RasterPIP: answering point-in-polygon query with GPU-native transformation and rasterization

Ziang Liu
School of Cyber Engineering, Xidian University
Xi'an, China
zaliu1@stu.xidian.edu.cn

Hui Li
School of Computer Science and Technology, Xidian University
Xi'an, China
Shanghai Yunxi Technology Co, Ltd.
Shanghai, China
hli@xidian.edu.cn

Yingfan Liu
School of Computer Science and Technology, Xidian University
Xi'an, China
liuyingfan@xidian.edu.cn

Hua Tong
School of Cyber Engineering, Xidian University
Xi'an, China
htong@stu.xidian.edu.cn

Zhenning Shi
School of Computer Science and Technology, Xidian University
Xi'an, China
znshi@stu.xidian.edu.cn

Hui Zhang
Shandong Inspur Database Technology Company Limited
Jinan, China
zhanghui@inspur.com

Jiangtao Cui
School of Computer Science and Technology, Xidian University
Xi'an, China
cuijt@xidian.edu.cn

ABSTRACT

Point-in-polygon (PIP) query is a fundamental problem in spatial-temporal databases, which aims to return a series of points covered by a (set of) given polygon(s). Extensive applications of this query can be easily found in a series of fields, including computer graphics, geographic information systems (GIS), spatial databases, etc. However, current solutions either require expensive computational geometry methods for PIP tests or struggle to efficiently distinguish the IDs of points within polygons. To address these limitations, inspired by recent progress in GPU-accelerated query processing, we propose a pair of GPU-native approaches, namely RasterPIP-b and RasterPIP-i, towards a pair of popular PIP query tasks, respectively. Our key idea is to convert PIP tests into a series of object rendering operations on a canvas, leveraging the native transformation and rasterization capabilities of the GPU to improve the efficiency. For each approach, we judiciously choose the storage mode for the essential information within the GPU to distinguish point IDs, while minimizing unnecessary resource waste. In order to demonstrate the performance, we conduct extensive experiments on real datasets. The results show that our method outperforms existing solutions by one to three orders of magnitude in terms of efficiency.

ACM Reference Format:
Ziang Liu, Hui Li, Yingfan Liu, Hua Tong, Zhenning Shi, Hui Zhang, and Jiangtao Cui. 2024. RasterPIP: answering point-in-polygon query with GPU-native transformation and rasterization. In *Proceedings of EDBT2025*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Due to the widespread use of location sensing technologies, there has been a significant increase in spatial-temporal data. We can analyze and extract a wealth of valuable insights by effectively managing and querying these spatial-temporal data [6, 15, 20, 28, 43, 47]. As a fundamental problem in spatial-temporal queries, point-in-polygon (PIP) queries have been widely applied in many fields [26, 29, 36] , such as computer graphics, geographic information systems (GIS), spatial databases, and *etc.*. For example, suppose that a large supermarket is delivering coupons to some specific residential regions, as shown in Figure 1. In this example, the advertiser has to at least identify all users (may not need to tell which region a user belongs to) within these target regions so that the coupons can be sent towards. Furthermore, users from different residential regions might have a different consumption custom. To boost sales, the supermarket needs to recognize in which regions an arbitrary user falls, customizing the type and quantity of coupons specifically for users in each region. To be formal, each user could be formed as a spatial point that represents its current location, while each region can be viewed as a spatial polygon. Let D be the set of spatial points and P be the set of spatial polygons; thus finding users located in specific target regions is, in fact, asking for 1) the set of points in D falling in at least one polygon in P; 2) the complete mapping between $o \in D$ and $p \in P$ such that o falls within p. Note that, the PIP problem here deals with multiple points and polygons simultaneously. In order to distinguish the two scenarios listed above, we

EDBT2025, March 25–28, 2025, Barcelona

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

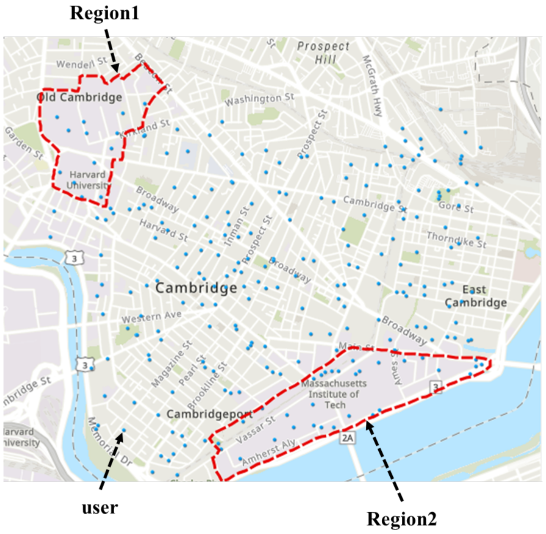


Figure 1: An example of PIP query in practice.

call them boolean PIP (bPIP) and integral PIP (iPIP), respectively. To make it clear, we illustrate both in the form of SQL queries as follows. Obviously, both are spatial selection queries that involve a spatial join between D and P, but with a different projection list.

```
Query 1 (example of bPIP):
SELECT Distinct D.id
FROM D, P
WHERE P.region CONTAINS D.location;

Query 2 (example of iPIP):
SELECT D.id, P.id
FROM D, P
WHERE P.region CONTAINS D.location
Order By P.id;
```

Several solutions to PIP have been proposed [18, 19, 22, 49], among which RayCasting [19] is the most popular. The distinct geometric properties exhibited by points inside and outside a polygon are used to test if a point is inside a polygon, which is known as a PIP test. Therefore, we can decompose both bPIP and iPIP into a series of PIP tests via RayCasting. However, it still suffers from scalability issues. First, its cost is linear to the number of sides in each given polygon, which is expensive when the polygons are associated with many sides. Second, its efficiency is sensitive to both the number of points and polygons, since it has to perform the PIP test for each pair of point and polygon, which make it costly when dealing with large dataset. An enhanced version of RayCasting is to employ an efficient spatial index such as R-tree [21] or QuadTree [17], combined with the Minimum Bounding Rectangle (MBR) of query polygons, to pre-filter points that are definitely outside the polygons, thereby reducing the number of PIP tests needed. This idea has been adopted and implemented by PostGIS[1], a popular open-source software that supports geographic objects on the basis of PostgreSQL. The advantages of PostGIS over RayCasting lie in the filtering ability of the spatial index, making it suitable for datasets with few polygons or small polygon coverage areas. However, when the data scale and polygon coverage areas are large, the filtering ability of spatial index will be diminished. In this case, the spatial indexing methods still require a large amount of PIP tests, resulting in no performance advantage or even worse [1, 31].

Recently, GPU-friendly approaches [9, 27, 48] have attracted research attention. RasterJoin [48] is the pioneering work in answering spatial *aggregation* queries as shown in **Query 3** below.

```
Query 3:
SELECT count(*)
FROM D, P
WHERE P.region CONTAINS D.location
Group by P.id;
```

RasterJoin proposed to exploit GPU-native transformation and rasterization. One of its advantages is that it can directly output aggregate values without explicitly materializing the intermediate results of spatial joins. RasterJoin utilizes the color channel of each pixel to store aggregated information associated with that pixel. However, this paradigm cannot directly answer spatial selection queries such as **Query 1** and **Query 2**. In the context of aggregation queries, the information stored in each pixel does not need to be distinguishable, such as COUNT and SUM. In comparison, **Query 1** and **Query 2** need to output the IDs of the points within the polygons. Therefore, it is necessary to ensure that the values in the color channel can accurately distinguish the IDs of different points. A simple way is to represent IDs by one-hot encoding. However, given the limited bit-depth of color channels and possibly large point IDs, following this method poses grand challenges. This is an inevitable issue in practical applications. In [9], a GPU-friendly geometric model and algebraic system are designed, demonstrating strong expressiveness. Their work fully showcases the advantages of utilizing the native rendering capabilities of the GPU for spatial data queries. However, it does not provide a direct solution for distinguishing multiple data points on the same pixel. GPU-Reg [27] transforms spatial selection queries into spatial aggregation queries through an encoding method, enabling RasterJoin to solve the PIP problem. However, it suffers from low efficiency because it complicates the query processing.

Undoubtedly, the parallel computational framework of GPU has great potential to efficiently answer PIP queries. The key challenge here is how to make full use of the GPU's limited built-in variables to efficiently store and distinguish crucial information, *i.e.,* the IDs, which cannot be solved by simply adding additional variables. To overcome the above challenges, inspired by RasterJoin and GPU-Reg, we introduce RasterPIP, a novel approach using GPU-native transformation and rasterization to solve both bPIP and iPIP. Following the paradigm in RasterJoin, we transform the PIP test into a canvas test on the GPU. For various PIP problems, we judiciously choose the storage mode for the essential information within the GPU. This allows us to distinguish point IDs while minimizing unnecessary resource consumption, thereby fully leveraging GPU-native operators for efficient query responses. We conduct extensive experiments on real-world datasets and compare our methods with state-of-the-art solutions, including CPU-based and GPU-based ones. According to our experimental results, our methods achieve

[1] http://postgis.net/

a speedup of one to three orders of magnitude over the existing solutions.

Our technical contributions in this work can be summarized as follows:

- We propose a pair of novel GPU-native solutions, namely RasterPIP-b and RasterPIP-i, to solve bPIP and iPIP, respectively.
- We theoretically prove that both RasterPIP-b and RasterPIP-i are no longer sensitive to shape, area or the number of query polygons.
- We implement and integrate our approach into existing spatial database systems and conduct extensive experiments on real-life data to demonstrate the advantages of our methods.

The rest of this work is organized as follows. In Section 2, we introduce the preliminaries required for our solution. Afterwards, we present the details of RasterPIP-b and RasterPIP-i in Section 3 and Section 4, respectively. Experimental results are reported in Section 5. We review related works in Section 6. Finally, in Section 7 we conclude this paper.

2 PRELIMINARIES

In this section, we first formalize the two PIP problems, and then introduce the native GPU operations exploited in our solution.

2.1 Problem definition

Definition 2.1. **Boolean Point-in-Polygon (bPIP) Query.** Given a set $D \subset \mathbb{R}^d$ and a set P of polygons in $\mathbb{R}^d$, bPIP query returns a set $R \subset D$, such that $\forall o \in R$, $\exists p \in P$ and p contains o.

That is, we need to tell whether or not an arbitrary point o is contained by P and hence it is referred to as *boolean* PIP problem. This problem can be easily found in various applications [27]. Differently, in some other applications [16], we may also wonder *which* polygon o is contained by, as shown in the following.

Definition 2.2. **Integral Point-in-Polygon (iPIP) Query.** Given a set $D \subset \mathbb{R}^d$ and a set P of polygons in $\mathbb{R}^d$, iPIP query returns a set $S = \{(o, p) | o \in D, p \in P \text{ and } p \text{ contains } o.\}$

Compared with bPIP, iPIP returns the polygon IDs, *i.e.,* integers, each point o is contained by, instead of whether or not. Therefore, it is referred to as Integral PIP. In the above definition, d represents the dimension of the point and polygon. Since PIP queries are typically conducted in the plane, the algorithms discussed in this paper are primarily applicable to two-dimensional spaces, *i.e.,* $d = 2$.

2.2 GPU-native operations

In this part, we introduce two operations that are adopted in our solution, namely point drawing and polygon drawing, both of which are natively supported by GPU. In PIP, we have to deal with two spatial objects, *i.e.,* point and polygon, respectively. To employ the GPU-native operations, we first embed points and polygons from the original spatial space to the coordinates in the screen canvas and then find efficient ways to determine their mappings at runtime.

Polygon Drawing. Drawing a polygon by the GPU is called polygon rendering, which is done by first decomposing a polygon into a set of triangles and then rendering those triangles on a screen with a specific resolution. There are a bulk of works that decompose a polygon into triangles. In this work, we employ the constrained Delaunay polygon triangulation [39] to solve this problem.

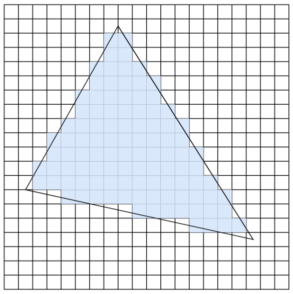


Figure 2: An example for rasterizing a triangle.

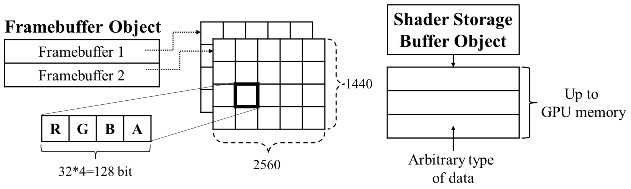


Figure 3: Illustration of FBO and SSBO.

Now, let us consider the rendering of a triangle, which is essentially the process of converting a triangle into a collection of pixels on the screen. First, the vertices of the triangle are projected onto the screen space from the original spatial space. Part of the triangle may fall outside the screen, which will be automatically discarded. Subsequently, the GPU rasterizes the part of the triangle that falls inside the screen space. Rasterization is an important module of the graphics rendering pipeline [33]. As shown in Figure 2, a triangle is transformed into a set of fragments through rasterization, each of which corresponds to a pixel in the screen space. Note that only the fragment whose center is covered by the triangle will be counted. Then, each fragment is assigned a pre-defined color for display on the screen. A screen is considered as a matrix of pixels, and its resolution determines the total number of pixels. In this paper, a fragment can be considered a pixel. Therefore, in the following discussion, we shall interchangeably use pixel and fragment.

As a cross-language and cross-platform API for rendering graphics, OpenGL [25] provides two types of programmable shader, the *i.e.,* vertex shader and the fragment shader for rendering a triangle. The former conducts the transformation of each vertex to the screen space, while the latter processes the attributes of fragments covered by the triangle, such as fragment colors.

Point Drawing. As another key operation concerned in this work, the process of drawing a point is similar to drawing a vertex of a triangle, as discussed above. According to OpenGL, we can follow the vertex shader program to implement this procedure. To answer bPIP and iPIP, we do not need to physically display the rendering result on the screen, but we need modules provided by OpenGL to store the immediate results *w.r.t.* each pixel during rendering. Luckily, we find **framebuffer object (FBO)** and **shader storage buffer object (SSBO)** in OpenGL for this purpose. Their structures are shown in Figure 3. In general, FBO is the object that creates and manages framebuffers for off-screen rendering. In our context, a framebuffer can be viewed as a pixel matrix. Given a specified resolution such as $2560px \times 1440px$, the corresponding framebuffer contains 2560×1440 pixels, each of which is represented by four 32-bit values *w.r.t.* 4 color channels (red, green, blue and alpha). In this work, we would like to use the 128-bit storage for each pixel to record important information. Unlike framebuffer, which stores data for each pixel, SSBO could be used to store data with a large size (up to the available GPU memory). In this work, the query result is temporarily stored in the GPU memory and finally returned to the main memory. SSBO is pretty applicable to our needs due to its large capacity.

For ease of following discussion, we list the key notations and the corresponding descriptions in Table 1.

Table 1: Notations and descriptions.

Notation	*Description*
D	the set of points
P	the set of polygons
o	a point in D
p	a polygon in P
m	the number of polygons in P
n	the number of points in D
v	the average number of vertices for each polygon
A	the average area of MBR for each polygon
S	the total side number of polygons in P
ϵ	the edge length of a pixel
F_{pg}	FBO for rendering polygons
$F_{pg}(x, y)$	the pixel whose coordinate is (x, y) in F_{pg}
B	batch size
R	the result set

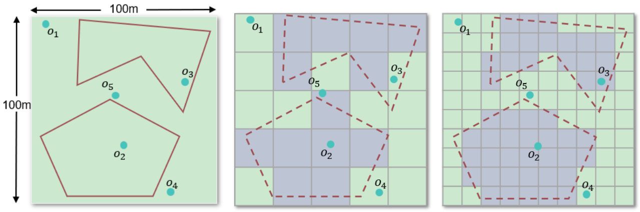
3 RasterPIP-b

In this section, we present a GPU-native solution named RasterPIP-b towards the bPIP problem.

3.1 Details of RasterPIP-b

As defined in Def. 2.1, bPIP query returns a set of points that are inside at least one polygon. Traditional CPU-based solutions first filter points that are definitely not within the polygons with the help of spatial indices, and then perform PIP tests on the remaining points. Although filtering is not time-consuming, the PIP test is still an expensive operation, especially when there are numerous polygons. The excellent parallel processing capabilities of GPU offer an opportunity to address bPIP queries. Considering the native rendering operations of GPU introduced in last section, it is possible for us to turn to GPU and address the problem by mapping the points and polygons to the same canvas. As a result, the PIP test can be transformed into a test of intersection between points and polygons on that canvas. In a GPU environment, the canvas is, in fact, a matrix of pixels. By checking whether a point and a polygon cover the same pixel, one can determine whether the point is inside that polygon. This paradigm was used to effectively solve spatial aggregation queries in [48]. However, as we introduced in Section 1, it cannot directly answer the bPIP query. Unlike spatial aggregation queries, bPIP queries need to return the point IDs. Since points can be numerous and their IDs can be large, the key challenge is how to effectively distinguish IDs on the same pixel with limited storage space. Fortunately, compared to point IDs, under the bPIP problem, we do not need to distinguish the polygons from each other, thus lowering the challenge a bit. Moreover, during a canvas test, there is a rendering order for different spatial objects. Only spatial objects rendered first need to store information in the pixel, so that later rendered objects can determine whether they cover the same pixel based on this information. Inspired by these observations, we propose RasterPIP-b, which utilizes GPU-native transformation and rasterization operations to solve the bPIP problem.

Before diving into the details of our method, let us first introduce the concept of *pixel edge length*, denoted as ϵ. Since each pixel corresponds to a rectangle in the real world, we use ϵ to represent the maximum length of that rectangle. On the other hand, ϵ also indicates the resolution of the projected screen space. The smaller the ϵ, the higher the required resolution.



(a) Spatial objects (b) ϵ = 20 meters (c) ϵ = 10 meters

Figure 4: Illustration of the effects of ϵ on Rasterization.

Example 3.1. Consider the real-world spatial objects in Figure 4(a), which contains two polygons and five points in a $100m \times 100m$ rectangular region. We illustrate the effects of ϵ in Figure 4(b) and Figure 4(c), where ϵ is set to 20 meters and 10 meters, respectively, while representing the same region in Figure 4(a). Obviously, a 5×5 pixel matrix is enough for the larger ϵ, while 10×10 for the smaller one. As a result, the resolution of the canvas is $5px \times 5px$ and $10px \times 10px$, respectively. Higher resolution results in more accurate rendering results. Therefore, we can adjust the specific ϵ according to the practical needs.

Intuitively, given a specific ϵ, our method first draws all polygons in P on a canvas by transforming each polygon from the original space to the screen space. Since we do not require the IDs of the polygons, we only need to record whether a pixel is covered by polygons, which is done by the flag stored in the color channel. Subsequently, we draw each point $o \in D$ on the same canvas. Recognizing the pixel hit by o, we can identify whether or not

```
Algorithm 1: RasterPIP-b
   Input: the set of points and polygons, $D$ and $P$; the edge
          length $\epsilon$ of a pixel
   Output: $R$
 1 // Procedure of DrawPolygons$^b$
 2 set the resolution of the FBO $F_{pg}$ based on $\epsilon$;
 3 initialize the FBO $F_{pg}$ as 0;
 4 for each $p \in P$ do
 5     triangulate $p$ to obtain a set $T$ of triangles;
 6     for each $t \in T$ do
 7         convert the vertices of $t$ to the screen space;
 8         rasterize $t$ into a set $F$ of fragments;
 9         for each $(x, y) \in F$ do
10             $F_{pg}(x, y) = 1$;
11         end
12     end
13 end
14 // Procedure of DrawPoints$^b$
15 $R = \emptyset$;
16 for each $o \in D$ do
17     transform $o$ to get its position $(x, y)$ in the screen space;
18     if $F_{pg}(x, y) > 0$ then
19         $R = R \cup \{o\}$;
20     end
21 end
22 return $R$
```

the same pixel has been covered by any polygon, by checking its corresponding flag. If so, o is identified as a valid result, which is temporarily stored in SSBO, and finally returned to the main memory.

The pseudocode of our method is shown in Algorithm 1. RasterPIP-b consists of two key steps, *i.e.,* DrawPolgons^b and DrawPoints^b, which correspond to polygon drawing and point drawing (introduced in Section 2.2), respectively. As shown in Line 2, we allocate F_{pg} to represent the result of drawing polygons and set the resolution of F_{pg} based on the pixel edge length ϵ. Then we initialize as 0 for each pixel in Line 3. Let (x, y) be the coordinates of a pixel. We use $F_{pg}(x, y)$ to denote whether the pixel is covered by any polygon. Then, we draw the polygons one by one in Line 4 to 12. For each polygon p, we first decompose it into a set of triangles (Line 5) and then transform each triangle into a set F of fragments in the screen space (Line 7). For each fragment (x, y) covered, we simply set the corresponding value in $F_{pg}(x, y)$ to 1 (Line 9-11). Next, let us focus on DrawPoints^b. We use R to represent the set of points that fall in at least one polygon. Initially, R is empty (Line 15). Then, we conduct the bPIP query in Line 16-21. For each point o, we first transform it with ϵ and obtain its coordinate (x, y) on the screen space (Line 17). With (x, y), we can easily access $F_{pg}(x, y)$, which indicates whether or not this pixel is covered by any polygon. If $F_{pg}(x, y) > 0$, we add o to R (Line 18-20). Note that R is stored in an SSBO, which could be used to store an array with a large size up to that of the memory (DRAM correlates with CPU).

Example 3.2. We present an example illustrating the pipeline of Algorithm 1 in Figure 5. bPIP query needs to return the IDs of all points located within the two polygons. Assuming that ϵ is specified to 20 meters, the required resolution is $5px \times 5px$. We first draw the polygons. For each pixel covered by the polygons, write "1" to its color channel, *e.g.,* $F_{pg}(5, 5)$. Next, each point is transformed to the screen space and the corresponding coordinates are obtained. Based on the coordinates, we can check the value of the corresponding pixel color channel. For example, for o_1, $F_{pg}(1, 5) = 0$, indicating that it is not covered by the polygons; but for o_2, the color channel value is 1. Therefore, its ID is recorded in SSBO. Through the above process, o_2 and o_5 are ultimately identified.

Interestingly, although the solution is efficient, there may be false positives and false negatives in the set of results R. In example 3.2, o_3 is a false negative while o_5 is a false positive, which can be easily observed in Figure 4(b). This is caused by the rasterization procedure. In general, a higher resolution reduces the likelihood of such errors, leading to more accurate results. As shown in Figure 4(c), compared to $\epsilon = 20$, RasterPIP-b with $\epsilon = 10$ correctly classifies both o_3 and o_5 and thus avoids returning false positives and negatives.

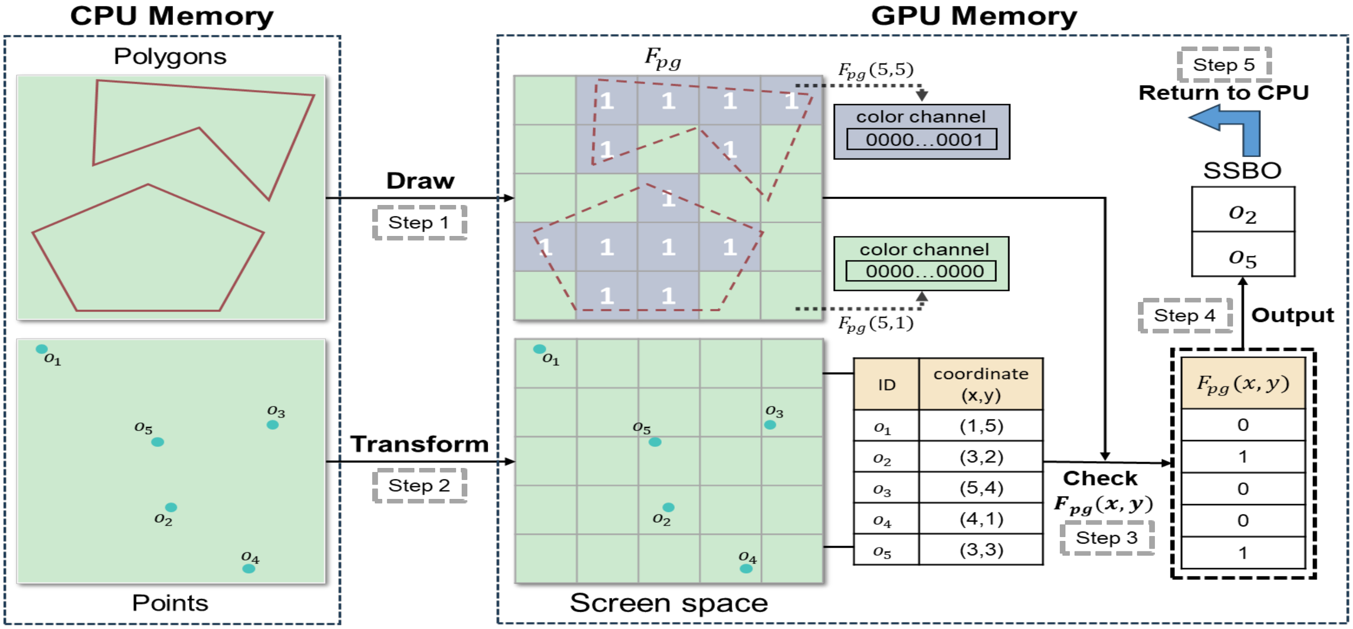
However, a higher resolution requires more pixels to draw the polygons. In fact, the number of pixels covered by both polygons and points is key to the cost of our method, since our solution has to deal with each of these pixels. The time complexity of RasterPIP-b and its corresponding proof are provided in Theorem 3.3. Therefore, ϵ is the key parameter to balance effectiveness and efficiency within our solutions, which should be carefully selected.

Theorem 3.3. *Given a set of polygons P and a set of points D, let m and n be the number of polygons and the number of points, respectively. Each polygon in P has an average of v vertices and the average area A of its minimum bounding rectangle (MBR). Suppose ϵ refers to the edge length of a pixel; then the upper bound of the time complexity for RasterPIP-b is $O(m \cdot (v \log v + A/\epsilon^2) + n)$.*

Proof. As a key step, DrawPolygons^b first decompose polygons into triangles and then render them by GPU. The former requires $O(m \cdot v \log v)$, while the latter is determined by the total of pixels covered by all polygons. Since the area of a polygon is never greater than that of its MBR, A can be considered as an upper bound for the area of the polygon. Thus, the total number of pixels covered by all polygons could be estimated as $A \cdot m/\epsilon^2$. As a result, the cost of DrawPolygons^b is $O(m \cdot (v \log v + A/\epsilon^2))$. On the other hand, the cost of DrawPoints^b is $O(n)$ since it first renders each point and then checks its corresponding position in the FBO F_{pg}. In summary, the upper bound of the time complexity for RasterPIP-b is $O(m \cdot (v \log v + A/\epsilon^2) + n)$. □

3.2 Partitioned RasterPIP-b

In some scenarios, users may have a high demand for the accuracy of the results. As mentioned in the previous subsection, we can increase the resolution by reducing the value of ϵ, thus improving the accuracy. However, when ϵ becomes sufficiently small, rendering a large area requires a considerable number of pixels, which may exceed the maximum resolution supported by the GPU. For instance, suppose that we are performing bPIP over the rectangular



ID	coordinate (x,y)
o_1	(1,5)
o_2	(3,2)
o_3	(5,4)
o_4	(4,1)
o_5	(3,3)

$F_{pg}(x,y)$
0
1
0
0
1

Figure 5: Illustrating the pipeline of RasterPIP-b.

region in Figure 4(a) and the maximum resolution supported by the GPU is $50px \times 50px$. When ϵ is set to 1 meter, the number of pixels required to render can be as large as 100×100. In this case, the canvas cannot cover all points and polygons. To address this issue, we partition the points and polygons into several parts, ensuring that the resolution required to render each part does not exceed the maximum supported by the GPU. Specifically, based on the actual resolution required and the maximum resolution supported by the GPU, we can obtain at least how many canvases are needed to fully render the entire region. Based on the number of canvases, we evenly split the region and render them separately. This approach allows each part to be successfully drawn on a canvas. Therefore, RasterPIP-b can be performed on each canvas, respectively. Finally, we merge the results of all partitions.

However, the results obtained from the partition strategy mentioned above sometimes contain duplicate points. Specifically, points located precisely on the partition boundaries will be included in adjacent parts at the same time, leading to duplicate results. A naive way to solve this problem is to remove duplicates by traversing the result set. However, as the number of points increases, it will produce additional overhead. Therefore, we need to avoid redundancy in the result set as early as possible.

Our solution is inspired by the bitmap index scan, which is a database query optimization technique in PostgreSQL [35]. For a query, the database creates a bitmap for each query condition, where each bit represents a row of data, and then combines these bitmaps through logical operations to identify the rows that satisfy all conditions. This method is particularly effective for large datasets and low-selectivity queries, as it reduces random access and improves query efficiency. Therefore, we adopt a similar strategy to employ a bitmap for all points at run-time to indicate whether a point has already been identified as inside the polygons. If it has, there is no need for further evaluation, thereby avoiding duplicates in the result set.

An example of the whole procedure is illustrated in Figure 6. For ease of representation, suppose that the maximum resolution supported by the GPU is $5px \times 5px$ and ϵ is 10 meters. As a result, for the spatial objects in Figure 4(a), the required resolution is $10px \times 10px$. Therefore, we need to divide the original area into four parts and render each part with a canvas resolution of $5px{\times}5px$. Then, RasterPIP-b is performed on each canvas. Note that since o_2 is located at the partition boundary, it is included in both canvases 3 and 4. In canvas 3, when o_2 is identified as being inside the polygons, the corresponding bit in the bitmap, *i.e.,* the second bit, is first checked. If it is 0, indicating that the point is not yet in the result set, then o_2 is inserted into the SSBO, and the second bit in the bitmap is updated to 1. In canvas 4, o_2 is again identified, but the second bit in the bitmap is 1, indicating that o_2 has already been processed, so there is no need to insert it into the SSBO again. For other points, the processing procedure is similar.

4 RasterPIP-i

In this section, we present a novel solution towards iPIP. Intuitively, the iPIP problem can be decomposed into a series of bPIP problems and solved by invoking RasterPIP-b multiple times. However, this approach is inefficient, so we propose RasterPIP-i tailored to the iPIP problem.

4.1 Details of RasterPIP-i

The iPIP query in Def. 2.2 not only finds out the points that fall inside at least one polygon, but also specifies which polygon(s) each point falls inside. Both the IDs of points and polygons are required,

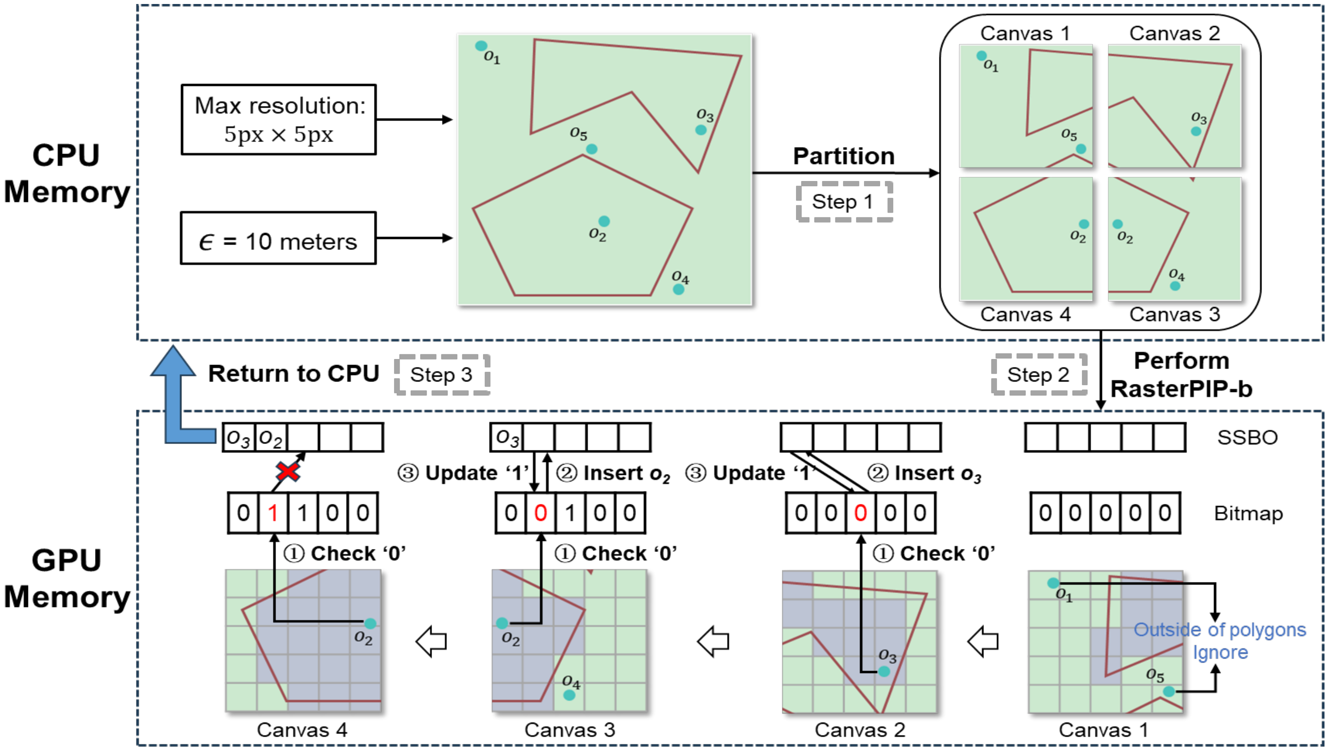


Figure 6: Illustrating the pipeline of partitioned RasterPIP-b.

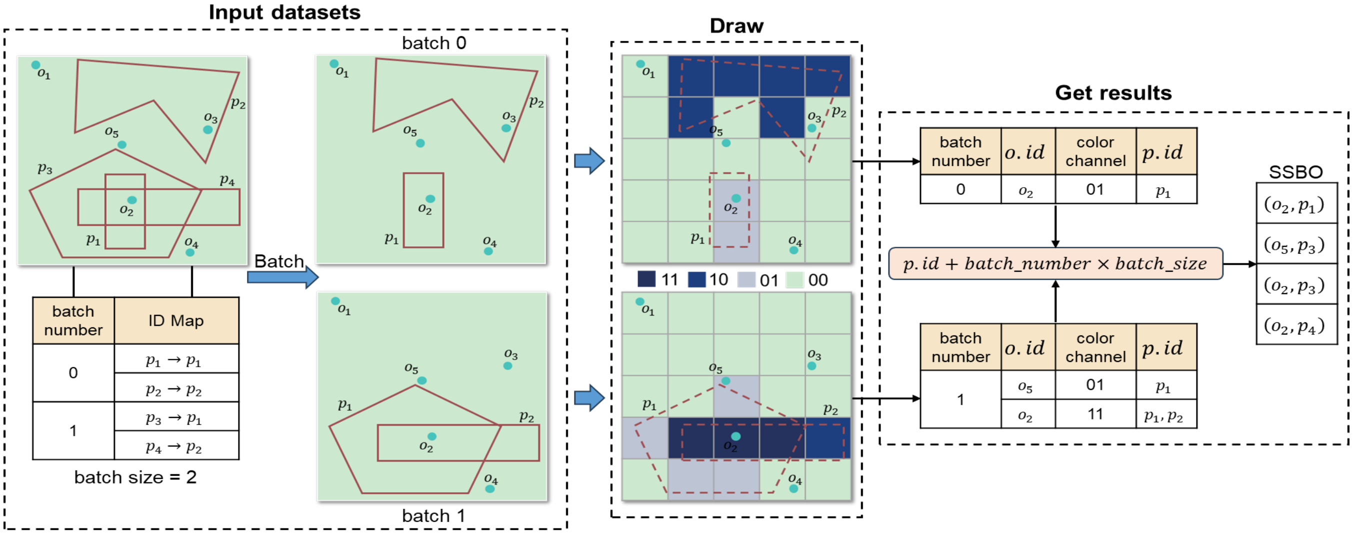
and thus at least one type of ID should be recorded in the color channel. This is blocked by the limited representation capability in a pixel's color channel. We cannot address it by adding an array for each pixel in order to record which polygon hits this pixel, which is unrealistic. Thus, we propose RasterPIP-i, which adopts a batch processing approach to overcome the above challenge.

Like RasterPIP-b, RasterPIP-i needs to draw the polygons and the points, but with a slight difference, denoted as $\texttt{DrawPolygons}^i$ and $\texttt{DrawPoints}^i$, respectively. In $\texttt{DrawPolygons}^i$, we need to draw each polygon in P to get the set of pixels covered by it. In addition, for a pixel with coordinate (x, y), we should store the IDs of the polygons that cover it. First, let us assume $|P| \leq 128$ (we shall remove this assumption soon) and that the polygon IDs are in the range $[0, |P|)$. We use the FBO F_{pg} to store the corresponding polygon IDs for each pixel, which corresponds to 4 32-bit color channels. For each pixel, we could use this 128-bit bitmap to represent any subset of P covering it. If a polygon with ID i covers a pixel, we set the i-th bit of its 128-bit bitmap as 1. During $\texttt{DrawPolygons}^i$, we update F_{pg} for each pixel. Like $\texttt{DrawPoints}^b$, $\texttt{DrawPoints}^i$ first transforms each point $o \in D$ into the screen space and gets the pixel (x, y) hit by o. Then, we can recognize the subset of polygons that cover the pixel by decoding the 128-bit bitmap *w.r.t.* $F_{pg}(x, y)$.

Now, let us consider $|P| > 128$. Since the 128-bit bitmap for each pixel can represent at most 128 polygons, we divide P into batches, where each batch contains at most 128 polygons. For a batch of 128 polygons, we encode each polygon ID id in a new range between 0 and 127 by $id\%128$, so that the 128-bit bitmap could be used to solve iPIP as discussed above. Obviously, we can easily decode an ID in the new range from its original ID. Then, we obtain the point-polygon pair where the point falls inside the polygon by first drawing the polygons in the batch and then drawing each point. The final results are merged once all batches have been processed in the same way.

Example 4.1. An example workflow of RasterPIP-i is illustrated in Figure 7. Assuming that the color channel contains only two bits, in which case a pixel can distinguish the IDs of at most two polygons. Therefore, to answer the iPIP query, we need to batch polygons with a size of 2. In each batch, the IDs of the polygons are first mapped as 1 to 2. Then, polygons and points are rendered in sequence. When rendering the polygons, their IDs are encoded into the color channel of the corresponding pixels. For instance, for the pixel $F_{pg}(3, 2)$ of batch 0, which is covered by polygon p_1, the first bit of its color channel is set to 1. We use different colors to represent different pixel values. When rendering each point, we retrieve the value of its pixel color channel and check which bit is set to 1. Taking o_2 of batch 1 as an example, the first and second bits are set to 1, indicating that it is inside both polygons p_1 and p_2. The polygon IDs obtained at this stage are processed through the ID map. It is easy to reconstruct the real IDs of the polygons in the current batch based on the batch number. Note that o_2 is located within three polygons, whose IDs will not be effectively distinguished without the batch strategy.

To be formal, the pseudocode of RasterPIP-i is shown in Algorithm 2. As shown in Lines 2-20, we iteratively deal with polygons



batch number	ID Map
0	$p_1 \rightarrow p_1$
	$p_2 \rightarrow p_2$
1	$p_3 \rightarrow p_1$
	$p_4 \rightarrow p_2$

batch size = 2

batch number	$o.id$	color channel	$p.id$
0	o_2	01	p_1

$p.id + batch\_number \times batch\_size$

batch number	$o.id$	color channel	$p.id$
1	o_5	01	p_1
	o_2	11	p_1, p_2

SSBO
(o_2, p_1)
(o_5, p_3)
(o_2, p_3)
(o_2, p_4)

Figure 7: Illustrating the pipeline of RasterPIP-i.

```
Algorithm 2: RasterPIP-i
   Input: the set $D$ of points and the set $P$ of polygons
   Output: $S$
1  $B = \lceil |P|/128 \rceil$;
2  for $j = 0$ to $B - 1$ do
3      // Procedure of DrawPolygons$^i$
4      initialize $F_{pg}$ as 0;
5      for each $p \in P_j$ do
6          $newPolyID$ = EncodeID($p.id$);
7          obtain the set $FS$ of fragments hit by $p$ after
             triangulation and rasterization;
8          for each $(x, y) \in FS$ do
9              $F_{pg}(x, y) = F_{pg}(x, y)$ OR $(1 << newPolyID)$;
10         end
11     end
12     // Procedure of DrawPoints$^i$
13     $S = \emptyset$;
14     for each $o \in D$ do
15         transform $o$ to get its pixel position $(x, y)$;
16         $originalPolyids$ = DecodeID($F_{pg}(x, y), j$);
17         for each $p.id \in originalPolyids$ do
18             $S = S \cup (o.id, p.id)$;
19         end
20     end
21 end
22 return $S$
```

```
Algorithm 3: EncodeID
Function EncodeID($original\_id$):
    newID = original_id mod 128;
    return newID;
```

```
Algorithm 4: DecodeID
Function DecodeID($color\_value, batch\_num$):
    original_ids = $\emptyset$;
    bit_position = 0;
    while $color\_value > 0$ do
        if $color\_value$ AND $1$ then
            original_id = batch_num * 128 + bit_position;
            original_ids = original_ids $\cup$ original_id;
        end
        color_value = color_value >> 1;
        bit_position = bit_position + 1;
    end
    return original_ids;
```

in each batch, which contains DrawPolygons^i and DrawPoints^i. The key difference between DrawPolygons^b and DrawPolygons^i is the information stored in F_{pg}. In Line 9, DrawPolygons^i records the ID of each polygon that covers the pixel, where the polygon ID is mapped from the original one to the range between 0 and 127 using the EncodeID function shown in Algorithm 3. On the other hand, as shown in Lines 16-19, the key difference between DrawPoints^i and DrawPoints^b is that the former decodes polygon IDs and returns point-polygon pairs instead of only point IDs, which is shown in detial in Algorithm 4.

4.2 Comparing RasterPIP-i with GPU-Reg

Both RasterPIP-i and GPU-Reg employ a batching strategy, but with different geometric objects, resulting in different total numbers of rendering pixels, which is the key difference between them. For ease of discussion, let us denote the number of polygons and points as m, n, respectively, A as the average area of the MBR for each polygon, B as the batch size, and N as the number of pixels rendered. We can easily conclude that all polygons involve $m \cdot A/\epsilon^2$ pixels at most. Considering that rendering each pixel has the same cost, we compare both methods in the total number of pixels rendered.

RasterPIP-i partitions the set of polygons into batches, resulting in repeatedly rendering of all points in each batch, while each individual polygon is rendered only once. Therefore, the number of pixels rendered in RasterPIP-i can be calculated as

$$N_1 = A \cdot m/\epsilon^2 + \lceil m/B \rceil \cdot n.$$

On the contrary, GPU-Reg partitions points, causing repeated rendering of all polygons in each batch, while all points are rendered once. Therefore, the number of pixels rendered in GPU-Reg is

$$N_2 = n + \lceil n/B \rceil \cdot A \cdot m/\epsilon^2.$$

As a result,

$$N_1 - N_2 = \left(\lceil \frac{m}{B} \rceil - 1\right) \cdot n - \left(\lceil \frac{n}{B} \rceil - 1\right) \cdot \frac{A \cdot m}{\epsilon^2}.$$

Obviously, in practice, $m \ll n$. Therefore, GPU-Reg requires much more batches than RasterPIP-i. In addition, it is reasonable to assume that the number of pixels covered by all polygons is approximately equal to or even more than n, especially considering that the points have been filtered by an efficient spatial index. Therefore, in practice, $N_1 \ll N_2$, which demonstrates the advantages of our method over GPU-Reg in the number of pixels rendered. Hence, our method, theoretically, has a significant advantage over GPU-Reg in efficiency.

5 EXPERIMENTS

In this section, we conduct extensive experiments to demonstrate the advantages of our methods over a series of baselines on two real-world datasets.

5.1 Experimental settings

Datasets. We conduct our experiments on two real-word datasets, called *OSM* and *Taxi* . *OSM* is the Open Street Map dataset provided by [12]. We randomly sampled 10 million points from *OSM* within the United States, each point containing its ID and geographical coordinates. *Taxi*[2] records the trip data of yellow taxis in New York City, including pick-up and drop-off locations, travel distances, etc. We select the trip data for one month and extract the pickup locations to form the set of points D. There are 13,294,789 points in *Taxi.* The two datasets mentioned above are both skewed. Especially, for *Taxi*, the data points are concentrated in Manhattan, New York City.

In terms of polygons, we utilize both real-world regions from *Taxi Zones* and *synthetic ones*. *Taxi Zones* represent the division of New York City taxi regions; *Taxi Zones* consist of 263 polygons, most of which are concave in shape, and each polygon represents a taxi service zone. *Taxi Zones* are publicly available[3]. *Synthetic polygons* are provided by RasterJoin [48], with the aim of generating a substantial number of polygons with properties similar to the real ones, and contains many simple and complex polygons with various sides number and shapes (concave and convex).

Computing Environments. All experiments are carried out on a server with two Intel Xeon E5-2680 CPUs, 256 GB of RAM and a Nvidia 2080TI GPU with 11 GB of memory. The solutions are implemented using C++ and OpenGL.

Performance Measures. Each method is evaluated from both the efficiency and the precision aspect. The former is measured by the elapsed time, denoted as *cost*, and the latter is evaluated by the F_1 score [45] defined as follows.

$$F_1 = \frac{2 \times \frac{TP}{|R|} \times \frac{TP}{|GT|}}{\frac{TP}{|R|} + \frac{TP}{|GT|}}$$

Here, TP denotes the number of points our algorithm accurately identifies inside the polygons. $|R|$ is the size of the algorithm's result set and $|GT|$ represents the true number of points within the polygons.

Compared Baselines. We compare our methods with both CPU-based methods, including **RayCasting** and **PostGIS**, as well as the latest GPU-based solution, *i.e.,* **GPU-Reg** [27].

RayCasting: This popular CPU-based solution is proposed in [19] and is paralleled with *OpenMP*[4], in order to exploit the computing power of multi-core CPUs.

PostGIS: As the leading industrial solution for spatial data management, **PostGIS** provides ST_Within function to identify points inside polygons. For a fair comparison, we use R-tree to filter irrelevant points and set the maximum parallel workers of PostGIS as the number of CPU threads.

GPU-Reg: It is the latest GPU-based solution proposed in [27].

RasterPIP: We integrate RasterPIP into PostGIS to replace its default module and report the performance accordingly.

For each method, we set their parameters for the best performance. For both **RasterPIP-b** and **RasterPIP-i**, we limit GPU memory to 4GB, set the maximum FBO size to $8192px \times 8192px$, and vary ϵ between 1 and 10 meters, by default. For **RasterPIP-i**, we set the batch size to 128 by default. For **GPU-Reg**, we set its ϵ to 10 meters. Note that for both RasterPIP and GPU-Reg, we use a simple MBR to filter point data at coarse granularity. In particular, all experimental results are averaged over ten runs.

5.2 Experimental results for RasterPIP-b

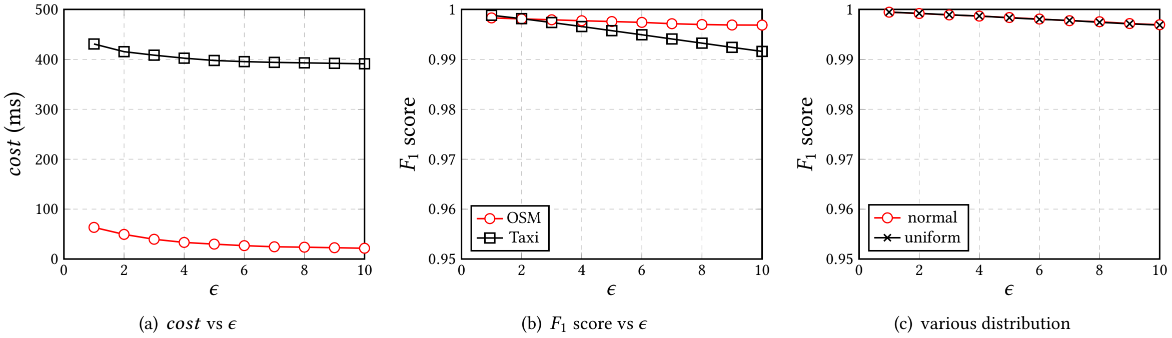
In this part, we present the performance of RasterPIP-b, and investigate the factors that impact its performance, including the pixel edge length ϵ, the number n of points, the number m of polygons, and the area A of polygons.

Effects of Pixel Edge Length ϵ. As an approximate algorithm, RasterPIP-b does not return all correct points, mainly caused by ϵ, as discussed in Section 3. We carried out experiments to investigate the effects of ϵ on RasterPIP-b. We randomly select 5 to 50 polygons from *Taxi Zones* to form the set P, and vary ϵ between 1 and 10 meters. Figure 8(a) and Figure 8(b) illustrate our experimental results. In general, both the *cost* and F_1 scores decrease as ϵ increases. Note that a larger pixel edge length leads to a lower resolution, where fewer pixels are enough to represent the same polygon during the

[2] https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[3] https://data.cityofnewyork.us/Transportation/NYC-Taxi-Zones/d3c5-ddgc

[4] https://www.openmp.org/



(a) *cost* vs ϵ (b) F_1 score vs ϵ (c) various distribution

Figure 8: The effects of the pixel edge length ϵ on RasterPIP-b.

rasterization, and thus less cost is required to process them. Even returning approximate results, the F_1 scores of RasterPIP-b are always larger than **0.99**, which means that our method is pretty accurate with various pixel edge lengths. To be more convincing, we further test the F_1 score by changing the data distributions. We generated two datasets, each containing 10,000,000 points within the scope of New York City, with normal and uniform distribution, respectively. The corresponding experimental results in Figure 8(c) show that the F_1 scores for the two distributions are almost indistinguishable, and even when ϵ is set to 10, the F_1 score still remains above 0.99. In the rest of this part, we present RasterPIP-b with two pixel edge lengths, *i.e.,* $\epsilon = 1$ and $\epsilon = 10$, representing a small pixel edge length and a large one respectively.

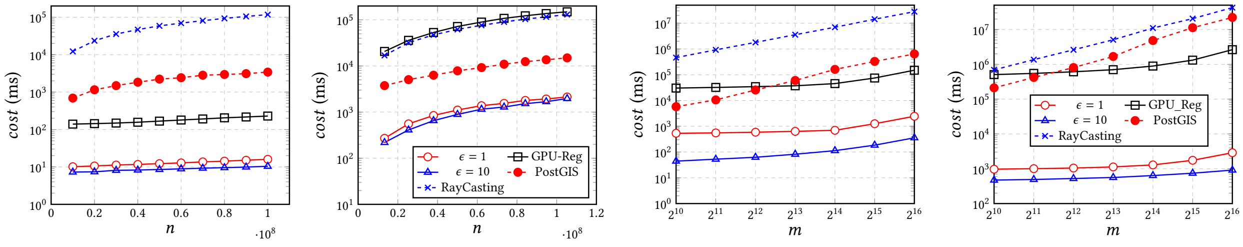
Effects of the Number n of Points. To show the scalability of RasterPIP-b *w.r.t.* n, we randomly select one-year data from *Taxi* and vary n from 10 million to 100 million. For *OSM*, we also randomly sample (without replacement) between 10 million and 100 million points within the United States. Furthermore, we randomly select five polygons from *Taxi Zones* to form the set P. The results are reported in Figure 9(a-b). RasterPIP-b consistently outperforms other approaches by at least one order of magnitude. The superiority of RasterPIP-b in *cost* is mainly due to the fact that the average cost of rendering a single pixel by the GPU-native operations is significantly lower than that of a PIP test using CPU-based methods. However, GPU-Reg needs to render the same query polygons for each batch of points, which contains at most 128 points. As a result, it has to render the same polygons in P more times as n increases. On the contrary, our method only renders the polygons *once* during the whole process. Note that GPU-Reg has good performance on *OSM*, but pretty bad on *Taxi*. This is because the filtering efficiency of the MBR on *OSM* is much better than that on *Taxi*, thereby reducing the number of times polygons are rendered repeatedly. By contrast, RasterPIP-b only renders the polygons once.

Effects of the Number m of Polygons. We utilize synthetic polygons to form P with various cardinality between 1024 and 65536. Figure 9(c-d) shows *cost* by varying m on both *OSM* and *Taxi*. We can see that RasterPIP-b obviously outperforms other methods with both $\epsilon = 1$ and $\epsilon = 10$. The cost of RasterPIP-b is closely related to the number of rendering pixels, as discussed in Section 3.1. Given the pixel edge length ϵ, the larger area P covers, the more pixels are rendered and the more time RasterPIP-b spends. However, as shown in Figure 9(c-d), the more polygons there are, the more expensive PIP tests must be performed, resulting in a rapid increase in the cost of CPU-based methods. For GPU-Reg, due to the repeated rendering of polygons, as the number of polygons increases, the rate of cost growth will be higher than that of RasterPIP-b. Overall, RasterPIP-b outperforms other methods and can achieve a speedup between 1 and 3 orders of magnitude in all cases.

Effects of the Area A of Polygons. As discussed in Section 3.1, given ϵ, the average area A of the polygons directly affects the number of pixels rendered by RasterPIP-b and further impacts *cost*. To justify that, we select a rectangle to form P, while varying its area between 10 and 10000 km^2 on both *OSM* and *Taxi*. The results are shown in Figure 10. The pure gray portion of each bar represents the cost of polygon rendering in GPU-based methods. RasterPIP-b and GPU-Reg cost more in polygon rendering as the growth of its area. However, RasterPIP-b only needs to render the polygon once, thus incurring a lower cost. Furthermore, as the area increases, the increase in cost is not significant. In addition, RasterPIP-b has a low rendering cost due to the large edge length of pixels $\epsilon = 10$, since only a few pixels are processed.

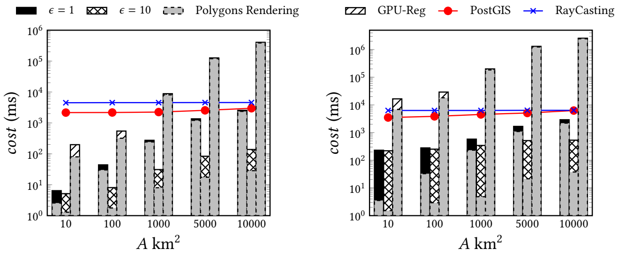
In terms of RayCasting, it conducts point-in-polygon tests for each point, whose computational cost is determined by the number of sides of the polygon. Thus its performance is not sensitive to the polygon's area. PostGIS employs an index to obtain candidates and then performs point-in-polygon tests for each candidate. Since the polygon's area has a positive impact on the number of candidates, PostGIS spends more time for a larger polygon. From Figure 10, it can be observed that the cost required by RasterPIP-b remains consistently the lowest.

It is worth noting that on *OSM*, as the area increases, the cost of PostGIS and RasterPIP-b becomes increasingly closer when ϵ is 1. This is due to the broader coverage of the *OSM* dataset. As the area grows, the increase in the number of candidate points is not significant, leading to only a modest increase in the number of PIP tests for PostGIS. This reveals a potential issue with RasterPIP-b: it might not perform well for a polygon that has larger area but contains few points. We simulate an extreme scenario where the rectangular area is 10000 km^2 but covers only 20 points. Specifically, when ϵ is 1, the cost of RasterPIP-b is 4038 ms, while PostGIS takes



(a) OSM (b) Taxi (c) OSM (d) Taxi

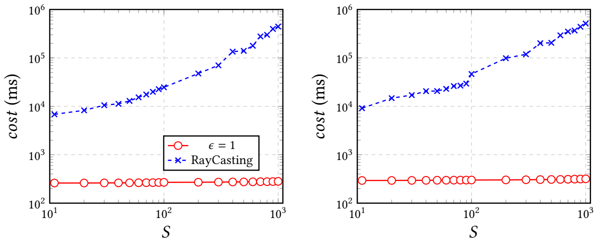
Figure 9: The effects of the number n of points (a, b) and the effects of the number m of polygons (c, d) on RasterPIP-b.



(a) OSM (b) Taxi

Figure 10: The effects of the area A on RasterPIP-b.

only about 1.5 ms. Although such extreme situations are rare in the real world, it still indicates that RasterPIP-b may not be superior all the time. Therefore, there might be a need to design a cost model to decide whether to use RasterPIP-b based on specific circumstances. Fortunately, when ϵ is 10, the cost of RasterPIP-b is acceptable, *i.e.,* 22 ms, allowing for quick query responses when precise results are not required.

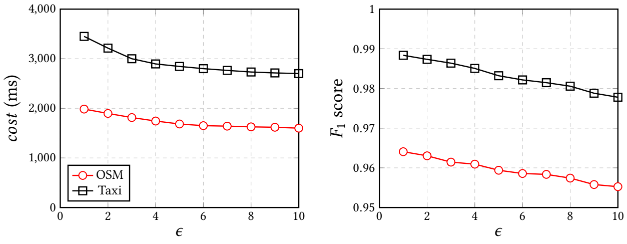


(a) OSM (b) Taxi

Figure 11: The effects of the side number S of polygons on RasterPIP-b.

Effects of the Side number S of Polygons. In order to investigate the sensitivity of RasterPIP-b to the side number of polygons, we construct a series of polygon collections where the side number in each collection is fixed and ranges from 10 to 1000. Each polygon collection contains an equal number of convex and concave polygons with a fixed area. The experimental results are shown in Figure 11. Because the difference between PostGIS and RayCasting lies only in the fitler for the candidate points, we use RayCasting as a comparison here. As the S increases, the cost of RayCasting, which is famously sensitive to the side number of polygons, also grows rapidly. In contrast, for RasterPIP-b, the cost remains within a relatively small range on both *OSM* and *Taxi*, even with ϵ=1. More specifically, Table 2 presents the maximum and minimum costs of RayCasting and RasterPIP-b in the above experiment. We can observe that Δc, which represents the difference between the maximum and minimum cost, is pretty small, *i.e.,* constantly less than 25 ms for RasterPIP-b. It is mainly caused by the triangulation process. In contrast, for RayCasting, it reaches up to several hundred thousand. Therefore, we can conclude that RasterPIP-b is not sensitive to the side number of polygons.

Summary of RasterPIP-b. In general, RasterPIP-b significantly outperforms all other baselines and presents three interesting properties. First, it is able to balance efficiency and accuracy by adjusting the pixel edge length ϵ. Second, it is not sensitive to either the number n of points or the number m of polygons, which implies that it scales well. Third, it is not sensitive to the polygon's shape, area or side number, which implies that its performance is stable for various input polygons.



(a) *cost* vs ϵ (b) F_1 score vs ϵ

Figure 12: The effects of pixel edge length ϵ on RasterPIP-i.

5.3 Experimental results for RasterPIP-i

In this part, we slightly modify other methods to enable them to support iPIP and test the performance.

Effects of Pixel Edge Length ϵ. Like RasterPIP-b, as ϵ increases, both *cost* and F_1 score decrease, as shown in Figure 12. However, on both *OSM* and *Taxi*, the F_1 score of RasterPIP-i is still at least

Table 2: The Δc of RasterPIP-b and RayCasting.

Method	minCost(ms)	maxCost(ms)	Δc(ms)	dataset
RasterPIP-b	260.57	281.74	21.17	OSM
RayCasting	6829.97	443651	436821.03	OSM
RasterPIP-b	294.07	317.71	23.64	Taxi
RayCasting	9140.32	511550	502409.68	Taxi

0.95. Similar to RasterPIP-b, we test the performance of RasterPIP-i at two representative pixel edge lengths, *i.e.,* 1 and 10.

Effects of the Number n of Points. We take the same P that contains five polygons in this experiment as that in Section 5.2. The result is shown in Figure 13(a-b), RasterPIP-i outperforms other methods with 10× speedups at least. Additionally, as n increases, the cost of RasterPIP-i increases slightly, which implies that it has good scalability.

Effects of the Number m of Polygons. The result is shown in Figure 13(c-d). Obviously, RasterPIP-i outperforms all baselines in almost all cases. On *OSM*, PostGIS achieves a performance comparable to that of our method when $\epsilon = 1$, even better in certain situations. This is because for *OSM*, few points remain after index filtering, leading to a few PIP tests later. However, when $\epsilon = 10$, our method has an absolute advantage. Furthermore, on *Taxi*, our method outperforms other methods by a large margin in all cases. Notably, under the same setting, RasterPIP-i costs more than RasterPIP-b, since it has to render the points $\lceil m/128 \rceil$ times.

Effects of Batch Size B. The batch size of the processed polygons is a key parameter for the performance of RasterPIP-i. Since each pixel has 4 32-bit color channels, we vary the batch size of polygons in the set $\{32, 64, 96, 128\}$. The experimental result is shown in Figure 14. We can see that as B increases, *cost* for rendering points decreases, since a larger batch size requires fewer iterations, where all the points are rendered per iteration. By default, we set B to 128.

Effects of the Side Number S of Polygons. We carried out experiments identical to those for RasterPIP-b to demonstrate that RasterPIP-i is not sensitive to the side number in polygons. The results are shown in Figure 15. Similar to RasterPIP-b, even if S increases by two orders of magnitude, the cost of RasterPIP-i remains stable. The Δc given in Table 3 is also very small, around 30ms, which indicates that RasterPIP-i is not sensitive to the side number of polygons.

Summary of RasterPIP-i. From the above experiments, it can be seen that the results obtained by RasterPIP-i are similar to those of RasterPIP-b. This is reasonable because RasterPIP-i is essentially an extension of RasterPIP-b. In summary, RasterPIP-i exhibits excellent performance, as detailed below. First, like RasterPIP-b, RasterPIP-i achieves pretty high accuracy and offers flexibility to balance efficiency and accuracy by controlling ϵ. Second, the batch size has a positive impact on the efficiency of our method. Third, RasterPIP-i shows excellent scalability *w.r.t.* both the number of points and polygons, and is insensitive to the side number of polygons. All of those advantages make our method more competitive in solving iPIP.

6 RELATED WORKS

6.1 CPU-based solutions for PIP

The point-in-polygon problem has been discussed for decades [40]. Later, Sutherland *et al.* proposed two versions of solutions based on the computational geometry: the *Ray Casting* and the *Angle Summation* [44]. Both methods conduct a PIP test for each point to determine if it lies within a polygon. Many techniques have been proposed to improve the performance of these methods [2, 18, 19, 22].

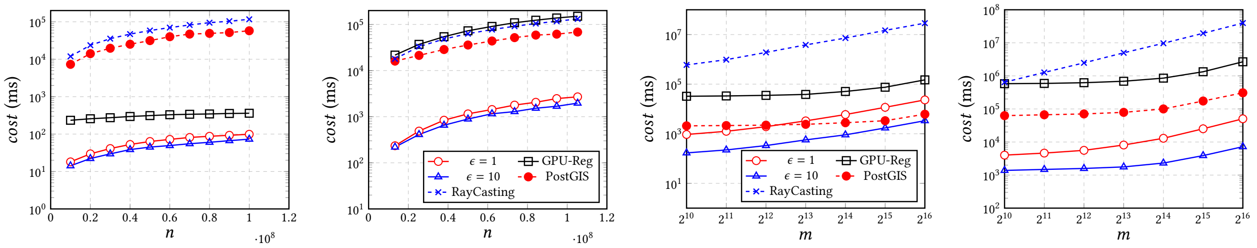
However, with the exponential growth of spatial data, traditional methods for performing a PIP test for each point could no longer meet the requirements for query response time. This leads to the development of spatial databases specifically designed to manage and query spatial data [7, 14, 30, 34, 38]. Solutions to the PIP problem have been incorporated into the spatial query processing of these databases.

To reduce the number of required PIP tests, current mainstream databases typically include two steps: filtering and refinement. In the filtering step, spatial databases use spatial indexes to manage spatial objects effectively [4, 17, 21]. The most common spatial index structures are R-trees and their variants [3, 8, 21, 23, 41], which are spatial extensions of B-trees and are currently implemented in PostGIS [34]. By combining R-trees with the Minimum Bounding Rectangle (MBR) of the query polygon, points that are definitely not inside that query polygon can be quickly filtered out. In the refinement stage, only the candidate points generated during the filtering stage need to be subjected to PIP tests, thereby reducing the overhead.

Other CPU-based methods, such as IDEAL [46], utilize the characteristics of the raster model to guide spatial queries over the vector model. Rasterization Filter [52] proposes a raster approximation for the processing of spatial joins. Additionally, there are large-scale cluster-based solutions that manage spatial data using powerful computer clusters to achieve better performance [13, 32]. Our methods are also applicable in the context of bit spatial data due to the powerful parallel processing capabilities of GPU.

6.2 GPU-based solutions for PIP

Many studies[5, 42, 48] have shown that, thanks to existing technologies, the filtering step generally does not consume much time, but refinement remains an expensive operation. Due to the widespread use of GPU and its powerful computing capabilities, numerous algorithms have emerged that utilize GPU to accelerate PIP queries. Harish et al. [11] proposed a novel index named STIG (Spatio-Temporal Indexing using GPU), which was designed to support complex spatio-temporal queries on large historical data



(a) OSM (b) Taxi (c) OSM (d) Taxi

Figure 13: The effects of the number n of points (a, b) and the effects of the number m of polygons (c, d) on RasterPIP-i.

Table 3: The Δc of RasterPIP-i and RayCasting.

Method	minCost(ms)	maxCost(ms)	Δc(ms)	dataset
RasterPIP-i	285.59	318.92	33.33	OSM
RayCasting	6954.47	403150	396195.53	OSM
RasterPIP-i	314.94	345.5	30.56	Taxi
RayCasting	7178.43	455937	448758.57	Taxi

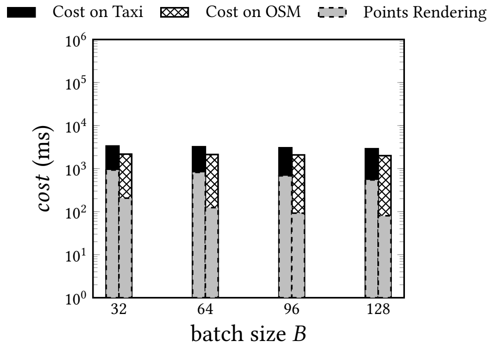
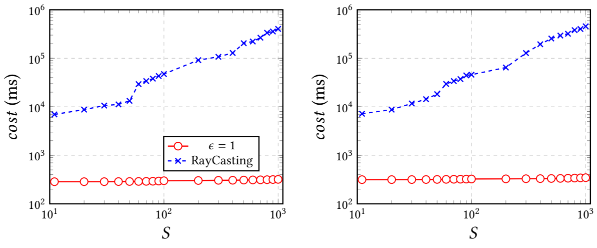


Figure 14: The effects of batch size B on RasterPIP-i. For each batch size, the left bar and right bar represent results on *Taxi* and *OSM* respectively.



(a) OSM (b) Taxi

Figure 15: The effects of the side number S of polygons on RasterPIP-i.

at interactive speeds. STIG aimed to utilize GPU for rapid spatial query operations, enabling a high degree of parallelization for numerous PIP tests. Zhang [51] et al., following the general spatial join strategy of spatial databases [24], utilized GPU to design an indexing method based on grid files [37] in the filtering phase and a parallelized PIP testing scheme in the refinement phase. Besides, [50] proposed a pair of methods for constructing quadtrees based on GPU.

Although the aforementioned algorithms utilize GPU, they still follow the design ideas and techniques of CPU methods. Recent studies [9, 10, 48] have shown that methods designed based on CPU do not fully exploit the capabilities of the GPU. These solutions are tightly coupled with the type of query, limiting their scalability, and are complex to implement and costly to maintain. GPU native algorithms can make better use of GPU performance. RasterJoin [48] converts the point-in-polygon join operation into a rendering operation, effectively leveraging the high concurrency of GPU to solve spatial aggregation queries within arbitrary polygon areas. Harish [9] et al. conceptually designed a geometric data model and a series of algebras for spatial databases on GPU and demonstrated its expressiveness through a conceptual prototype. Our method might be an option for one of its operators in the future.

7 CONCLUSION AND FUTURE WORK

In this paper, we study two variants of point-in-polygon queries, *i.e.,* boolean point-in-polygon (bPIP) and integral point-in-polygon (iPIP). To efficiently answer those two queries, we exploit the GPU-native transformation and rasterization. We propose two methods, called RasterPIP-b and RasterPIP-i, for both queries, respectively. Each method judiciously chooses the storage mode for the essential information within the GPU to efficiently distinguish point IDs. Compared with traditional CPU-based methods, our solutions are insensitive to shape, side number, area and total number of query polygons. Compared to the GPU-based method, our method is obviously faster. Our methods obviously outperform their competitors, as demonstrated in the extensive experiments on real-world datasets. However, in some extreme scenarios, our methods are not always superior to traditional methods such as PostGIS. Therefore, in order to integrate them into existing systems, we need to design a cost model to decide whether to use our methods based on specific scenarios in the future.

REFERENCES

[1] Sarthak Agarwal and KS Rajan. 2017. Analyzing the performance of NoSQL vs. SQL databases for Spatial and Aggregate queries. In *Free and Open Source Software for Geospatial (FOSS4G) Conference Proceedings*, Vol. 17. 4.

[2] David Hook Andrew Glassner et al. 1990. Ray Tracing News. http://jedi.ks.uiuc.edu/~johns/raytracer/rtn/rtnv3n4.html

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. 322–331.

[4] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM* 18, 9 (1975), 509–517.

[5] Panagiotis Bouros and Nikos Mamoulis. 2019. Spatial joins: what's next? *SIGSPATIAL Special* 11, 1 (2019), 13–21.

[6] Lu Chen, Qilu Zhong, Xiaokui Xiao, Yunjun Gao, Pengfei Jin, and Christian S Jensen. 2018. Price-and-time-aware dynamic ridesharing. In *2018 IEEE 34th international conference on data engineering (ICDE)*. IEEE, 1061–1072.

[7] Judith R Davis. 1998. IBM's DB2 spatial extender: Managing geo-spatial information within the DBMS. *IBM corporation, May* (1998).

[8] Haowen Dong, Chengliang Chai, Yuyu Luo, Jiabin Liu, Jianhua Feng, and Chaoqun Zhan. 2022. Rw-tree: A learned workload-aware framework for r-tree construction. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2073–2085.

[9] Harish Doraiswamy and Juliana Freire. 2020. A gpu-friendly geometric data model and algebra for spatial queries. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1875–1885.

[10] Harish Doraiswamy and Juliana Freire. 2020. A gpu-friendly geometric data model and algebra for spatial queries: Extended version. *arXiv preprint arXiv:2004.03630* (2020).

[11] Harish Doraiswamy, Huy T Vo, Cláudio T Silva, and Juliana Freire. 2016. A GPU-based index to support interactive spatio-temporal queries over historical data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1086–1097.

[12] Ahmed Eldawy and Mohamed F Mokbel. 2015. Spatialhadoop: A mapreduce framework for spatial data. In *2015 IEEE 31st international conference on Data Engineering*. IEEE, 1352–1363.

[13] Ahmed Eldawy, Mohamed F Mokbel, et al. 2016. The era of big spatial data: A survey. *Foundations and Trends® in Databases* 6, 3-4 (2016), 163–273.

[14] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. 2008. Spatial indexing in microsoft SQL server 2008. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 1207–1216.

[15] Ziquan Fang, Shenghao Gong, Lu Chen, Jiachen Xu, Yunjun Gao, and Christian S Jensen. 2023. Ghost: A General Framework for High-Performance Online Similarity Queries over Distributed Trajectory Streams. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[16] Nivan Ferreira, Jorge Poco, Huy T Vo, Juliana Freire, and Cláudio T Silva. 2013. Visual exploration of big spatio-temporal urban data: A study of new york city taxi trips. *IEEE transactions on visualization and computer graphics* 19, 12 (2013), 2149–2158.

[17] Raphael A Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4 (1974), 1–9.

[18] James D Foley. 1996. *Computer graphics: principles and practice*. Vol. 12110. Addison-Wesley Professional.

[19] Michael Galetzka and Patrick O. Glauner. 2017. A Simple and Correct Even-Odd Algorithm for the Point-in-Polygon Problem for Complex Polygons. In *VISIGRAPP*. 175–178.

[20] Yunjun Gao, Ziquan Fang, Jiachen Xu, Shenghao Gong, Chunhui Shen, and Lu Chen. 2023. An Efficient and Distributed Framework for Real-Time Trajectory Stream Clustering. *IEEE Transactions on Knowledge and Data Engineering* (2023).

[21] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[22] Kai Hormann and Alexander Agathos. 2001. The point in polygon problem for arbitrary polygons. *Computational geometry* 20, 3 (2001), 131–144.

[23] Shuai Huang, Yong Wang, and Guoliang Li. 2023. ACR-Tree: Constructing R-Trees Using Deep Reinforcement Learning. In *International Conference on Database Systems for Advanced Applications*. Springer, 80–96.

[24] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)* 32, 1 (2007), 7–es.

[25] John Kessenich, Graham Sellers, and Dave Shreiner. 2016. *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. Addison-Wesley Professional.

[26] Guoliang Li, Shuo Chen, Jianhua Feng, Kian-lee Tan, and Wen-syan Li. 2014. Efficient location-aware influence maximization. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 87–98.

[27] Hui Li, Qiting Yang, and Jiangtao Cui. 2022. a2RegInf: An Interactive System for Maximizing Influence within Arbitrary Number of Arbitrary Shaped Query Regions. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*. 1585–1588.

[28] Chengyang Luo, Qing Liu, Yunjun Gao, Lu Chen, Ziheng Wei, and Congcong Ge. 2023. Task: An Efficient Framework for Instant Error-Tolerant Spatial Keyword Queries on Road Networks. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2418–2430.

[29] Nicolas Moyroud and Frédéric Portet. 2018. Introduction to QGIS. *QGIS and generic tools* 1 (2018), 1–17.

[30] MySQL. 2023. *MySQL 8.0 Reference Manual*. https://dev.mysql.com/doc/refman/8.0/en/

[31] Thanh Thi Nguyen. 2009. Indexing PostGIS databases and spatial Query performance evaluations. *International Journal of Geoinformatics* 5, 3 (2009), 1.

[32] Varun Pandey, Andreas Kipf, Thomas Neumann, and Alfons Kemper. 2018. How good are modern spatial analytics systems? *Proceedings of the VLDB Endowment* 11, 11 (2018), 1661–1673.

[33] Juan Pineda. 1988. A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. 17–20.

[34] PostGIS PostGIS. 2013. Spatial and Geographic objects for PostgreSQL.

[35] PostgreSQL. 2023. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/

[36] Philippe Rigaux, Michel Scholl, and Agnes Voisard. 2002. *Spatial databases: with application to GIS*. Morgan Kaufmann.

[37] Hanan Samet. 2006. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann.

[38] Shashi Shekhar, Hui Xiong, and Xun Zhou (Eds.). 2017. *Oracle Spatial*. Springer International Publishing, Cham, 1522–1522.

[39] Jonathan Richard Shewchuk. 2002. Delaunay refinement algorithms for triangular mesh generation. *Computational geometry* 22, 1-3 (2002), 21–74.

[40] Moshe Shimrat. 1962. Algorithm 112: position of point relative to polygon. *Commun. ACM* 5, 8 (1962), 434.

[41] Jaewoo Shin, Jianguo Wang, and Walid G Aref. 2021. The LSM RUM-tree: a log structured merge R-tree for update-intensive spatial workloads. In *2021 IEEE 37th international conference on data engineering (ICDE)*. IEEE, 2285–2290.

[42] Bogdan Simion, Daniel N Ilha, Angela Demke Brown, and Ryan Johnson. 2013. The price of generality in spatial indexing. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 8–12.

[43] Xinyu Su, Jianzhong Qi, Egemen Tanin, Yanchuan Chang, and Majid Sarvi. 2024. Spatial-temporal Forecasting for Regions without Observations. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 488–500. https://doi.org/10.48786/EDBT.2024.42

[44] Ivan E Sutherland, Robert F Sproull, and Robert A Schumacker. 1974. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys (CSUR)* 6, 1 (1974), 1–55.

[45] Abdel Aziz Taha and Allan Hanbury. 2015. Metrics for evaluating 3D medical image segmentation: analysis, selection, and tool. *BMC medical imaging* 15, 1 (2015), 1–28.

[46] Dejun Teng, Furqan Baig, Qiheng Sun, Jun Kong, and Fusheng Wang. 2021. IDEAL: a Vector-Raster Hybrid Model for Efficient Spatial Queries over Complex Polygons. In *2021 22nd IEEE International Conference on Mobile Data Management (MDM)*. IEEE, 99–108.

[47] Guang Yang, Abhirup Ghosh, Liang Liang, and Thomas Heinis. 2024. In-Network Approximate and Efficient Spatiotemporal Range Queries on Moving Objects. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, Letizia Tanca, Qiong Luo, Giuseppe Polese, Loredana Caruccio, Xavier Oriol, and Donatella Firmani (Eds.). OpenProceedings.org, 34–46. https://doi.org/10.48786/EDBT.2024.04

[48] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T Silva, and Juliana Freire. 2017. GPU rasterization for real-time spatial aggregation over arbitrary polygons. *Proceedings of the VLDB Endowment* 11, 3 (2017), 352–365.

[49] Borut Žalik and Ivana Kolingerova. 2001. A cell-based point-in-polygon algorithm suitable for large sets of points. *Computers & Geosciences* 27, 10 (2001), 1135–1145.

[50] Jianting Zhang and Le Gruenwald. 2019. Efficient Quadtree Construction for Indexing Large-Scale Point Data on GPUs: Bottom-Up vs. Top-Down.. In *ADMS@VLDB*. 34–46.

[51] Jianting Zhang and Simin You. 2012. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 23–32.

[52] Geraldo Zimbrao and Jano Moreira de Souza. 1998. A raster approximation for processing of spatial joins. In *VLDB*, Vol. 98. 24–27.