Hardware Acceleration for Spatial Selections and Joins

Chengyu Sun  Divyakant Agrawal  Amr El Abbadi
Department of Computer Science
University of California, Santa Barbara
Email:{cysun, agrawal, amr}@cs.ucsb.edu

ABSTRACT
Spatial database operations are typically performed in two steps. In the filtering step, indexes and the minimum bounding rectangles (MBRs) of the objects are used to quickly determine a set of candidate objects, and in the refinement step, the actual geometries of the objects are retrieved and compared to the query geometry or each other. Because of the complexity of the computational geometry algorithms involved, the CPU cost of the refinement step is usually the dominant cost of the operation for complex geometries such as polygons. In this paper, we propose a novel approach to address this problem using efficient rendering and searching capabilities of modern graphics hardware. This approach does not require expensive pre-processing of the data or changes to existing storage and index structures, and it applies to both intersection and distance predicates. Our experiments with real world datasets show that by combining hardware and software methods, the overall computational cost can be reduced substantially for both spatial selections and joins.

Key words. hardware acceleration, spatial selection, spatial join

1. INTRODUCTION
Spatial databases are commonly used in applications such as geographical information systems (GIS) and computer-aided design (CAD) systems. The data stored in spatial databases are spatial objects such as locations, road segments, and geographical regions, which can be abstracted as geometries such as points, polylines, and polygons in a 2D or 3D coordinate system.

Spatial database queries are typically evaluated in two steps: the filtering step and the refinement step. In the filtering step, the minimal bounding rectangles (MBRs) of the objects and spatial indexes such as R-tree [1] are used to quickly determine a set of candidate results. In the refinement step, the final results are determined by retrieving the actual geometries of the candidates from the database, and comparing them to either a query geometry or to each other. For complex geometries such as polygons, the cost of the refinement step usually dominates the query cost due to the complexity of the underlying computational geometry algorithms.

The cost of the refinement step consists of two factors: the I/O cost of loading the geometries from disk to main memory, and the computational cost of geometry-geometry comparison. The ratio of the computational cost over the I/O cost varies significantly depending on the types of spatial queries and the complexity of the geometries, which can be roughly characterized by the number of vertices of a geometry. Generally speaking, the more complex the data, the more significant the computational cost. For instance, a recent study on spatial selections [2] shows that for point geometries, the I/O cost is the dominant factor, but for polygon geometries, both costs are significant. In the case of a spatial join, the computational cost could be orders of magnitude higher than the I/O cost, because once a geometry is loaded, it is buffered in the main memory and compared to many other geometries.

The high computational cost of the spatial operations comes from the complexity of the data in the real world, where it is not uncommon that a polygon has tens of thousands of vertices. Furthermore, the shapes of the polygons can be arbitrarily complex, as can be seen from Figure 1, which shows the first 100 polygons in the Wyoming land cover dataset (LANDC) and the Wyoming land ownership dataset (Lando). In many cases, the polygons are concave, and sometimes, even non-simple1. Processing these types of polygons is very expensive. For instance, assuming the numbers of vertices of two polygons are n and m, the complexity of the commonly used intersection test algorithm is $O((n + m) \log(n + m))$ [3], and the distance calculation algorithm is even more expensive with $O(n \times m)$ worst case complexity [4]. Since the I/O cost remains linear, the computational cost quickly outweighs the I/O cost as the complexity of data increases.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
© 2003 ACM 1-58113-634-X/03/06 ... $5.00.

1This work was partially supported by NSF grants IIS98-17432, EIA-0080134, IIS98-77142, IIS02-20512, and EIA-9986057.

1Non-simple polygons are polygons with self-intersecting edges or with vertices that have degrees greater than 2 (more than 2 edges incident to a vertex).
simple geometries such as convex hulls, For intersection queries, Brinkhoff et al. [5] proposed using proving spatial query processing for complex geometries. ware acceleration, is also included. The technique proposed in this paper, which is based on hard-

<table>
<thead>
<tr>
<th>Query Type</th>
<th>Processing Step</th>
<th>Pre-processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric Filter [5]</td>
<td>Intersection</td>
<td>Filtering</td>
</tr>
<tr>
<td>TR* Tree [5]</td>
<td>Intersection</td>
<td>Refinement</td>
</tr>
<tr>
<td>Rasterization Filter [6]</td>
<td>Intersection</td>
<td>Filtering</td>
</tr>
<tr>
<td>Tiling Filter [7]</td>
<td>Intersection</td>
<td>Filtering</td>
</tr>
<tr>
<td>Interior Filter [2]</td>
<td>Intersection and Containment</td>
<td>Filtering</td>
</tr>
<tr>
<td>0-Object Filter [4]</td>
<td>Distance</td>
<td>Filtering</td>
</tr>
<tr>
<td>1-Object Filter [4]</td>
<td>Distance</td>
<td>Refinement</td>
</tr>
<tr>
<td>Hardware Acceleration</td>
<td>Intersection and Distance</td>
<td>Refinement</td>
</tr>
</tbody>
</table>

Table 1: Techniques for Efficient Spatial Query Processing

Figure 1: Sample Objects from Two Dataset

Over the last decade, much effort has been directed to improving spatial query processing for complex geometries. For intersection queries, Brinkhoff et al. [5] proposed using simple geometries such as convex hulls, n-corners, and maximum enclosing rectangles to approximate complex polygons. These simple geometries serve as an intermediate filtering step, in addition to the MBR filtering, and can identify a significant number of false or positive hits without performing the costly geometry-geometry comparison. Brinkhoff et al. also proposed the TR* tree technique, which builds an index structure for each data object to speed up intersection detection directly. Recent work has proposed several tiling based intermediate filters [6, 7, 2], which approximate polygons with rectangular tiles. Tiling based filters are relatively easier to implement, and provide a certain degree of flexibility because the tiling levels can be adjusted according to the data. For distance queries, or more specifically, within-distance join, Chan proposed the 0-Object and 1-Object intermediate filters [4]. 0-Object filter uses only the MBRs of the objects, and 1-Object filter requires retrieving the actual geometry of one of the objects. Table 1 summarizes the work in this area by query type, processing step, and requirement of pre-processing. For comparison purpose, the technique proposed in this paper, which is based on hardware acceleration, is also included. The filtering techniques only access spatial indexes, assuming the polygon approximations are stored in index structures for better I/O performance [8], while the refinement techniques require access to the databases tables that hold the actual geometries. For the same query type, different filtering and refinement techniques can be arbitrarily combined. The techniques that do not require pre-processing, or runtime techniques, only rely on MBRs and the actual geometries, while the pre-processing techniques require pre-computation of the polygon approximations or auxiliary data structures such as TR* tree. Pre-processing techniques are more efficient for query processing, but suffer from two drawbacks. First, the pre-computation is usually expensive, which degrades update performance, and cannot be applied to cases where one of the datasets is an intermediate dataset, e.g. the results of a map overlay operation. Second, the pre-computed polygon approximations or auxiliary data structures have to be stored on disk. The extra storage and I/O costs may not be significant, but it still requires changes of existing storage or index structures, therefore it is more difficult to incorporate these techniques into commercial DBMSs [9].

In this paper, we propose using graphics hardware to speed up the refinement step. Our technique is based on the observation that most low-level algorithms spatial databases rely on have been well studied by the computational geometry community. Under current computer architectures, it is unlikely that algorithmic advances will significantly reduce the cost of geometry-geometry comparison. On the other hand, the last few years have seen tremendous advances in graphics hardware technologies. The performance and features that used to belong to high-end graphics workstations are now available on consumer graphics cards at affordable prices. Off-the-shelf graphics cards are capable of handling thousands of polygons in real time, and are widely used in computer games, 3D modeling, and virtual reality applications. Since both graphics hardware and spatial databases work on geometries such as points, lines, and polygons, and they both deal with geometric relations such as intersection and containment in a 2D or 3D space, it is only natural to exploit the computational power of graphics hardware to speed up spatial database operations.

Previous studies in computer graphics have shown successful use of graphics hardware for non-rendering purposes. In the early nineties, Rossignac et al. [10] showed that expensive geometric calculations for interactive 3D modeling can be greatly simplified by the use of high-end graphics workstations. In particular, to expose the interior of a 3D solid with a clipping plane or a clipping volume, the depth buffer and the masking operation can be used to provide a parity
check to determine if a point is in the interior of the solid or not. In recent years, with the performance price ratio of graphics hardware improving rapidly, new techniques have been developed to take advantage of the highly optimized hardware functions. For instance, the depth buffer and the stencil buffer are used in the RECODE algorithm [11] to accelerate 3D object collision detection, and depth calculation and comparison are used for computing generalized Voronoi Diagrams [12]. In 2001, Hoff et al. [13] discussed several ways to implement intersection tests for 2D geometries, which partially inspired our work. On the other hand, we also note that existing techniques cannot be directly applied to spatial databases for two reasons. First, techniques designed for graphics applications typically address objects in 3D space, and are not optimized to handle large numbers of 2D geometries. Second, visualization applications have relatively low accuracy requirements due to the limitations of human eyes, and in contrast, most spatial database applications expect the results to be absolutely correct. In this paper, we address these issues and make the following contributions:

- We analyze two runtime filtering techniques, the interior filter for intersection selection [2], and 0/1-Object filter for within-distance join, and show that despite effective filtering, the high computational cost of the refinement step remains a problem to be addressed.

- We propose a novel technique to accelerate the refinement step using graphics hardware with guaranteed accuracy. This technique does not require pre-processing of the data or change of existing index structures, and it is applicable to both intersection and distance predicates.

- We present a comprehensive performance study comparing the hardware accelerated algorithms to the leading software algorithms. The experimental results show that in most cases, the hardware algorithms outperform the software algorithms by a substantial margin.

The rest of the paper is organized as follows. Section 2 gives a brief overview of graphics hardware functions and the OpenGL software interface. In Section 3, we develop two hardware acceleration techniques, one for intersection queries and one for distance queries, and evaluate their performance in Section 4. Section 5 concludes the paper and discuss some future work.

2. GRAPHICS HARDWARE AND OPENGL RASTERIZATION PROPERTIES

In this section we introduce the graphics hardware rendering process, and the associated functions and terminologies. We also discuss point, line, and polygon rasterization properties of OpenGL. These rendering properties, defined in the OpenGL Specification [14], provide the basis for the accuracy guarantee in our hardware acceleration techniques.

2.1 Graphics Hardware Rendering Pipeline

From a programmer’s perspective, rendering a filled triangle on screen may be as simple as specifying the coordinates of the vertices and the color of the triangle, and the graphics hardware will take care of the rest, such as determining the deformation of the triangle due to different viewing angles, and the pixels inside the triangle that need to be colored. In recent years, the computational power of graphics hardware has improved tremendously. In 1999, hardware T&L, which offloads the calculations of transformation and lighting from CPU to graphics processor, first appeared in consumer graphics products. Currently, the top-of-the-line consumer graphics processor has 63 million transistors, while modern CPUs such as AMD AthlonXP and Intel Pentium 4 have only 37.5 and 55 million transistors, respectively.

There are two types of data inputs to the graphics hardware: vertex data which describes geometries, and pixel data such as images and fonts. After a series of processing stages, usually called a rendering pipeline, both types of inputs are transformed into pixels, stored in the frame buffer and are ready to be displayed on a screen. Figure 2 shows a simplified structure of the rendering pipeline.

![Figure 2: Rendering Pipeline](image)

For vertex data, the coordinates of the vertices in the data space are first transformed to the window coordinates on screen. Then the vertices are assembled into geometries, to which several operations are applied. For example, the geometries that are distant in perspective are made smaller, and the parts of geometries that are outside the viewing area are clipped. In the rasterization step, the geometries are converted to pixels. For example, the internal pixels of a filled polygon are calculated during this step. A pixel, often called a fragment, consists of several values including the RGBA color components and a depth value. Once the rasterization step is completed, several tests such as scissor test and depth test can be applied to each fragment to determine if the fragment should be written to the frame buffer.

The frame buffers consist of several different types of buffers: color buffer, depth buffer, stencil buffer, and accumulation buffer. In graphics hardware, different types of frame buffers not only serve as data storage, but also have various operations associated with them. For example, the blending operation is associated with the color buffers, and can blend the colors of two pixels when they are drawn to the same location; the depth test operation is associated with the depth buffer, and is used to decide whether a pixel is obscured by another based on their distances to the view point; the masking (or logical) operation is associated with all buffers, and applies a bit mask string to each pixel before it is writ-
ten into a buffer; and finally, the accumulation buffer, which also stores the RGB data of the pixels, is commonly used to produce effects such as motion blur by accumulating several images in the color buffer into a composite image. The techniques presented in this paper use mainly the color buffer and the accumulation buffer.

2.2 OpenGL Rasterization Properties

Applications access hardware functions through an application programming interface (API). The graphics API we use in this paper is OpenGL, which is available on multiple platforms including Linux and MS Windows. The OpenGL-compliant hardware, which include virtually all modern graphics processors, guarantee a number of properties when rendering geometries such as points, line segments, and polygons.

2.2.1 Window Coordinates and Point Rasterization

A rendering window is a two dimensional array of pixels, and can be considered as a grid with integer coordinates. Each grid cell represents a pixel, which is identified by the coordinates of its lower-left corner. For example, Figure 3(a) shows a 3 × 3 rendering window, and (1, 1) refers to the pixel in the center.

![Figure 3: Rasterization](image)

To rasterize a point, the data coordinates \((x, y)\) of the point are first translated to the window coordinates \((x_w, y_w)\). The window coordinates are then truncated to integers, and the pixel \(\lfloor x_w \rfloor, \lfloor y_w \rfloor \) is colored. Note that most modern graphics processors have built-in 32bit FPUs, therefore it is safe to assume that there is no accuracy loss during the coordinate translation from \((x, y)\) to \((x_w, x_c)\). On the other hand, two different points in the data space may still end up being rasterized to the same pixel because of the truncation. For example, Figure 3(b) shows that both point \((1.1, 1.1)\) and point \((1.9, 1.9)\) result in the center pixel being colored.

2.2.2 Line Segment Rasterization

For basic line segment rasterization, a diamond shape \(R_f\) is defined for each pixel centering at window coordinates \((x_f, y_f)\), where \(R_f = \{(x, y) | |x - x_f| + |y - y_f| < 1/2\}\). Given a line segment whose starting point is \(p_0\) and ending point is \(p_e\), all the pixels whose diamond-shape intersect the line segment will be colored, except the pixel whose diamond-shape contains \(p_e\). This rule is sometimes referred to as the diamond-exit rule, which ensures that when a set of connected line segments are rasterized, the end points will not be colored twice. An example is shown in Figure 3(c), where a line segment intersects the diamond-shapes of three pixels. The first two pixels are colored, but the last one is not, because the line segment “enters” the diamond-shape of the last pixel, but does not “exit”.

The problem with basic line segment rasterization is that line segments may simply “disappear”. For example, Figure 3(d) shows two line segments \(l_1\) and \(l_2\), where \(l_1\) does not intersect the diamond-shape of any pixel, and \(l_2\) intersect the diamond-shape of one pixel, but its ending point (assuming \(l_2\) pointing upward) is also inside the diamond-shape. By the diamond-exit rule, both line segments will not be rasterized. For our purpose, this behavior is highly undesirable.

![Figure 4: Anti-aliasing](image)

Due to the size and shape of the pixels, rasterized line segments sometimes appear to be “jagged”, as the one shown in Figure 4(a). Anti-aliasing is a technique developed to address this problem by rendering the pixels close to the jagged edges with a slightly different color. To rasterize an anti-aliased line segment with width \(w\), a bounding rectangle is first constructed. Two of the rectangle edges are parallel to the line segment, and the distance to the line segment is \(w/2\); the other two edges are perpendicular to the line segment, and intersect the line segment at its two end points. Figure 4(c) show a line segment with its bounding rectangle, where the boundary of the rectangle is shown in dotted line in Figure 4(b). For simplicity, we assume the line width is \(\sqrt{2}\), which is the length of the pixel diagonal.

The color of a pixel for an anti-aliased line segment is determined by the color of the line segment and the alpha value of pixel, where the alpha value is the ratio of the pixel coverage by the bounding rectangle. For example, suppose that the line color is \((0, 0, 0)\) (black), and the background color is \((1, 1, 1)\) (white). If half of a pixel is covered by the bounding rectangle, then the alpha value of the pixel is 0.5. When the blending function is enabled, the color of the pixel is \(0.5 \times (1, 1, 1) + (1 - 0.5) \times (0, 0, 0) = (0.5, 0.5, 0.5)\), which is gray. On the other hand, if the blending function is disabled, the alpha value is ignored and the color of the pixel will be the color of the line segment. Figure 4(c) and (d) show an anti-aliased line segment with blending enabled and disabled, respectively. In our implementation, all line segments are rendered with anti-aliasing enabled and blending disabled.
2.2.3 Polygon Rasterization

Compared to point and line segment rasterization, the algorithm for polygon rasterization is more involved, but no matter what algorithm is used, the resulting pixels must follow two simple rules. First of all, a pixel is colored only if its center lies inside the polygon. Secondly, when the center of a pixel lies on the boundary of a polygon, the pixel may or may not be colored; however, in the case where the pixel center lies on the common edge (with identical endpoints) of two polygons, it is required that the pixel is colored exactly once.

3. HARDWARE-ASSISTED INTERSECTION AND DISTANCE TESTS

The general strategy to test for intersection of two polygons with graphics hardware is quite straightforward:

- Render the first polygon with color \( c_1 \)
- Render the second polygon with color \( c_2 \)
- Search the frame buffer for overlapping pixels with color \( c_1 + c_2 \). If such pixels exist, these two polygons intersect each other.

An example of this strategy is illustrated in Figure 5(a), where the two polygons are rendered with color gray, and the overlapping pixels are black. Note that this strategy can also be used for distance tests. For example, to determine if two polygons are within distance \( D \), we simply expand each polygon boundary by \( D/2 \), as shown in Figure 5(b).

Calculating a new set of vertices for an expanded polygon is expensive in software, but performing this operation with graphics hardware is very efficient using anti-aliased line segments.

![Figure 5: Hardware Intersection and Distance Tests](image)

Because of the versatility of graphics hardware, there are a number ways to implement this strategy. For example, Hoff et al. [13] suggested four possible implementations, using hardware blending, logical operations, depth buffer, and stencil buffer, respectively. However, there are two important problems remain unsolved. First of all, in order to achieve maximum performance for the most common tasks, graphics hardware is designed to support only a limited set of geometries, which include points, line segments, and convex polygons. Concave polygons, which are common in spatial datasets, must be triangulated before they can be rendered by hardware. Polygon triangulation, which has to be performed in software, is much more expensive than hardware operations. Secondly, the coordinates of publicly available GIS data typically have 4 to 6 digits accuracy, or about 11 to 18 bit accuracy in binary form. For example, the longitude is between -180.000 and 180.000, and the latitude is between -90.000 and 90.000. On the other hand, the resolution of a rendering window is rather limited. For instance, a 32 × 32 rendering window only provides 5 bit accuracy in each dimension. This disparity between the data resolution and the window resolution means that the hardware approach may produce false results, which are usually not acceptable for spatial database applications.

In the rest of the section, we first present our hardware assisted intersection and distance algorithms which address these issues. We then discuss two implementation details which further improves efficiency.

3.1 Hardware-assisted Intersection and Distance Tests

Let \( P = \{p_0, p_1, \ldots, p_n\} \) and \( Q = \{q_0, q_1, \ldots, q_m\} \) be two simple polygons with vertices \( p_i \) and \( q_j \), respectively. Given \( P \) and \( Q \), the software intersection test [3] consists of two steps:

- **Point-in-Polygon Test.** Take any vertex of a polygon, and test if this vertex is inside the other polygon. The commonly used algorithm [15] for this step shoots a ray from the vertex, and count the number of proper intersections of the ray with the polygon edges. If the count is odd, the vertex is inside the polygon, and \( P \) intersects \( Q \); otherwise it rules out the case where one polygon is completely inside the other.
- **Segment Intersection Test.** Test if each edge of one polygon intersects any edge of the other polygon. The commonly used algorithm [3] for this step is plane-sweep, which sweeps a horizontal (or vertical) line through \( P \) and \( Q \). Edges that intersect the sweep line at the same time are tested against their immediate left and right neighbors for intersection. If two edges from different polygon intersect, \( P \) intersects \( Q \); otherwise \( P \) and \( Q \) are disjoint.

We note that in the software intersection test, the complexity of the Point-in-Polygon Test is \( O(n + m) \). This step is also cache efficient, since the vertices are accessed sequentially. The Segment-Intersection Test, however, has to maintain a random access structure (usually a balanced search tree such as AVL and Red-Black tree), and has the worst case complexity of \( O((n + m)\log(n + m)) \). Based on this observation, we propose the following algorithm using hardware segment intersection test as an additional filtering step.

**Algorithm 3.1. Hardware-assisted Intersection Test**

Given \( P \) and \( Q \), return true if \( P \) and \( Q \) intersect, and false otherwise

1. **Software Point-in-Polygon Test.** return true if test succeeds.
2. **Hardware Segment Intersection Test**
   2.1 Enable anti-aliasing
   2.2 Clear the color buffer and the
accumulation buffer
2.3 Render the edges of the first polygon with color (0.5,0.5,0.5)
2.4 Copy the content of the color buffer to the accumulation buffer
2.5 Render the edges of the second polygon with color (0.5,0.5,0.5)
2.6 Copy the content of the color buffer to the accumulation buffer
2.7 Load the content of the accumulation buffer back to the color buffer
2.8 Return false if the color (1,1,1) is not found in the color buffer

3. Software Segment Intersection Test

We make the following observations about Algorithm 3.1:

- The correctness of Algorithm 3.1 depends on the hardware segment intersection test, which should not filter out any positive results. This is ensured by the OpenGL rendering properties of anti-aliased line segments. As we discussed in Section 2.2.2, with anti-aliasing enabled, every pixel that intersect the line segment is colored, therefore if two line segments intersect, there exists at least one pixel that is colored twice.

- From an algorithmic perspective, Algorithm 3.1 may be considered as worse than the original software algorithm, since it includes an additional hardware test. However, note that spatial queries process large numbers of geometries, and the overall cost is reduced if the negative results can be identified quickly without going through the expensive software segment intersection test. The hardware segment intersection test serves the same purpose as existing filtering techniques using polygon approximations, except that it is performed in the refinement stage with the actual geometries.

- Algorithm 3.1 renders polygons as chains of segments instead of filled polygons, therefore avoids the costly triangulation step. The drawback is that the point-in-polygon test is still required in order to handle the case where one polygon is completely contained in the other. However, note that point-in-polygon test uses a very efficient linear algorithm, while triangulation, although can also be performed in linear or near linear time, is far more complicated.

Algorithm 3.1 can be easily extended to perform distance test, which determines if two polygons are within distance $D$. The general approach is to widen the polygon edges, including the end points, by $D$, as shown in Figure 6. Let $w \times h$ be the dimension of the data space, and $n \times n$ be the rendering window resolution, the line width and point width in pixels can be calculated with the following equations:

$$\text{LineWidth} = \text{PointWidth} = \lceil \frac{\max(w,h) \times D}{n} \rceil$$  \hspace{1cm} (1)

It should be pointed out that most graphics hardware has limits for maximum allowable line and point width. In our current implementation, we simply revert back to the software algorithm if $\lceil \frac{\max(w,h) \times D}{n} \rceil$ exceeds these limits.

3.2 Implementation Details

In this subsection, we discuss two implementation details, namely, searching the frame buffer and projecting the data space to the window space, which have a significant performance impact on the hardware-assisted algorithms.

Searching the frame buffer for a particular value could be expensive if not implemented properly. In our implementation, we take advantage of the hardware Minmax function, which returns the minimum and maximum color values when moving a block of pixels within the frame buffer. This approach avoids transferring the pixel data from the video memory back to the main memory, which would require the data going through the video memory bus, the AGP bus, the main memory bus, and the frontside bus. Considering that for a single spatial selection or join query, this operations has to be performed thousands or millions of times, such cost savings become very important.

Another factor that has a large impact on performance is the projection of the data space to the rendering window. For intersection test, we project the intersection region of the two bounding boxes to the rendering window, as shown in Figure 7(a). For distance test with distance $D$, we project the expanded bounding rectangle of the smaller object to the rendering window. This approach maximizes the utilization of the window resolution, and avoids rendering unnecessary edges of the polygons.

4. PERFORMANCE EVALUATION
We evaluate the performance of the hardware-acceleration techniques for three classes of spatial queries: intersection selection, intersection join, and within-distance join (or buffer query [4]). In this section, we discuss the experimental setup and analyze the results.

4.1 Experimental Setup

The experiments are performed on a desktop PC with an AMD AthlonXP 1800+ CPU and 1GB Double Data Rate (DDR) memory. The graphics card is equipped with an NVIDIA GeForce4 Ti4600 processor and 128MB on-board memory.

4.1.1 Query Processing and Cost Measurements

The queries are processed in three stages, MBR filtering, intermediate filtering, and geometry comparison, as shown in Figure 8.

Figure 8: Query Processing

MBR filtering uses the MBRs of the objects to produce a set of candidate results. For intersection selection, the candidates are the objects whose MBR intersects the query MBR. For intersection join, the candidates are the object pairs whose MBRs intersect each other. For within-distance join, the candidates are object pairs whose MBRs are within distance $D$. Note that the distance between two MBRs is a lower bound of the distance between two objects.

For intermediate filtering, we implemented two runtime filters, the interior filter [2] for intersection selections, and the 0-Object and 1-Object filters [4] for within-distance join. The interior filter partitions the query polygon into $2^l \times 2^l$ tiles, and keeps the tiles that are completely inside the query polygon as an approximation of the polygon interior. Figure 9(a) shows an example of the interior filter with tiling level 2. The three shaded tiles are the interior tiles. Given an object, the interior filter identifies the object as a positive result if the MBR of the object is completely covered by the interior tiles. For within-distance join, the 0-Object and 1-Object filters compute an upper bound of the distance between a pair of objects. If this upper bound is less than or equal to $D$, the object pair is identified as a positive result.

Geometry comparison processes objects that cannot be determined by MBR filtering or intermediate filtering. Actual geometries are retrieved, and intersection or distance test are performed in this stage. For software intersection test, we implemented a plane-sweep algorithm [3] with the restricted search space optimization [5], which performs the segment intersection test only on edges that intersect both MBRs (the bold edges shown in Figure 9(b)). This optimization does not reduce the theoretical complexity of the algorithm, but in practice provides about 30% to 40% performance improvement. For software distance test, we implemented a modified version of the $\text{minDist}$ algorithm by Chan [4]. This algorithm identifies a frontier chain in each polygon (the bold edges in Figure 9(c)), and computes the minimum distance between these two chains instead of the whole polygons. In our implementation, we augment the original $\text{minDist}$ algorithm with two simple optimizations. First, for within-distance queries, the algorithm returns as soon as the current distance is found to be less than the given distance $D$. Second, after the frontier chains are identified, we extend the MBRs by $D$ in each direction, and only calculate the distances between parts of the frontier chains that intersect the extended MBRs, as shown in Figure 9(d). This optimization is similar to the restricted search space approach for intersection test, and in practice reduces the computational cost by a factor of 2 to 6.

To compare the performance of different solutions, we measure the computational costs of each processing step using wall clock time. One measurement that is noticeably missing is the I/O cost, which is not included in this paper for two reasons. First, I/O cost depends mainly on the index structures used, which are not the focus of this paper. Second, in all of our experiments, including selections, the costs of loading the full datasets is one to three orders of magnitudes less than the computational costs, therefore have little to no impact on the overall performance.

4.1.2 Datasets and Querysets

The experiments are conducted with the following real world datasets:

- **LANDC** [16]. Land cover (mostly vegetation types) information for the state of Wyoming at 1:100,000 scale.
- **LANDO** [17]. Land ownership and management information for the state of Wyoming at 1:100,000 scale.
- **STATES50** [18]. The boundaries of the 50 US states (excluding islands) at 1:2,000,000 scale.
- **PRISM** [19]. Average annual precipitation in the contiguous United States at 1:2,000,000 scale for the climatological period 1961-1990.
EarthEngine/CHM 2017

Some statistics of the datasets are summarized in Table 2, where \( N \) is the number of objects in a dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>( N )</th>
<th>Number of Vertices Per Polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>LANDC</td>
<td>14,731</td>
<td>3 4,397 192</td>
</tr>
<tr>
<td>LANDO</td>
<td>33,860</td>
<td>3 8,807 20</td>
</tr>
<tr>
<td>STATES50</td>
<td>31</td>
<td>4 10,744 138</td>
</tr>
<tr>
<td>PRISM</td>
<td>6,243</td>
<td>3 29,556 68</td>
</tr>
<tr>
<td>WATER</td>
<td>21,866</td>
<td>3 39,360 91</td>
</tr>
</tbody>
</table>

Table 2: Statistics of Some Polygon Datasets

For intersection selection, we use the fifty state boundaries in STATES50 as a query set, and report the average cost per query. For within-distance join, we calculate a base distance with the following equation:

\[
BaseD = \frac{\sqrt{w_1 \times h_1} + \sqrt{w_2 \times h_2}}{2} \tag{2}
\]

where \( \bar{w_1} \) and \( \bar{h_1} \) are the average MBR width and height of the two datasets. The distance \( D \) used in the experiments are \([0.1, 0.5, 1.0, 2.0, 4.0] \times BaseD \), with \( 0.1 \times BaseD \) representing queries about close vicinity, and \( 4 \times BaseD \) covering a reasonably long distance.

4.2 Performance of Intersection Selection

Figure 10 shows the average cost breakdown for intersection selections with software-only intersection test. The X-axis is the tiling levels of the interior filter, and the datasets used in these experiments are WATER and PRISM. We note from Figure 10 that MBR filtering is very efficient. In fact, the MBR cost is so low (around 1 milli-second or less) that the curve overlaps with the X-axis. The total query cost is determined by the costs of interior filter and geometry comparison. As the tiling level increases, more objects can be identified by the interior filter as positive results, so the cost of geometry comparison decreases. At higher tiling levels, the overhead of the interior filter becomes significant, and the total query cost increases.

A somewhat surprising result from Figure 10 is that the performance improvement introduced by the interior filter is quite limited. Even if we assume interior filtering is free\(^2\), we note that at tiling level 4, the geometry comparison cost is only reduced by less than 10% for both datasets. The reason is that interior filter only identifies positive results; negative results that pass the MBR filter still have to go through the geometry comparison step. Also the positive results identified by the interior filter are the cases where one polygon is completely contained in the other. Such cases can be processed efficiently by the point-in-polygon test in the geometry comparison step.

\(^2\)The cost of interior filter is mainly the overhead of computing the interior tiles. This overhead is amortized over the number of objects processed, thus is negligible for large datasets.

It should be pointed out that this experiment is not intended as an evaluation of the interior filter. For example, the I/O cost savings of the interior filter is not included, and intersection is only one of the spatial predicates that the interior filter targets. However, this experiment does show that at least for some datasets and some predicates, the cost of the geometry comparison remains an issue despite intermediate filtering. In the rest of this section, we show that this issue can be addressed with proposed hardware techniques.

Figure 11 compares the average costs of geometry comparison using software and hardware-assisted intersection tests. The X-axis is the rendering window resolution, from \( 1 \times 1 \) pixel to \( 32 \times 32 \) pixels. The general observation is that the cost of the hardware-assisted approach first decreases as the window resolution increases, because more object pairs that are closely located but not intersecting each other can be distinguished at higher resolutions, and these object pairs are filtered out by the efficient hardware segment intersection test. As the window resolution keeps increasing, the overhead of the hardware segment intersection test becomes significant due to the larger number of pixels to be processed, and the cost of the hardware approach starts to increase. Compared to the software approach, the hardware approach reduces the geometry comparison cost by 42% to 56% for WATER, and 46% to 64% for PRISM. For both datasets, the hardware approach performs best with a \( 16 \times 16 \) rendering window.

An interesting result from Figure 11 is that even at the win-
4.3 Performance of Intersection Join

In this section we compare the performance of software and hardware algorithms for two intersection joins, LANDC $\gg$ LANDO and WATER $\gg$ PRISM. The results are summarized in Figure 12, where the X-axis shows the resolution of the rendering window from $1 \times 1$ and $32 \times 32$.

The general trend of the hardware performance is similar to the curve in the selection experiments. The cost of geometry comparison first decreases then increases as the window resolution increases. Compared to the software algorithm, the hardware intersection test reduces the computational cost by 68% to 80% for WATER $\gg$ PRISM, and in the best case, 38% for LANDC $\gg$ LANDO.

We note that for LANDC $\gg$ LANDO, the hardware approach becomes worse than the software approach at higher resolutions. This result brings up an important question: when should we use the hardware intersection test instead of the software algorithm? And the answer is that it depends on the hardware platform, rendering window resolution, as well as the complexity of the data. For a given platform and a window resolution, the hardware intersection test incurs a fixed overhead of searching the color buffer for overlapping pixels. The finer the window resolution, the more pixels have to be searched, which leads to a larger overhead. For complex geometries, this overhead is justified, but for simple geometries, this overhead may be more expensive than performing the software intersection test. To address this issue, we introduce a parameter called software threshold, or $sw_{threshold}$. Given two polygons with $n$ and $m$ vertices, if $n + m \leq sw_{threshold}$, Algorithm 3.1 skips the hardware segment test and performs the software segment test directly. Using $8 \times 8$ and $16 \times 16$ window resolutions as an example, we show the effect of this optimization for LANDC $\gg$ LANDO in Figure 13.

We note that the hardware performance first improves as the threshold increases until an optimal threshold is reached. For $16 \times 16$ resolution, this optimal threshold is around 900, and for $8 \times 8$, about 300, which correspond to different overheads incurred at different window resolutions. The performance then slowly degrades as the threshold increases, and although not shown in the figure, we can expect that the hardware curves will eventually converge to the software curve, since all intersection tests will be performed in software. One thing to be noted is that a reasonable threshold can be selected from a wide range, and in particular,
sensitive to the software threshold. For example, for $8 \times 8$ resolution, the performance difference using any threshold value between 0 to 1000 is within 12%. Overall, tuning software threshold provides some additional performance gains, but more importantly, it enables the hardware technique to adapt to data with various complexity.

4.4 Performance of Within-distance Join

In this section we study the performance of the hardware distance test for two within-distance queries, LANDC $\bowtie_{dis}$ LANDO and WATER $\bowtie_{dis}$ PRISM. For each query we also vary distance $D$ between $0.1 \times BaseD$ and $4 \times BaseD$.

The computational costs of the two queries using software distance test are summarized in Figure 14. Compared to intersection joins, within-distance joins are more expensive, because distance testing is a more general operation than intersection testing, which can be considered as a special case of distance testing where $D = 0$. We also note that despite very aggressive filtering (the 1-Object filter uses the actual geometry of the larger object), the geometry comparison cost still dominates the total cost.

Figure 15 compares the geometry comparison costs of the software distance test to the hardware distance test with window resolution from $1 \times 1$ to $32 \times 32$. The query distance in this experiment is $1 \times BaseD$, and the software threshold is 0. The results show that at lower resolutions, the effectiveness of hardware filtering improves with finer resolution, and at higher resolutions, the hardware overhead becomes significant, and the performance starts to degrade. We also note that rendering widened line segments is much more expensive than rendering line segments with the default line width 1. As a result, the hardware approach barely outperforms the software approach for LANDC $\bowtie_{dis}$ LANDO. On the other hand, for more complex datasets such as WATER and PRISM, the hardware approach still maintains a significant performance advantage, where the computational cost is reduced by 60% to 81%. The performance of the hardware approach can be further improved by tuning the software threshold. Similar to the results shown in Figure 13, such performance gains are generally minor (less than 10%) at lower window resolutions, but it helps to ensure that the hardware approach will perform at least as well as the software threshold.
 hardware approach.

![Cost of Geometry Comparison, Dataset1=LANDC, Dataset2=LANDO, Res=8x8](image1)

(a) LANDC $\triangleright_{\text{dis}}$ LANDO

![Cost of Geometry Comparison, Dataset1=WATER, Dataset2=PRISM, Res=8x8](image2)

(b) WATER $\triangleright_{\text{dis}}$ PRISM

Figure 16: Performance of Hardware Within-Distance Join, Resolution=$8 \times 8$, SW_Threshold=$500$

Figure 16 shows the hardware performance as a function of the query distance $D$, which varies from $0.1 \times \text{BaseD}$ to $4 \times \text{BaseD}$. The window resolution used in this experiment is $8 \times 8$, and the software threshold is set to be 500. From Figure 16, we note that the performance margin between the hardware algorithm and the software algorithm narrows as the query distance increases. For LANDC $\triangleright_{\text{dis}}$ LANDO, the performance improvement reduces from 43% to almost none, and for WATER $\triangleright_{\text{dis}}$ PRISM, from 83% to 74%. This is partly due to the increased overhead of rendering “thicker” lines, and partly due to the hardware limitation on the line width of anti-aliased edges, which is 10 pixels on our testing platform. As we discussed in Section 3.1, we expand the polygons by widening the polygon edges by $D$. If $D$ exceeds the line width limit, we revert back to the software distance test, and the performance of the hardware solution degrades to the software-only solution. We are currently working on a new approach that is insensitive to query distances.

5. CONCLUSION AND FUTURE WORK

In this paper we propose the use of computer graphics hardware to support spatial selections and joins for large complex datasets. Software solutions for such datasets are very expensive in terms of computational costs, and are usually performed only on high-end servers. In this paper, we have demonstrated the use of general-purpose graphics hardware to solve this problem, and the proposed techniques do not require expensive pre-processing or changes to existing storage or index structures. Performance of our techniques depends on hardware platform, complexity of the data, and the resolution of the rendering window. Experimental study shows that on our testing platform, $8 \times 8$ window resolution strikes a good balance between hardware and software workloads. For complex polygon data, these techniques provide up to 4.8 times speedup for intersection joins, and up to 5.9 times speedup for within-distance join. Further performance improvements can be gained by tuning the software threshold, which also enables the proposed techniques to adapt to various data complexity. In the near future, we will continue to improve the performance of the hardware-assisted algorithms, as well as integrate the current implementation into a commercial DBMS. We also plan to explore other spatial operations such as nearest neighbor queries using hardware calculated Voronoi diagrams [12].

6. REFERENCES


