后基于某个阈值，只处理包含重要值的块，而跳过那些具有低值的块。[3]

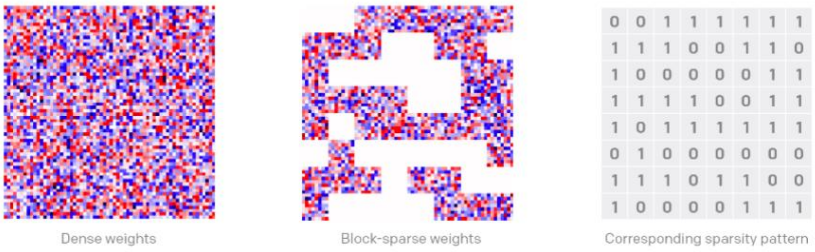

图 1-6 OpenAI：块稀疏 GPU 内核

应用类似的概念，我们计算边界框中的每个块矩形是否与多边形重叠，以确定是否应选择该块进行遍历算法计算。两个多边形之间的重叠可以通过检查每个块直角的任何边是否与目标多边形的任何边相交来确定。如果它们相交，那么块不可否认地与多边形重叠。

因此，该算法的流程包括确定边界框，根据块的大小划分边界框，然后通过使用分离轴定理[4]判断每个块是否重叠来生成块掩码布尔矩阵。接下来，遍历算法并行应用于每个边界框，生成最终采样结果。

2 基于块的遍历采样实验方法描述

初始化

首先，用布尔值初始化二维字段（数组），其大小等于输入字段的大小（field_size）。所有值最初都设置为 False（0）。

$$field = np.zeros(field\_size, dtype = np.bool\_)$$

包围盒生成

接下来，使用 generateBoundingBoxes 函数为输入多边形列表（polygonList）中的所有多边形生成边界框列表。边界框是包含多边形的最小矩形框。边界框角点的坐标由多边形顶点的最小值和最大值 x 和 y 确定。

块掩码生成

对于每个多边形，将计算块的布尔掩码，指示多边形与哪些块相交。这是通过调用函数 calculateMaskFromPolygon 来实现的。对于多边形边界框内的每个块，都会定义一个矩形，并使用函数 isRectangleInPolygon 检查其与多边形的交点。此函数的输出是一个布尔值，指示是否存在交集。

字段遍历

生成多边形的掩码后，使用函数 traverseMask 基于掩码遍历和修改字段。对于掩码中为 True 的每个块（即它与多边形相交），使用函数 traverseBlock 进行更详细的遍历。在该函数中，算法在多边形内的块内搜索起点，然后扫描块，在字段中标记多边形内的点。这是通过向右移动直到到达多边形或块的边缘，然后向下移动到下一行来重复该过程，直到扫描整个块。

即时编译

为了提高算法的性能并使算法并行，使用了 NumPy 和 Numba 库，特别是用@jit 和@njit 修饰的函数。这允许将 Python 代码实时（JIT）编译为机器代码，从而加快执行时间。

3 基于块的遍历采样实验过程

测试简介

光栅化方法的功能和性能在三个不同复杂度的多边形列表上进行了测试，每个列表都有不同的字段大小。多边形列表被定义为 numpy 数组，每个数组包含多个 2D 数组，每个 2D 数组表示由其顶点指定的多边形。字段大小从 600 x 600 到 2400 x 2400 不等。

此外，本文还评估了改变块大小对基于块的遍历方法性能的影响。所考虑的砌块尺寸从 25 x 25 到 400 x 400 不等。

为了更合理的测试，还对遍历算法进行了并行处理，这两种算法都采用相同的块大小来分割块。

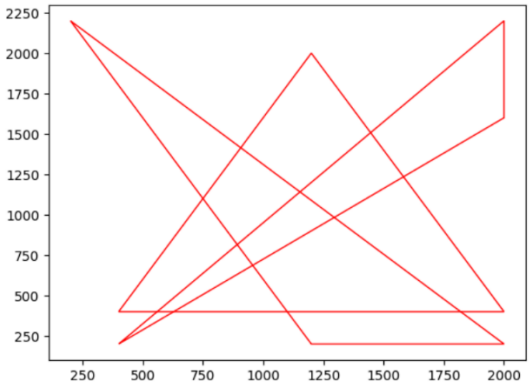

图 3-1 要光栅化的原点三角形

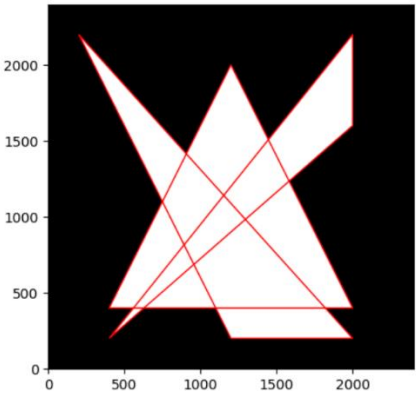

图 3-2 光栅化的结果

图 3-1 显示了要光栅化的原点三角形。字段大小为 2400 x 2400。图 3-2 是光栅化的结果。

测试流程

测试流程如下：

（1）使用 initPolygonList_custom 函数初始化自定义多边形列表。

（2）基于块的遍历方法运行两次：第一次是为了预热 JIT 编译器，第二次是为了测量执行时间。输出是一个布尔字段，指示每个像素是否在多边形内以及经过的时间。

（3）类似地，遍历方法运行两次，同时捕获布尔字段和经过的时间。

对多边形列表、字段大小和块大小的每个组合重复此测试流程。

测试环境

测试环境如下：

（1）CPU: 12th Gen Intel(R) Core(TM) i7-12700H　　2.30 GHz

（2）GPU: Intel(R) Iris(R) Xe Graphics

（3）OS: Windows 11

（4）Python version: 3.8.8

（5）NumPy version: 1.20.1

测试结果

从图 3-3 中所示的实验结果中，我们可以对并行遍历和基于块的遍历这两种算法的性能得出一些结论。

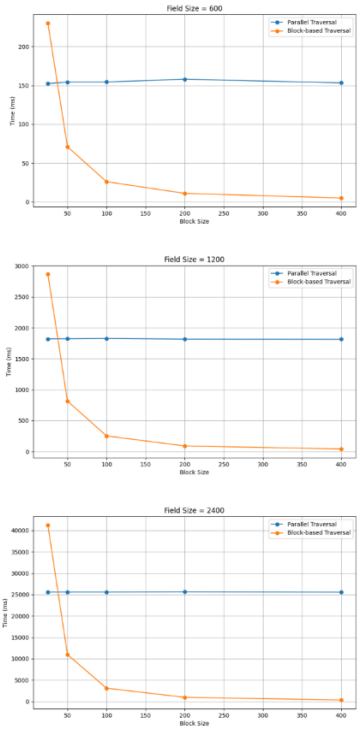

图 3-3 两种算法的时间随块大小而变化

首先，当我们观察并行遍历算法所花费的时间时，很明显，无论字段大小如何，当块大小发生变化时，该值都不会显示出任何显著的波动。这与我们对算法的理解一致，因为并行遍历在其设计中没有包含块大小的概念。因此，更改块大小不会直接影响其执行时间。

接下来，我们将注意力转向基于块的遍历算法。从数据中，我们可以看到执行时间随着块大小的增加而减少的明显趋势。当我们比较相同的字段大小时，这一点尤其明显。这种行为背后的原因是，基于块的遍历算法通过将像素分组为块，可以减少当块完全位于多边形内部或外部时的检查操作次数，从而提高光栅化的效率。随着块大小的增加，出现这种情况的可能性更高，因此执行时间减少。

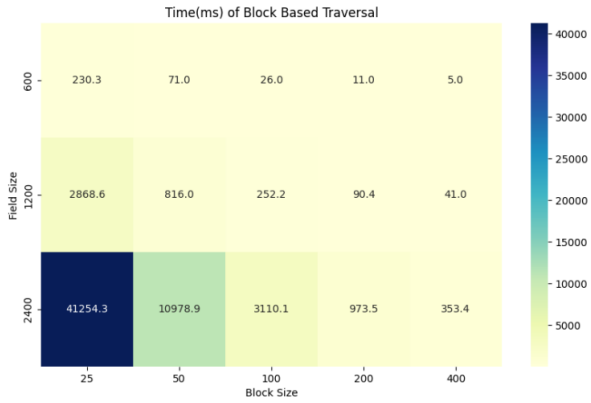

图 3-4 基于块的遍历算法的时间（单位：ms）

总之，与并行遍历算法相比，基于块的遍历算法在效率方面显示出明显的优势，尤其是当为不同的字段大小选择最佳块大小时。然而，在现实应用程序中，考虑块大小和字段大小之间的权衡以最大限度地提高性能是至关重要的。

4 结论

正如 Juan Pineda 所说：“可能有很多遍历算法。最佳算法将取决于实现中的成本/性能权衡。”[1]尽管使用基于块的遍历算法在时间复杂性方面有明显优势，但需要消耗额外的内存来存储掩码信息。此外，该算法仅模拟简单的光栅化，该光栅化仅判断像素是否在多边形中。

光栅化还有很多步骤，这只是其中之一；使基于块的遍历算法适应其他步骤（如抗锯齿）是未来将要考虑的问题。

事实上，在研究的最初阶段，很难找到近年来进行的关于多边形光栅化的研究。让人猜测这个问题可能已经解决了。本人把这归因于两个主要因素。

首先，在计算机图形学领域，几乎每个像素都基本上包含在特定的多边形中，这使得场景异常复杂。在这种情况下，直接遍历每个点可能比使用块优化更有效。这是因为基于块的遍历算法需要额外的计算来确定每个块和多边形之间的关系。一旦多边形的数量增加，所谓的好处可能会变得微不足道。

其次，实验仅在基于 CPU 的环境中进行。然而，光栅化任务通常是使用 GPU 计算的。随着 GPU 的快速发展，选择更高效的算法变得不那么重要。这是由于 GPU 能够并行化数百个（如果不是数千个的话）流处理器，将计算时间减少到 10 毫秒以下。任何进一步的优化都可能带来难以察觉的改进。

参考文献

[1] Pineda and Juan. “A parallel algorithm for polygon rasterization”. In: Acm Siggraph Computer Graphics 22.4 (1988), pp. 17–20.

[2] Yunhong et al. “Analysis of a bounding box heuristic for object intersection”. In: Journal of the ACM 46.6 (1999), pp. 833–857.

[3] Gray S., Radford A., and Kingma D.P. GPU kernels for block-sparse weights. 2017. url: https : / / cdn . openai . com / blocksparse /blocksparsepaper.pdf.

[4] Gottschalk Stefan, M.C. Lin, and D. Manocha.“OBBTree: A Hierarchical Structure for Rapid Interference Detection”. In: ACM SIGGRAPH Computer Graphics 30 (1996).