Hardware Acceleration for Spatial Selection and Join

Chengyu Sun Divyakant Agrawal Amr El Abbadi

Department of Computer Science

University of California, Santa Barbara

Email:{cysun, agrawal, amr}@cs.ucsb.edu

Abstract

Unlike traditional relational database operations which are typically I/O bound, many spatial database operations are CPU-intensive due to the complexity of the spatial objects and the complexity of the underlying computational geometry algorithms. In this paper, we propose a novel approach to accelerate spatial selections and joins with computer graphics hardware. Our work is focused on hardware-based implementation of the polygon-polygon intersection test, which is the computational primitive for many spatial database operations. The general approach is to use graphics hardware to render the polygon interiors or boundaries, then check the frame buffer for overlapping pixels. Due to the disparity between the data resolution and the pixel resolution, the main challenge is to provide not only speed improvements, but also the high accuracy required by spatial database applications. We develop three approximations algorithms using hardware blending and accumulation functions. The blending-based algorithm is simple and fast, and the two accumulation-based algorithms provide provable guarantees on either precision or recall at a small cost of efficiency. We also show that by combining the accumulation-based algorithms with the conventional software approach, significant speedup can be achieved for complex data without sacrificing accuracy.

Key words. hardware acceleration, spatial selection, spatial join, polygon-polygon intersection test

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. In this paper we focus on polygon data and two types of spatial operations: intersection-base spatial selection and spatial join. Spatial selection requires retrieving in a dataset the objects that intersect a query object, and a spatial join retrieves all intersecting pairs of objects (a, b) in datasets A and B, where $a \in A$ and $b \in B$.

In most spatial database implementations, spatial operations are performed in two steps: the *filtering step* and the *refinement step*. In the filtering step, the minimal bounding rectangles (MBRs) of the objects and the query are used to quickly determine a set of candidate results. In the refinement step, the real geometries of the objects are retrieved from the database and sophisticated computational geometry algorithms are performed to determine the final results. Previously, most work in spatial databases was focused on the filtering step. A number of successful index structures such as R-tree [Gut84] were proposed, and as a result, the filtering step can be performed very efficiently, and the refinement has now become the performance bottleneck [BKSS94, KR01].

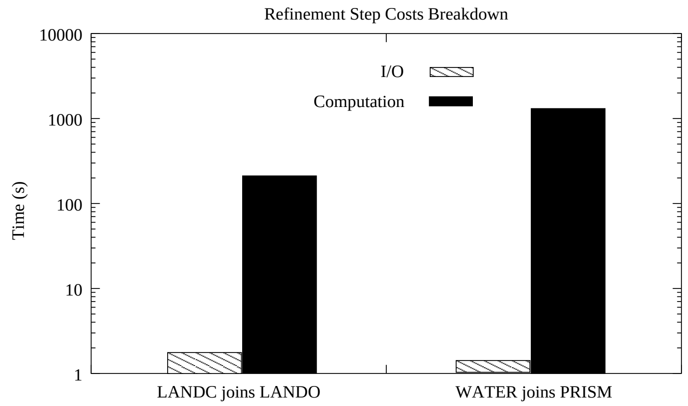
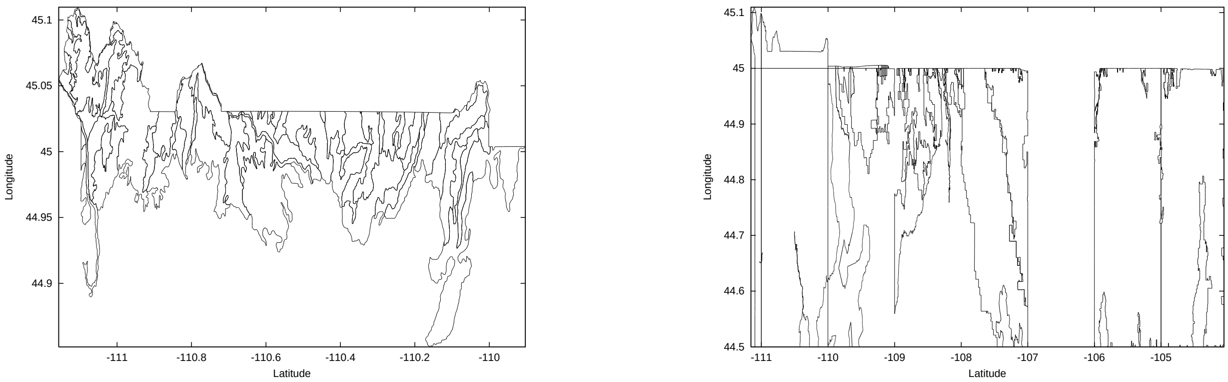


Figure 1: Refinement Step Costs Breakdown

The cost of the refinement step comes from two factors. First, the data geometries are loaded from hard disk to main memory, which incurs *I/O costs*, and secondly, the data geometries are compared to the query geometries (or in the case of spatial join, compared to each other), which incurs *computational cost*. For spatial selections, a recent study [KR01] shows that both costs are significant when the datasets consists of polygons; and for spatial joins, the computational cost completely dominates the I/O cost. For example, Figure 1 shows the cost breakdown of the refinement step in two spatial joins (detailed information about the datasets and the testing environment can be found in Section 5). Note that the computational cost is about two to three orders of magnitude higher than the I/O costs. The reason is that in a spatial join, the geometry of a data object is usually loaded once, buffered in main memory, and then compared to many other geometries.

The high computational cost of the spatial operations comes from the complexity of real world geometries. It is not uncommon that a polygon has tens of thousands of vertices. Furthermore, the shapes of the polygons can be arbitrarily complex, as we can see from Figure 2, which shows the first 100 geometries 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-simple[1]. Processing these types of polygons is very expensive. For instance, to test the intersection of two simple polygons with a total of N vertices, the optimal algorithm has a complexity of $O(N \log N)$ [DVOS00]. From an algorithmic perspective, an $O(N \log N)$ algorithm is usually considered as rather efficient, however, we have to keep in mind that for a single spatial selection or join, this algorithm has to be executed thousands or even millions of times.



(a) LANDC (b) LANDO

Figure 2: Sample Objects from Two Dataset

Algorithms for operations such as polygon-polygon intersection test are well studied in computational geometry. Many optimal or pragmatic solutions have already been implemented in spatial databases. Under current computer architectures, it is unlikely that algorithmic advances will significantly improve the performance of spatial databases. On the other hand, the last few years have seen great 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.

[1] Non-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).

Previous studies in computer graphics have shown successful use of graphics hardware for non-rendering purposes, but 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 efficient enough 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. To address these issues, we propose a new set of techniques that are designed for spatial database applications. In this paper, we focus on the polygon-polygon intersection test, which is a commonly used and expensive primitive that is likely to see the immediate benefit of hardware acceleration. The software algorithm for this primitive is well-established and has been proven to be optimal, therefore provides a good benchmark to measure the competitiveness of the hardware-based approach. We developed several hardware-based implementations of the polygon-polygon intersection test, which greatly reduce the computational costs for spatial selections and joins. We also show that by combining these techniques with software solutions, we are able to achieve full accuracy while still maintaining significant performance advantage over conventional software-only approaches.

The rest of the paper is organized as follows. Section 2 gives a brief overview of graphics hardware functions and related work. Section 3 discusses our general strategy and issues in implementing the polygon-polygon intersection test with computer graphics hardware. We present four hardware-based intersection algorithms in Section 4, and evaluate their performance in Section 5. Section 6 concludes the paper and discuss some future work.

2 Graphics Hardware and Related Work

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 3 shows a simplified structure of the rendering pipeline.

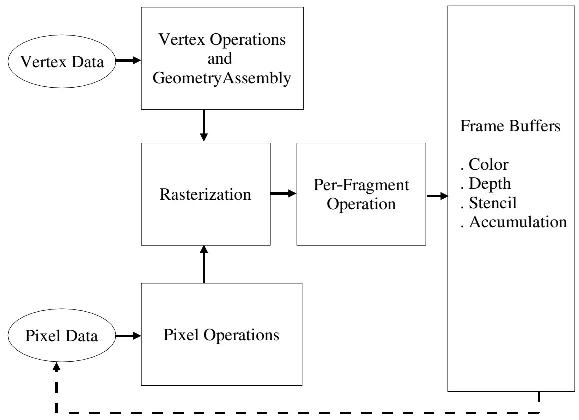


Figure 3: Rendering Pipeline

For vertex data, the coordinates of the vertices in the data space are first transformed to the window coordinates[2] 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, the 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 logic)* operation is associated with all buffers, and applies a bit mask string to each pixel before it is written into a buffer.

In the early nineties, Rossignac et al. showed that expensive geometric calculations for interactive 3D modeling can be greatly simplified by the use of high-end graphics workstations [RMS93]. In particular,

[2]The coordinate transformation itself may consist of several stages [SA99].

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 frame buffer and associated operations. For instance, the depth buffer and the stencil buffer are used in the RECODE algorithm [BWS99] to accelerate 3D object collision detection, and depth calculation and comparison are used for computing generalized Voronoi Diagrams [HCK+99]. In 2001, Hoff et al. discussed several ways to implement intersection tests for 2D geometries [HZLM01], which partially inspired our work.

3 General Implementation Strategy and Related Issues

In computer games or simulations, we frequently see dynamic scenes where an object moves in front of a background, and part of the background is blocked by the object. As the object moves, the blocked region, which is the intersection of the object and the background, is re-calculated continuously in real time. For a static example, Figure 4 shows the result of a code sample from the OpenGL Programming Guide [WNDS99]. In this example, two triangles are rendered with different colors, and their intersection region, which is computed by the graphics hardware, is rendered with a blended color of the two triangles.

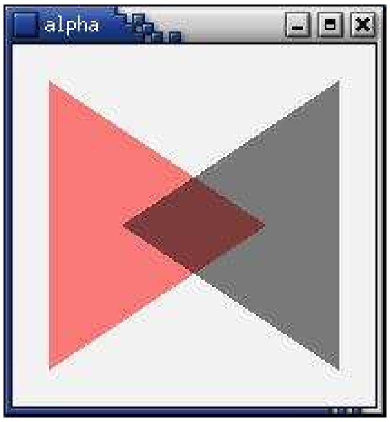


Figure 4: A Blending Example

Based on the observation that when programmed properly, the graphics hardware can automatically calculate the intersection region of two objects, and apply certain operations to this region, the general strategy to perform a polygon-polygon intersection test with graphics hardware can be described as follows:

- Specify the operations to be performed on an intersection region. Such operations should make the intersection region distinguishable from the original polygons. For instance, in the blending example, the intersection region is colored differently from the two triangles.

- Render the first polygon.
- Render the second polygon.
- Check the frame buffer for the existence of an intersection region.

Because of the versatility of graphics hardware, there are numerous ways to implement this strategy. However, regardless of the hardware function used, all implementations have to address some general issues in order to be practical and efficient for spatial database applications. In the rest of this section, we illustrate these issues using a straightforward implementation based on the hardware blending function. The datasets used in this section are the LANDC and LANDO datasets, which contain land cover and land ownership data for the state of Wyoming at the scale of 1:2,000,000. The empirical results are obtained on a desktop PC equipped with an AMD AthlonXP 1800+ CPU, an NVIDIA Geforce3 Ti200 graphics card, and an ATA100 hard disk (more details about the experimental setup is provided in Section 5).

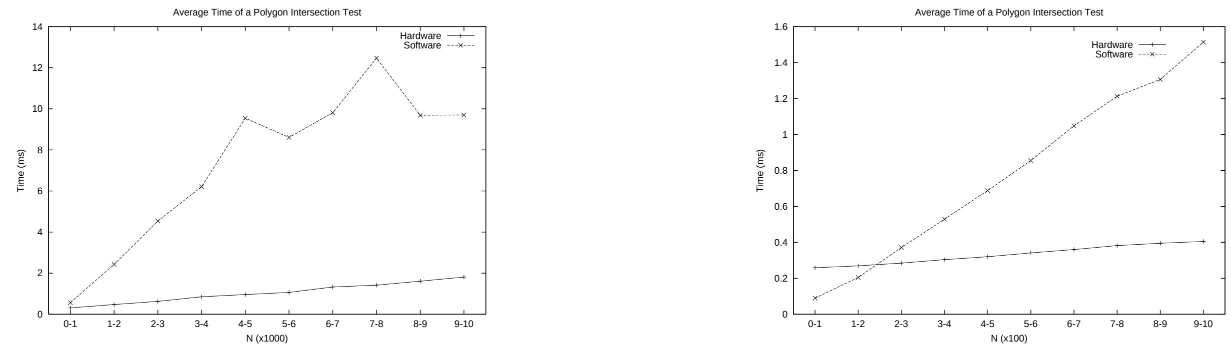
3.1 Efficiency Issues

In this subsection we discuss the efficiency of hardware-based polygon intersection tests. For issues such as the resolution of the rendering window that affect both efficiency and accuracy, we will focus on the efficiency aspect, and delay discussion of accuracy for the next subsection.

3.1.1 Complexity of the Polygons

To determine if two polygons intersect each other, a software solution only needs to process the vertices of the polygons, while a hardware implementation must render all points inside the polygons and/or the points along the edges. This implies that although hardware operations are typically more efficient than software operations, a hardware intersection test may not always outperform a software solution since more data has to be processed. Intuitively, a software solution is more suitable for polygons with a small number of vertices, while the hardware solutions may perform better for more complex polygons.

This intuition is confirmed by Figure 5, which compares the speed between the blending-based hardware intersection test and the optimal software intersection test with complexity of $O(N \log N)$ [DVOS00], where N is the sum of the vertices of the two polygons. From Figure 5(a), we see that the hardware solution is



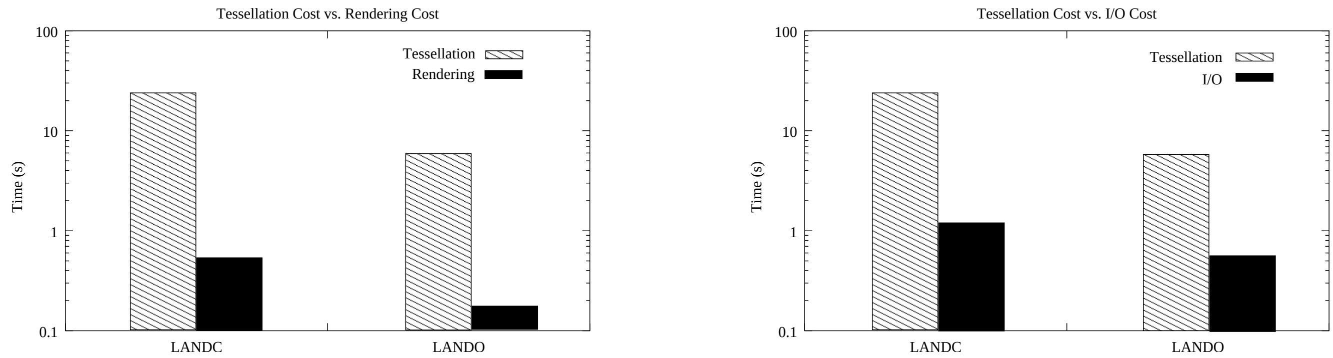
(a) Average Intersection Test Time, $N \leq 10,000$ (b) Average Intersection Test Time, $N \leq 1,000$

Figure 5: Speed Comparison between Hardware and Software Intersection Test

fairly insensitive to the number of vertices, while the performance of the software solution degrades rapidly as N increases. Figure 5(b) zooms into the polygon pairs whose total number of vertices is less than 1,000, and shows that when N is small, e.g. less than 250, the software solution has a speed advantage over the hardware solution. This also suggests that for spatial selections and joins, it is desirable to combine both the hardware and the software solutions.

3.1.2 Tessellation

In order to achieve optimal 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, such as the those found in LANDC and LANDO datasets (see Figure 2), must be triangulated before they can be rendered by the graphics hardware.



(a) Tessellation Cost vs. Rendering Cost (b) Tessellation Cost vs. I/O Cost

Figure 6: Tessellation

The triangulation process, usually referred to as *tessellation* in graphics terminology, is a rather expensive process compared to hardware operations. In our study, we use the tessellation function provided by

SGI's GLU[3] implementation as a benchmark to measure the tessellation cost. As shown in Figure 6(a), tessellating datasets LANDC and LANDO is almost ten times more expensive than actually rendering them. To avoid paying such high cost repeatedly for each query, a more sensible approach is to pre-process the dataset and store a triangulated copy of each object. Since a polygon with n vertices can be triangulated into $(n-2)$ triangles, the storage cost increases by roughly a factor of 3. However, in the cases where the storage space is not a constraint, the runtime performance improvement with pre-triangulated data is significant. For example, Figure 6(b) shows that the cost of loading both the original and the triangulated data from disk is only about 10% the cost of triangulating the original data. Moreover, most spatial database applications such as GIS and CAD systems have important visualization components, which also benefit from, or may even require, pre-triangulated data.

3.1.3 Window Resolution

Another factor that has a large impact on performance is the resolution of the window in which the polygons are to be rendered. Window resolution affects the efficiency of a hardware implementation of the polygon-polygon intersection test in two aspects. First, for each pixel in the rendering window, the graphics hardware has to determine if the pixel belongs to a polygon or not. Higher window resolution means that more pixels have to be processed. Second, to find out if an intersection occurs, we need to examine the pixel information stored in the frame buffers after the polygons are rendered. With higher window resolution, more pixels have to be examined since we cannot predict where the intersection may occur.

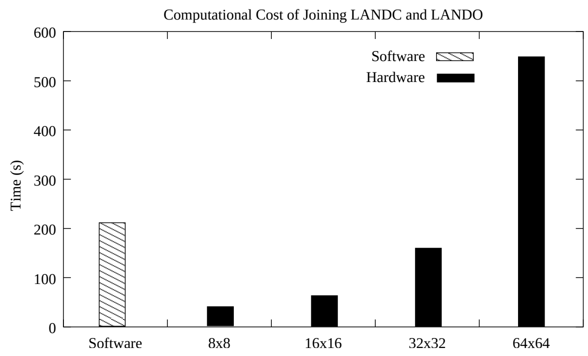


Figure 7: Computational Cost of Joining LANDC and LANDO

[3]GLU stands for OpenGL Utility Library, which provides some high level API on top of basic OpenGL functions to support more sophisticated geometries such as spheres, cylinder, and complex polygons.

Figure 7 shows the computational costs of joining the LANDC and the LANDO datasets. The rendering window resolutions used in the experiments range from 8×8 to 64×64. For comparison purposes, we also include the cost of the $O(N \log N)$ software algorithm. From Figure 7, we observe that the hardware approach has a significant performance advantage at lower windows resolutions. For example, with a 16×16 rendering window, the hardware approach outperforms the software approach by at least a factor of 3. However, as the window resolution increases, the hardware approach begins to lose its appeal. At 64×64 resolution, the hardware approach is about two and a half times as slow as the software approach.

3.1.4 Minmax Function

Now that we have discussed the pre-triangulation of the polygons and their rendering in a window with a proper resolution, the only task left is to check whether an intersection occurs, which involves searching the frame buffers for a particular value. For example, in the blending approach, an intersection is detected if we can find in the color buffer a pixel with the blended color.

An efficient way to implement this operation is to take advantage of the hardware Minmax function, which returns the minimum and maximum color value 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 extremely important.

3.2 Accuracy Issues

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, Figure 7 shows that in order for the hardware approach to be competitive against the software approach, the resolution of the rendering window should not exceed 32×32, which provides only 5 bit accuracy. The large disparity between the accuracy of the data coordinates and the window resolution means that the hardware approach alone simply cannot be completely accurate. However, all is not lost. Note that the rendering process consists of two stages. In the first stage, which we will refer to as *coordinate transformation*, the coordinates of a geometry in the data space is transformed to the coordinates in the window space. It is safe to consider that there is no accuracy loss in this stage, given the fact that even midrange graphics chips nowadays have 32-bit FPUs and provide single-precision float-point computation. The second stage, called *rasterization*, converts the geometries to pixels in a rendering window based on their window coordinates. Rounding or truncation errors may occur in this stage due to the limited resolution of a rendering window. However, despite such errors, graphics hardware must guarantee certain properties for the rasterization results in order to conform with well-known standards. One such standard is OpenGL, which is supported by virtually all current generation graphics hardware. In this section, we first give a brief introduction to the rasterization properties specified by the OpenGL 1.2 standard [SA99], and then we discuss a simple heuristic which significantly improves accuracy.

3.2.1 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 8(a) shows a 3×3 rendering window, and $(1, 1)$ refers to the pixel in the center.

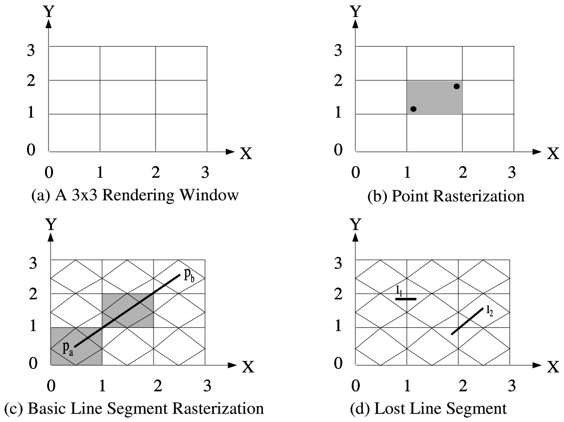


Figure 8: Rasterization

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. One thing to be noted is that although there is no accuracy loss during the coordinate translation from (x, y) to (x_w, x_w), two different points in the data space may still end up being rasterized to the same pixel because of the truncation. For example, Figure 8(b) shows that both point $(1.1, 1.1)$ and point $(1.9, 1.9)$ result in the center pixel being colored.

3.2.2 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_a and ending point is p_b, all the pixels whose diamond-shape intersect the line segment will be colored, except the pixel whose diamond-shape contains p_b. 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 8(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 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 8(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 line segment intersection detection, this behavior is highly undesirable.

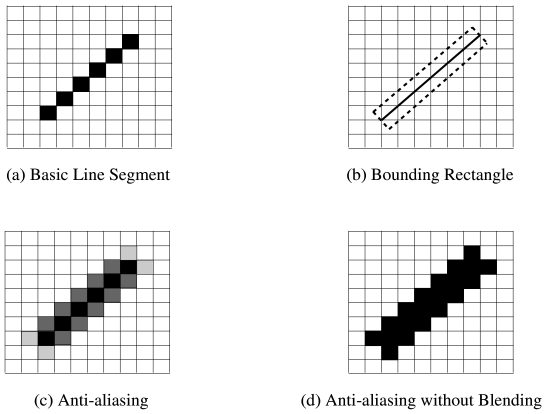


Figure 9: Anti-aliasing

Due to the size and shape of the pixels, rasterized line segments sometimes appear to be "jagged", as the one shown in Figure 9(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 9(c) show a line segment with its bounding rectangle, where the boundary of the rectangle is in dotted line. 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 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 is black. Figure 9(c) and (d) shows an anti-aliased line segment with blending enable and disabled, respectively.

3.2.3 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.

3.2.4 Data Space Projection

For the intersection test of two polygons, an easy way of reducing the disparity between the data resolution and the window resolution is to project to the window space only part the data space where the intersection may occur. There are several ways to decide the data space to be projected, as illustrated in Figure 10, where Figure 10(a) shows two objects and their bounding boxes (the dotted rectangles).

The first approach is to project the whole data space of a dataset, which is the minimal bounding rectangle that encloses all the objects in the dataset, to the rendering window, as shown in Figure 10(b). For spatial query processing, this approach has a slight performance advantage because the mapping can be done once for each query. However, for real world datasets, the size of an object is often very small compare to the size of the data space. So using this approach, most polygons become small dots or single pixels in the rendering window, making the intersection test highly inaccurate.

The second approach, illustrated in Figure 10(c), is to map the bounding box of one of the objects to

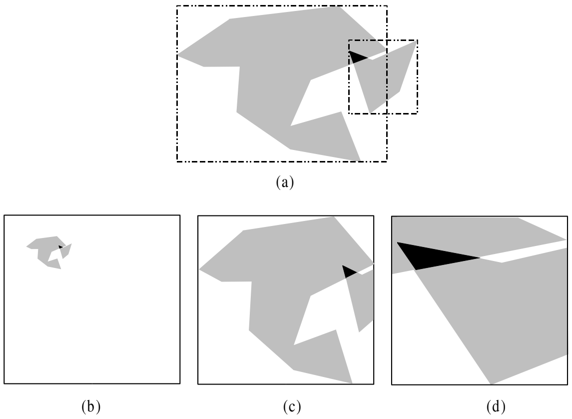


Figure 10: Data Space Projection

the rendering window. In our early experiments, we discovered that even within the same dataset, different objects may have vastly different sizes. If we map the bounding box of a very large object to the rendering window, then the result is similar to the first approach: the smaller object may be reduced to a single pixel, or in some cases simply disappear.

One way of addressing the problem of the second approach is to always map the bounding box of the smaller object to the rendering window. The third approach, shown in Figure 10(d), goes one step further by mapping only the intersection region of the two bounding boxes to the rendering window. This approach suffers a slight performance drawback since for each intersection test, the intersection region of the two bounding boxes must be calculated. However, experiments show that the overhead of calculating the bounding box intersection is negligible, while the accuracy of the intersection test improve greatly. One thing to be noted though is that in this approach, an actual intersection region may appear in any place within the rendering window, therefore the whole frame buffer of the rendering window must be examined to determine the occurrence of an intersection.

4 Hardware Accelerated Polygon-Polygon Intersection Tests

In this section we present four hardware-accelerated algorithms for the polygon-polygon intersection test. For different application requirements, these algorithm provide different speed/accuracy tradeoffs, ranging from fast approximation algorithms, to slower algorithms with accuracy guarantees provable under the following two assumptions:

- Assumption A. The graphics hardware is OpenGL 1.2-compliant.

- Assumption B. Coordinate translation is completely accurate, and hence errors only occur due to rasterization.

4.1 Polygon Rendering with Blending (BL)

The blending-based algorithm is first discussed by Hoff et al. [HZLM01] as one of four ways to detect intersection between 2D geometries using graphics hardware. In the previous section, we have used this algorithm to illustrate some efficiency issues for hardware-based intersection tests in general, and here we give the formal description of the algorithm as follows:

Algorithm 4.1 *Polygon Rendering with Blending (BL)*

```
1. Clear the color buffer
2. Enable the blending function
3. Render the first polygon with color (0.5, 0.5, 0.5)
4. Render the second polygon with color (0.5, 0.5, 0.5)
5. Search the color buffer for color (1.0, 1.0, 1.0)
```

The advantage of the BL algorithm is its simplicity and efficiency. However, this algorithm cannot provide any assurance on the accuracy of the results.

4.2 Polygon Rendering with Accumulation (PA)

In our early experiments, we noticed that for each selection or join test, the BL algorithm always produces a small number of false hits. Upon closer examination, we discovered that the problem was caused by the tessellation process.

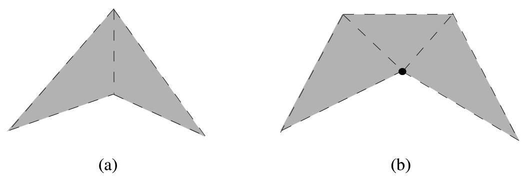


Figure 11: False Hits Caused by Tessellation

After tessellation, rendering a polygon becomes rendering a group of triangles, where the triangles may intersect each other by sharing edges as well as vertices. When two triangles share a common edge, as shown in Figure 11(a), the graphics hardware guarantees that no pixels on the common edge are colored twice. However, when more than two triangle share a vertex, the coloring of the shared vertex (such as the one shown in Figure 11(b)) is undefined. Sometimes such a pixel is colored twice, which may result in a false hit. Among the four algorithms described by Hoff et al. [HZLM01], three of them, which rely on the blending function, the logical operators, and the depth buffer operations, all suffer from this problem. The only other algorithm, which utilize the stencil buffer, does not have the false hit problem, but suffer from performance penalties because the stencil buffer does not support the hardware Minmax function. So in this section, we propose a new algorithm based on the accumulation buffer.

Like the color buffer, the accumulation buffer stores the RGB data of the pixels, and is commonly used to produce effects such as motion blur by accumulating several images in the color buffer into a composite image. Assuming that the non-convex polygons are pre-triangulated, we can implement a polygon-polygon intersection test as follows:

Algorithm 4.2 *Polygon Rendering with Accumulation Buffer (PA)*

```
1. Clear the color buffer and the accumulation buffer
2. Render the first polygon with color (0.5, 0.5, 0.5)
3. Copy the content of the color buffer to the accumulation buffer
4. Render the second polygon with color (0.5, 0.5, 0.5)
5. Copy the content of the color buffer to the accumulation buffer
6. Load the content of the accumulation buffer back to the color buffer
7. Search the color buffer for color (1.0, 1.0, 1.0)
```

Compared to the blending approach, the advantage of Algorithm 4.2 is that even if a pixel is colored twice in the color buffer, the color of the pixel is still $(0.5, 0.5, 0.5)$ since the blending function is disabled (by default). In the PA algorithm, the color accumulation happens in the accumulation buffer, after the triangles that belong to the same polygon are rendered. Therefore, this approach avoids the false hits problem caused by tessellation. Furthermore, based on Assumption A and B, we prove that the PA algorithm does not produce *any* false hits, whether the false hits are caused by tessellation or not.

Theorem 4.1 *The PA algorithm does not produce any false hits.*

Proof. For the PA algorithm to produce a hit (false or not), at least one pixel must have the accumulated color value $(1.0, 1.0, 1.0)$. From the properties of polygon rasterization, and considering tessellation, we know a pixel may be colored twice only in the following cases:

- The center of the pixel lies inside two triangles, or, it lies inside one triangle and on the boundary of another triangle.

  In this case, the two triangles intersect each other, and they must belong to different polygons because the tessellated triangles of the same polygon only share edges. Therefore, the result is not a false hit.

- The center of the pixel lies on the boundaries of two or more triangles, and the triangles belong to different polygons.

  In this case, by definition[4] the two polygon do intersect, so the result is not a false hit either.

- The center of the pixel lies on the boundaries of two or more triangles, and the triangles belong to the same polygon.

  In this case, the pixel may be colored twice, as the example shown in Figure 11(b). However, note that in the PA algorithm, all triangles of the same polygon are rendered with blending turned off, so even if a pixel is colored twice, the pixel will not have the accumulated value because second color value simply overwrite the first color value instead of adding to it. Therefore, not false hits will result from this case.

In conclusion, the PA algorithm does not produce any false hits. □

4.3 Edge Rendering with Accumulation (EA)

In this subsection we present an algorithm addressing the false dismissal problem, which usually occurs when the intersection region of the two polygons is very small. This algorithm is partially inspired by the software intersection test algorithm [DVOS00], which works as follows:

- Step 1. Take any vertex in a polygon, and check if this vertex is inside the other polygon. This step is called *Point in Polygon Test*, and is performed by shooting a ray from the vertex, and count the number of polygon edges that properly intersect the ray[5]. If the number is odd, the vertex is inside the polygon.

- Step 2. If the vertex is not inside the polygon, which means that one polygon does not contain the other, a *Line Segments Intersection Test* is performed. In this step, the edges of the two polygons are first sorted by their Y-coordinate, then a horizontal line is used to sweep the edges from top to

[4]Polygons sharing edges or vertices are considered intersecting.

[5]Edges that are collinear with the ray or intersect the ray at end points do not count.

bottom. The sweep line stops at each end point of the edges, where an edge is either inserted or removed from a balanced search structure such as a Red-Black tree. When an edge is inserted into the search structure, it is checked against its two neighboring edges; and when an edge is removed from the search structure, its two neighboring edges are checked against each other. This step ends once an intersection of the edges is found.

We note that in the software intersection test, the complexity of the first step is $O(N)$, where N is the total number of edges of the two polygons. The first step is also cache efficient, since the vertices are accessed sequentially. The second step, however, has to maintain a tree structure, and has a complexity of $O(N \log N)$. Based on this observation, we propose another hardware intersection algorithm using the software point-in-polygon test as an additional filtering step.

Algorithm 4.3 *Edge Rendering with Accumulation (EA)*

```
1. Perform software point-in-polygon test.
2. Clear the color buffer and the accumulation buffer
3. Enable anti-aliasing
4. Render the edges of the first polygon with color (0.5, 0.5, 0.5)
5. Copy the content of the color buffer to the accumulation buffer
6. Render the edges of the second polygon with color (0.5, 0.5, 0.5)
7. Copy the content of the color buffer to the accumulation buffer
8. Load the content of the accumulation buffer back to the color buffer
9. Search the color buffer for color (1.0, 1.0, 1.0)
```

By rendering the polygon edges, the EA algorithms has several desirable properties compared to previously discussed approaches. First, the tessellation step is no longer necessary, therefore the storage cost is reduced. Second, since the pixels inside the polygons are not calculated, the rendering performance improves. The third property, which is the most important but not immediately obvious, is that the EA algorithm does not produce any false dismissals because of the use of anti-aliased line segments. A proof for this property is as follows:

Theorem 4.2 *The EA algorithm does not produce any false dismissals.*

Proof. Since the first step of the EA algorithm performs a point-in-polygon test which does not produce any false dismissals, we only need to show that if two polygons intersect by having intersecting edges, at least one pixel will have the accumulated color.

Let l_1 and l_2 be two intersecting edges, and the intersection point is p (when l_1 and l_2 are collinear, any point on the shared segment can serve as p). When p is inside the rendering window, which is ensured by any of the three project methods discussed in Section 3.2.4, one of the pixels in the rendering window must intersect p. Consequently, this pixel must intersect the bounding rectangles of the two anti-aliased line segments, and therefore must be colored twice.

In conclusion, the EA algorithm does not produce any false dismissals. □

4.4 Combining Hardware and Software Approaches

The hardware techniques we discussed so far can be best described as approximate query processing techniques, which improve query processing time at the cost of reduced accuracy. From another perspective, the hardware techniques, in particular PA and EA, can also be regarded as secondary filters in addition to the MBR filtering, which reduces the workload of the refinement step.

Secondary filters can be classified as *conservative*, *progressive*, and *generalizing* [BKS93]. A secondary filter is conservative if the boundary of the original object is completely contained in the approximation. It is progressive if all the points of the approximation is contained in the original object. And for generalizing filters, neither of the two conditions above has to hold. Intuitively, conservative filters only filter out negative results, and progressive filters only accept positive results. In our case, the EA algorithm can be considered as a conservative filter, while the PA algorithm is progressive. Combining these two algorithms with the software algorithm as a final refinement step, a hybrid intersection test algorithm can be easily constructed as follows:

Algorithm 4.4 *Hybrid Polygon-Polygon Intersection Test (HT)*

```
1. Perform EA test. Return false if test fails.
2. Perform PA test. Return true if test succeed.
3. Perform Software test and return the result.
```

5 Performance of Spatial Selection and Join

In this section we study the performance of the hardware-accelerated intersection algorithms for answering spatial selection and join queries.

5.1 Experimental Setup

The platform for the experiments is an AMD AthlonXP 1800+ PC with 512MB Double Data Rate (DDR) memory. The graphics card is equipped with an NVIDIA GeForce3 Ti200 processor and 64MB video memory. The software package is Mesa 4.0.1, which is a free OpenGL implementation by Brian Paul.

The experiments were conducted with the following five real world datasets:

- **LANDC** [Wyo96a]. Land cover (mostly vegetation types) information for the state of Wyoming at 1:100,000 scale.
- **LANDO** [Wyo96b]. Land ownership and management information for the state of Wyoming at 1:100,000 scale.
- **STATES50** [Sur99]. The boundaries of the 50 US states (excluding islands) at 1:2,000,000 scale.
- **PRISM** [DT00]. Average annual precipitation in the contiguous United States at 1:2,000,000 scale for the climatological period 1961-1990.
- **WATER** [Wat01]. Hydrologic unit boundaries for the United States, Puerto Rico and the US Virgin Islands at 1:2,000,000 scale.

Some statistics of the datasets are summarized in Table 1, where N is the number of objects in a dataset.

Dataset	N	Number of Vertices Per Polygon		
		Min	Max	Average
LANDC	14,731	3	4,597	192
LANDO	33,860	3	8,007	20
STATES50	50	4	70,710	138
PRISM	6,243	3	29,556	68
WATER	21,866	3	39,360	91

Table 1: Statistics of Some Polygon Datasets

For polygon-polygon intersection test, we implemented **BL**, **PA**, **EA**, **HT**, as well as the optimal software algorithm (**SW**). In each selection or join experiment that involves hardware intersection test, we include a simple optimization suggested by Figure 5(b). Namely, when the sum of the vertices of two polygons is below a threshold, e.g., 200, we always use the software intersection test. The source code for the experiments is available at `http://www.cs.ucsb.edu/~cysun/cg.tar.gz`.

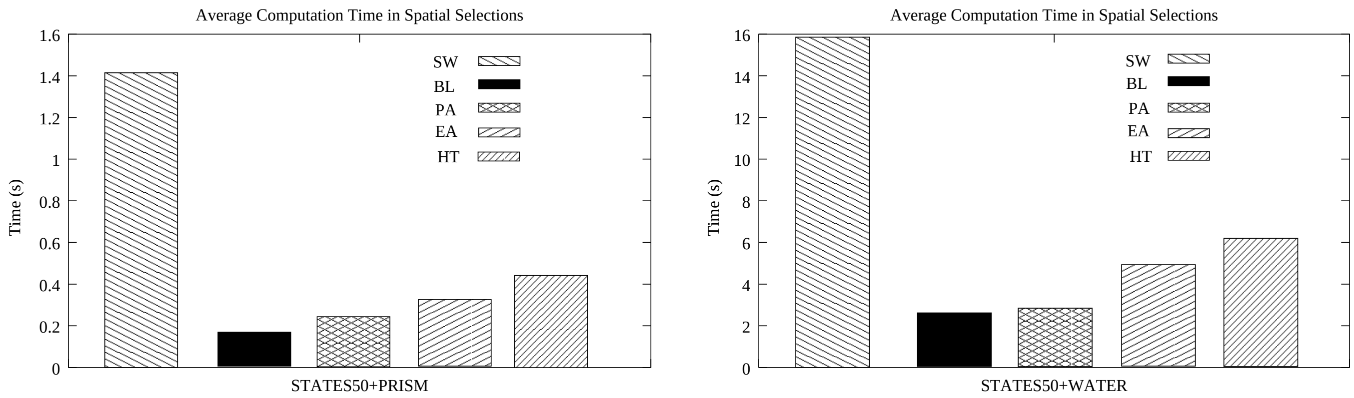
To compare the performance of different solutions in both speed and accuracy, we use the following measurements in the experiments:

- *Computation_Time.* This is the wall clock time of the refinement step, with all geometries loaded in the main memory. The cost of the filtering step is not included, because it typically constitutes less than 10% percent of the total cost, and is dependent on the index structures used.

- *Precision and Recall.* Precision and recall are measurements often used to characterize the rates of false hits and false dismissals. Let C be the total number of correct results, c be the number of correct results found, and R be the total number of results found, then precision is defined as c/R, and recall is defined as c/C. For example, if a hardware solution finds 100 results with 30 false hits and 20 false dismissals, then R is 100, C is 90, c is 70, and the precision and recall are 70% and 77.8%, respectively. For multiple queries $q_1, ..., q_i, ..., q_n$, the *average precision* is defined as $\sum c_i / \sum R_i$, and the *average recall* is defined as $\sum c_i / \sum C_i$.

5.2 Performance of Spatial Selections

In this section we present the results for spatial selection. The datasets used in this experiment are PRISM and WATER, and the query set is STATES50, which consists of 50 queries. For the hardware approaches, the resolution of the rendering window is 16×16, and the average computation time per query is shown in Figure 12.



(a) STATES50+PRISM (b) STATES50+WATER

Figure 12: Average Computation Time in Spatial Selections

We first observe that the hardware approaches uniformly outperform the software approach, by a factor between 2 and 7. Among the hardware approaches, the BL algorithm performs slightly better than the PA algorithm, because PA copies the content of the rendering window between the color buffer and the accumulation buffer back and forth three times, while BL performs the color blending, or, the "accumulation" of the color values, directly in the color buffer. The EA algorithm performs worse than BL and PA, because the software point-in-polygon test, though very efficient, is still not fast enough compared to hardware operations. Among the four hardware-accelerated algorithms, the HT algorithm is the slowest, but is still 2 to 3 times faster than the software-only approach.

Datasets	Alg.	False Hits	False Dismissals	Avg. Precision	Avg. Recall
STATE50+PRISM	BL	5	59	99.91%	98.90%
STATE50+PRISM	PA	0	59	100%	98.90%
STATE50+PRISM	EA	190	0	96.56%	100%
STATE50+WATER	BL	26	1383	99.85%	92.28%
STATE50+WATER	PA	0	1400	100%	92.19%
STATE50+WATER	EA	357	0	98.05%	100%

Table 2: Selection Errors

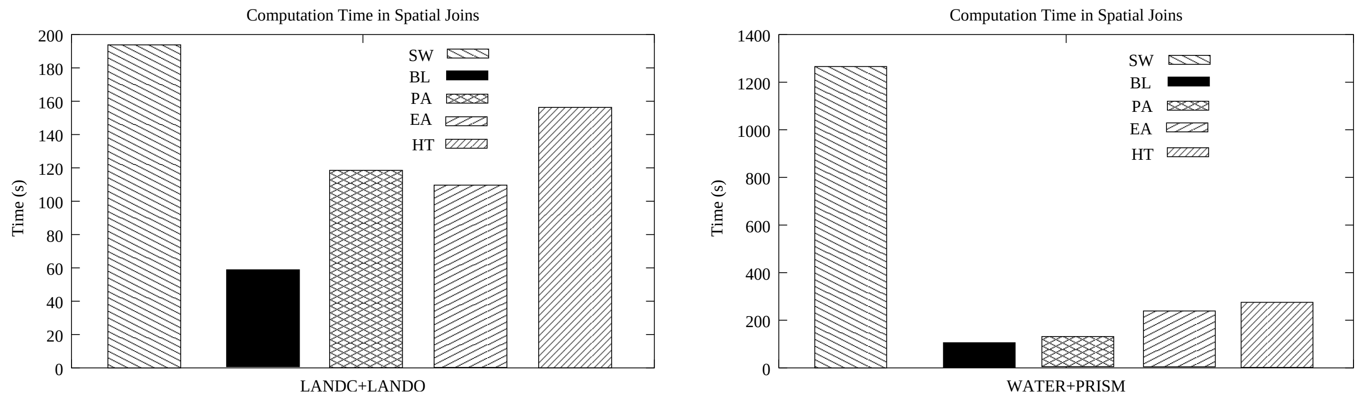
The average precision and recall of BL, PA and EA are shown in Table 2. HT is not included because both precision and recall of HT are provably 100%. As we expected, the PA algorithm does not produce any false hits, and the EA algorithm does not produce any false dismissals. In the worst case, the average recall of PA is above 92%, and the average precision of EA is above 96%. The accuracy of the BL algorithm is very close to the PA algorithm, except that the precision is slightly less than perfect.

5.3 Performance of Spatial Joins

In this section we evaluate the performance of the hardware-based algorithms for spatial join queries, and the results for the computational costs are shown in Figure 13. As in the spatial selection experiments, we use 16×16 rendering window in all hardware implementations.

From Figure 13, we see again that all the hardware algorithms outperform the software algorithm, however, the performance disparity varies significantly. For example, Figure 13(b) shows that in the WATER $\bowtie$ PRISM experiment, the PA algorithm outperform the software algorithm by almost a factor of 10. On the other hand, in the LANDC $\bowtie$ LANDO experiment, the performance improvement is only about 25%.

The difference in the performance improvements reflects the discussion in Section 3.1.1: the hardware



(a) LANDC+LANDO

(b) WATER+PRISM

Figure 13: Computation Time in Spatial Joins

algorithms are more suitable for the computation of more complex polygons. From Table 1, we note that in WATER and PRISM, there are polygons with about 30,000 or more vertices. For such polygons, the cost saving by the hardware approaches are more significant than for simpler polygons, such as the ones in LANDC and LANDO, which have at most 8,000 vertices.

Datasets	Alg.	False Hits	False Dismissals	Precision	Recall
LANDC+LANDO	BL	169	4461	99.81%	95.22%
LANDC+LANDO	PA	0	4501	100%	95.18%
LANDC+LANDO	EA	5563	0	94.37%	100%
WATER+PRISM	BL	10	1732	99.94%	91.26%
WATER+PRISM	PA	0	1753	100%	91.16%
WATER+PRISM	EA	820	0	96.03%	100%

Table 3: Join Errors

The accuracy results of the join experiments are similar to the ones in the selection experiments, as shown in Table 3. In the worst case, the recall of BL and PA is above 91%, and the precision of BL and EA is above 94%. Again HT is not included in this table because it guarantees 100% precision and recall.

6 Conclusion and Future Work

In this paper we proposed the use of computer graphics hardware to support spatial selections and joins for large complex spatial 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 proposed four hardware-accelerated solutions. In particular, the BL algorithm is 5 to 10 times faster than the software approach, with close to perfect precision and above 90% percent recall. Compared to the BL algorithm, the PA algorithm solves the false hit problem at a small cost of efficiency. The PA algorithm, though slower than both BL and PA, has two advantages. First, PA provides 100% recall, which is desirable for many database applications; and second, PA does not require the data to be pre-triangulated, therefore greatly reduces the storage costs. Finally, the most expensive HT algorithm guarantees full accuracy, and still significantly outperforms the software-only approach for complex data. In the near future, our work will be focused on further improving the performance of the hardware-based techniques, as well as exploring other application opportunities such as distance related queries.

References

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Ralf Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 40–49, 1993.

[BKSS94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 197–208, 1994.

[BWS99] George Baciu, Wingo Sai-Keung Wong, and Hanqiu Sun. Recode: An image-based collision detection algorithm. *The Journal of Visualization and Computer Animation*, 10(4):181–192, 1999.

[DT00] Chris Daly and George Taylor. *United States Average Annual Precipitation, 1961-1990*. Spatial Climate Analysis Service, Oregon State University; USDA - NRCS National Water and Climate Center, Portland, Oregon; USDA - NRCS National Cartography and Geospatial Center, Fort Worth, September 2000.

[DVOS00] Mark De Berg, Marc Van Kreveld, Mark Overmars, and O. Scharzkopf. *Computational Geometry: Algorithms and Applications*, chapter 2. Springer, 2000.

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57, 1984.

[HCK+99] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming C. Lin, and Dinesh Manocha. Fast computation of generalized voronoi diagrams using graphics hardware. In *Proceedings of the SIGGRAPH 1999 Annual Conference on Computer Graphics (SIGGRAPH'99)*, pages 277–286, 1999.

[HZLM01] Kenneth E. Hoff III, Andrew Zaferakis, Ming Lin, and Dinesh Manocha. Fast and simple 2d geometric proximity queries using graphics hardware. In *Proceedings of 2001 Symposium on Interactive 3D Graphics*, pages 145–148, 2001.

[KR01] Ravi K. Kothuri and Siva Ravada. Efficient processing of large spatial queries using interior approximation. In *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases (SSTD 2001)*, pages 404–421, 2001.

[RMS93] Jarek Rossignac, Abe Megahed, and Bengt-Olaf Schneider. Interactive inspection of solids: Cross-sections and interferences. In *Proceedings of the 19th Annual Conference on Computer Graphics (SIGGRAPH'92)*, pages 353–360, 1993.

[SA99] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., April 1999.

[Sur99] U.S. Geological Survey. *State Boundaries of the United States*. U.S. Geological Survey, November 1999.

[Wat01] John Watermolen. *1:2,000,000-Scale Hydrologic Unit Boundaries*. U.S. Geological Survey, 2001.

[WNDS99] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide, 3rd Edition*, chapter 10, pages 96–99, 226–229, 438–447. Addison-Wesley, 1999.

[Wyo96a] Wyoming Gap Analysis. *Land Cover for Wyoming*. University of Wyoming, Spatial Data and Visualization Center, December 1996.

[Wyo96b] Wyoming Gap Analysis. *Land Ownership and Management for Wyoming*. University of Wyoming, Spatial Data and Visualization Center, December 1996.