Using Graphics Processing in Spatial Indexing Algorithms

Mayuresh Kunjir & Aditya Manthramurthy

Abstract

In this project, we explore the use of graphics processing capabilities to speed up spatial indexing in spatial databases. The R-Tree is a data structure that is commonly used in databases to index spatial data. It can index and query 2D geometries by keying on the Minimum Bounding Rectangle (MBR) of each geometry. A query on this index is usually an input geometry, and the output is a set of intersecting geometries from the table. The MBR of the input geometry is used to search the R-Tree index and find the output set of overlapping MBRs. The geometries corresponding to the output MBRs obtained from the R-Tree are used to perform an actual geometry intersection test to determine the presence of intersections.

The properties of the R-Tree allow searching its different branches in parallel. We implement parallel R-Tree search using NVidia's CUDA GPGPU architecture. We also added a geometry intersection test in the PostGIS extension of the PostgreSQL database engine using the OpenGL framework. We tested the performance of these algorithms using real datasets. We also made an R-Tree visualizer to demonstrate the working of the R-Tree join.

1 Introduction

A spatial index, indexes geometric objects. The R-Tree [1] is a commonly used data structure that indexes geometric objects based on their Minimum Bounding Rectangle (MBR) - the smallest rectilinear rectangle that covers the geometry. A typical query

- gives a geometry and asks for all the geometries that overlap it - Intersection query; or
- gives a set of geometries (perhaps in the form of a separate spatial index), and asks for each pair of overlapping geometries - Join query.

The *intersection query* is evaluated using the MBR of the input geometry and then finding all overlapping MBRs using the R-Tree. The geometries in the MBRs returned by the tree are tested with the query geometry for an actual intersection. In searching the R-Tree, multiple branches may need to be traversed, and as each branch in independent of the others, this can be parallelized. We do this using NVidia's CUDA [5] General Purpose GPU processing architecture. Another area we identified for improvement is the geometry intersection test. For polygons with many edges, a full geometry test will involve testing for line

segment intersection of many pair of edges. By drawing these polygons in the off-screen framebuffer using blending, we are able to detect intersections using the OpenGL framework. We improve on the algorithm to do geometry intersections in hardware given in [4] and implement it in PostgreSQL database's PostGIS [6] extension. PostGIS provides data types, indexing and query operations for spatial entities.

A *join query* is evaluated by taking each geometry in one of the sets and performing the intersection query with it on the other set. We built a visualization tool that simulates the execution of a spatial join query and displays all the steps on the screen. We demonstrate with real datasets the execution of the join in the PostgreSQL engine.

Distribution of the work

Aditya formulated and implemented the R-Tree search algorithm using CUDA. We both wrote the core graphics intersection test using OpenGL and Mayuresh integrated the test in PostGIS. The R-Tree visualization tool was written by Mayuresh.

Overview

We begin with a description of the R-Tree enhancement using the graphics processor in section 2. After that we describe the visualization tool. In section 4, we describe the geometry intersection test implementation and some performance results. We summarize our work and mention possible future work in the last section.

2 R-Tree search using the Graphics Processor

The R-Tree is an important spatial indexing structure. An example is shown in the figure 1. The pink colored rectangles are the MBRs of the geometries being indexed. The rectangles drawn dashed are internal nodes of the R-Tree. The tree representation at the bottom shows how each node's MBR contains the MBRs of all its children.

The R-Tree [1] search algorithm can be parallelized, as shown in [3], but it is not a good algorithm because of its quadratic space requirement. The usual (un-parallelized) search algorithm starts searching from the root and proceeds recursively on each child node that intersects with the query MBR. We give a new linear-space search algorithm suited for the CUDA programming model.

2.1 Overview of the CUDA programming model

A *kernel* is a function that can be defined in the *C for CUDA* extension of the C programming language. These functions are executed N times in parallel by N different *CUDA threads*. Each thread has a thread index, that is unique within a thread block, and this can be used to map parts of the input/output that need to be processed by that thread. Each thread has private local memory. Threads are grouped into thread blocks, which run in parallel on a single core of the GPU and can share memory among threads of that block. Multiple thread

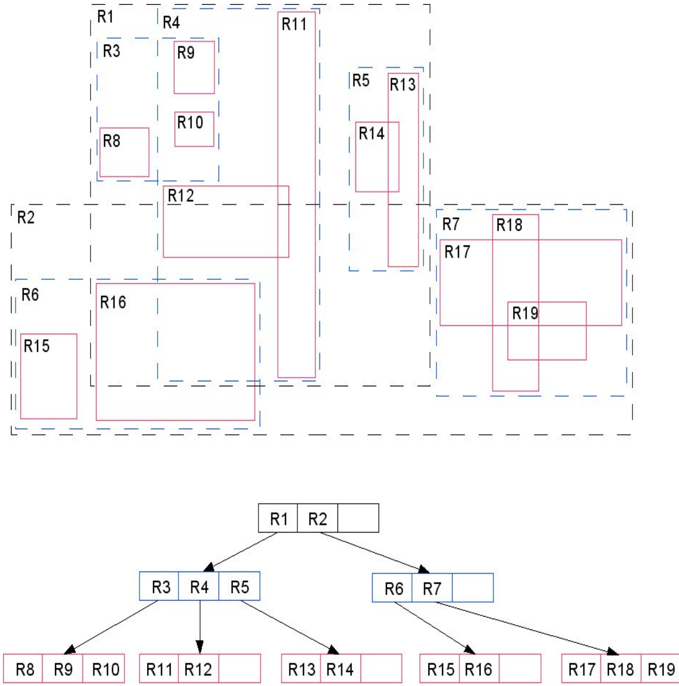


Figure 1: Example R-Tree

blocks may be running at a time on the GPU depending on the availability of cores on the GPU. All threads also have access to the same global memory. There is no locking facility in CUDA, so threads have to be careful about writing shared data.

Data for CUDA threads has to be copied from the host device (the CPU of the computer) using CUDA functions to device memory (the GPU). The results have to be copied back from the device memory after the GPU has completed execution. In our scenario, we use only one thread block to search the R-Tree.

2.2 R-Tree search algorithm on CUDA

We create two structures required for the R-Tree search algorithm on the CPU and then load it into the device memory. Whenever a search query arrives, we send it to the GPU and the threads execute the CUDA R-Tree search using the R-Tree data that has been initially copied. If the R-Tree changes, the structures for the CUDA R-Tree algorithm need to be copied once again. However we assume that this will not be often because spatial data does not change very frequently between queries.

The two structures that need to be stored in the GPU memory before hand are two arrays. The first is an array of MBR co-ordinates, which we call *Coord*. *Coord[i]* refers to the bottom-left and top-right co-ordinates of the i-th MBR in the index. The second is an array of structures called *Node*. *Node[i]* consists of (*mbrID*, *childNodes[t]*), and represents the R-Tree node with id i. The *mbrID* is an index into the *Coord* array that gives the co-ordinates of the MBR of that

node. *childNodes* is an array of size t, where t is the capacity of the tree. Each *childNode* element is an index into the *Node* array representing the children of the node i.

When the search query is made, a *kernel* call is made into the GPU. A thread block is launched. The threads in the thread block first copy the two structures into shared thread-block memory for efficiency. The threads next declare two more shared memory areas, for arrays of bits *currentSearch* and *nextSearch*. The length of each array equals N, the number of nodes in the R-Tree (which is less than the number of geometries indexed by the tree). These two bit-arrays are shared between all threads. Before executing the main loop, the bit for the root node is set in the *currentSearch* array.

Before we delve into the workings of the main loop, it is worth mentioning that only a constant number (say C) of threads can be running in a single thread block (current devices can run about 512). The arrays that are being shared above are going to be manipulated in parallel. Since CUDA has no locking mechanisms, we have to clearly separate which threads manipulate what. We do this using thread indexes. The thread with index i, only touches array locations $(N/C)k + i$ for each k. This should make sure that no thread accidentally corrupts shared memory.

The main loop executes the following instructions in order as long as *currentSearch* has a set bit. In the following, <SyncThreads> represents a CUDA synchronization primitive that will make each thread wait until all threads reach that point in the execution.

1. Clear the *nextSearch* array (in parallel). <SyncThreads>.
2. For each bit i belonging to this thread, if *currentSearch[i]* is set:
   (a) For each child node j (looked up from *childNodes*) that overlaps with the query MBR:
      i. If the child node is a leaf, mark it as part of the output.
      ii. If the child is not a leaf, mark it in the *nextSearch* array.
3. <SyncThreads>.
4. Copy *nextSearch* into *currentSearch* (in parallel). <SyncThreads>.

The output is copied to the CPU and returned in the application as the result of the query.

It is easy to see that the algorithm is a straightforward parallelization of the R-Tree search algorithm.

2.3 Implementation

The R-Tree base source code that we used for the implementation is at [9]. This source is intended for researching/testing R-Tree and related structures. We used data from [10]. We construct and copy the R-Tree structures Coord and Node after loading all the geometries into the tree. For each search query, a kernel call is made that uses the copied data to find the candidate geometries as described above. The copied data is persistent across multiple kernel calls, and persists till the application quits.

2.3.1 Difficulties in implementing the above algorithm in a real database

We initially proposed to implement the algorithm in the Oracle and PostgreSQL database systems. Modifying these databases involves re-implementing a part of the R-Tree code, namely the intersection search function. To the best of our knowledge Oracle Spatial does not export the R-Tree's search function implementation. Oracle Spatial sources are proprietary and not open. We turned to the open source PostgreSQL database to try to implement the algorithm, but there were a few difficulties that delayed us. PostgreSQL does not use a normal R-Tree to implement indexing. It uses a Generalized Search Tree (GiST) [2] to maintain indexes. By defining certain key methods for these trees, they can become B-Trees, R-Trees, etc. The search function is generic to all these types of trees. The search code in PostgreSQL in also complicated by the fact that each call to the function should return only one tuple, and so it saves the context of the execution internally. Because of this, implementing search in the GPU would be even more complicated, as we should return only one tuple on each call and perhaps maintain state in the GPU. Due to this increased complexity, we could not complete coding it in the database.

3 Visualization of Spatial Join

Normal join on the database compares every tuple of one relation with each tuple of other relation and based on some condition(like equality on attributes) decides whether to include the pair in the result. In the case of spatial joins, since we are working on the attributes representing some geometries(their location in particular); the join conditions can be area intersection or containment. For example, suppose you wanted to tell customers where they can find the nearest branch office of your business, or you want to compare different wildlife species with information about the habitats they live in. These types of queries can be answered with a spatial join. To handle such queries, we need to build spatial indexes and algorithms which operate on them to produce the result efficiently. In this section, we will first see how index can be used to calculate spatial join. We will then show the working of our visualization tool which shows the whole join process visually and then give the implementation details.

3.1 Spatial index for joins

The naive way to do spatial join would be considering all pairs of tuples of relation A and relation B and check for join condition. If join condition is finding geometries intersections, we will check intersection between all the pairs. The following block shows one such nested loop join plan generated from PostgreSQL server.

```
 temp=# explain select * from land, building where Intersects
(land.the_geom, building.the_geom);
                               QUERY PLAN
-----------------------------------------------------------------------
 Nested Loop  (cost=18.41..6077.01 rows=89380 width=1517)
   Join Filter: intersects(land.the_geom, building.the_geom)
   ->  Seq Scan on land  (cost=0.00..25.45 rows=545 width=887)
```

```
   ->  Materialize  (cost=18.41..23.33 rows=492 width=630)
         ->  Seq Scan on building  (cost=0.00..17.92 rows=492 width=630)
(5 rows)
```

Nested loops consume lot of time because of the processing of all the tuples. $R - tree$ index is used in such cases to improve the performance. The index is built by considering $Minimum\ Bounding\ Rectangles$ of the geometries. The internal nodes of R-tree consist of rectangles representing a collection of MBRs while each leaf represents MBR of a geometry. So while joining two relations, we check whether MBRs at root level intersect and only if they are intersecting, we descend to the next level. This saves a lot of intersection tests as in most of the cases, we find that geometries only intersect at a very small number places and we don't have to check geometries lying in other locations.

We are using PostgreSQL database engine and its spatial database extender $PostGIS$ which allows execution of spatial queries from PostgreSQL engine by adding some new functions and operators in PostgreSQL. Postgres uses a generalized index called $GiST$(Generalized Search Tree) [2] which provides all the basic search tree logic required by a database system, thereby unifying disparate structures such as B+-tree and R-tree in a single piece of code. PostGIS provides methods to make GiST behave as R-tree. An example query which uses the index is shown below.

```
 temp=# explain select * from land, hydrology where (land.the_geom
&& hydrology.the_geom);
                              QUERY PLAN
------------------------------------------------------------------------
 Nested Loop  (cost=0.00..30.16 rows=1 width=1559)
   ->  Seq Scan on hydrology  (cost=0.00..1.04 rows=4 width=672)
   ->  Index Scan using assets_land_idx_the_geom on land  (cost=0.00..7.27
rows=1 width=887)
         Index Cond: (land.the_geom && hydrology.the_geom)
(4 rows)
```

In this query, we are trying to find all the entries from relations *land* and *hydrology* whose MBRs intersect[1]. The query plan shows that index on *land* is used while *hydrology* is sequentially scanned.

3.2 Visualization tool

We have developed a visualization tool to see the working of R-tree index visually. The tool takes two relations whose join we want to find and the log generated by databases engine when the join was carried out as its input and then shows the step-by-step execution of join visually. In summary, following steps are carried out.

1. Get the contents of the two relations from database table in a file.
2. Execute the spatial join on database engine and get the generated *log*.
3. Parse the files and load all the geometries in a structure.

[1] && operator finds whether MBRs of its two arguments intersect or not

4. Draw all the geometries on the screen.
   Use different color for the two relations to distinguish them.
   Tessellate the polygons as they may be concave.

5. Start scanning the *log* file.
   Do this for all the entries of the *log*.
   If current entry is a R-tree node, highlight the MBRs it is representing. (The internal nodes and leaves are highlighted differently.)
   If current entry is the actual geometries which is checked when the corresponding MBRs intersect, highlight those geometries.

To show an example, we executed a join query on relations *land* and *hydrology* to find if there are any land encroachments happening on the area reserved for water canals. The query plan is shown below.

```
 temp=# explain select * from land, hydrology where ST_Intersects
(land.the_geom, hydrology.the_geom);
                              QUERY PLAN
-------------------------------------------------------------------
 Nested Loop  (cost=0.00..30.17 rows=1 width=1559)
   Join Filter: _st_intersects(land.the_geom, hydrology.the_geom)
   ->  Seq Scan on hydrology  (cost=0.00..1.04 rows=4 width=672)
   ->  Index Scan using assets_land_idx_the_geom on land
(cost=0.00..7.27 rows=1 width=887)
         Index Cond: (land.the_geom && hydrology.the_geom)
(5 rows)
```

This query scans each tuple of relation *hydrology* and uses index on *land* to first check whether MBRs intersect. The actual geometry intersection is carried out only if the MBRs are found to be intersecting. The generated log tells us the execution of all the tests. An example entry from log is given below:

GIST: rtree_leaf_consist called with strategy=3
0 257643.73 896902.56 257843.34 897113.25 252280 898880 254100 899740 0

The first line tells us that this is a leaf node entry of R-tree. Second line has the following format
<Sr. No.> <MBR of first geometry(x1 y1 x2 y2)> <MBR of second geometry(x1 y1 x2 y2)> <result of intersection (0:no intersection and 1:intersection)>

we use the log to highlight the geometries/MBRs which are processed currently. A snapshot of the tool in figure 2. The yellow colored polygons represent relation *land* while blue ones are from *hydrology*. The highlighted MBRs are the MBRs which are being considered currently and those which are found intersecting which will be later on checked for actual intersection. The hollow white rectangles show the internal R-tree nodes which are found intersecting with the current MBR of *hydrology*.

Some of the features of the visualization tools are:

- It can represent any type of polygon because of the use of tessellation.

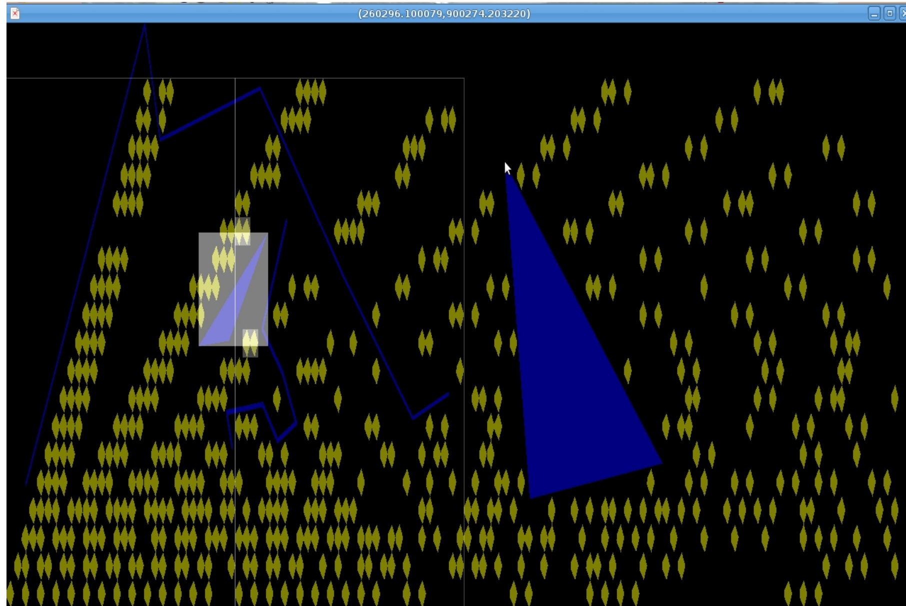


Figure 2: R-tree visualization tool

- There is provision to see the actual co-ordinates of any point shown on screen by clicking the particular point.
- You can zoom into the scene if you want to see the geometries in high resolution.
- The simulation of R-tree can be paused/resumed and also controls are provided to increase/decrease the speed.

3.3 Implementation details

The R-tree visualization tool is developed using standard OpenGL library. To get the inputs for this visualization tool, we made some changes to database engine. The geometry inputs which are the contents of the database table are easily generated by projecting the geometry column of the required relation and using resulting file to parse the geometry. We used the datasets provided by [10]. The important part is to generate the log. Though database engine maintains the log, it contains lot of other information like noting the entry in each function which is of no importance to our visualization tool. So we decided to create a separate log file for our purpose.

To build the log file, we had to put the necessary debug information from database engine. The PostGIS extension of the PostgreSQL database engine provides the functions to compare the index entry with the provided key(i.e. the geometry MBR) and also the functions to check the intersection of the geometries. To generate debug information of index processing, we had to understand about 1500 lines of code of PostGIS and add code to put the debug data to log file.

For the actual intersection test, the PostGIS carries out some short-circuit tests first and then uses a library called *GEOS*[7] which implements the OpenGIS [12] simple features for SQL spatial predicate functions and spatial operators to carry out the main intersection test. The details of this test are given in next section. To generate the debug information, we read about 10,000 lines of PostGIS code and about 1000 lines of GEOS code and added the code to write the data to log file.

4 Intersection using Graphics

As we saw in last section, a spatial join can have intersection test as the join condition and in the evaluation of the join, we need to carry out a lot of intersection tests. Since polygons can have arbitrary number of edges, it's difficult to find intersection between given two polygons in a non-exhaustive way all the time. We will discuss in the following subsections how PostGIS carries out intersections first and a simple graphics intersection test suggested in [4]. We implemented this test in PostGIS, the details of which and the results are given in next subsections.

4.1 Geometry intersection in PostGIS

There are three possible intersection queries possible in PostGIS viz.
a. Intersection between MBRs
b. Find if geometries intersect
c. Find intersecting points of two geometries.

We focus on second query which tells if two geometries intersect or not. PostGIS carries out MBR intersection test first to rule out some possibilities and then actual intersection test is carried out. The entire flow is shown in figure 3. PostGIS uses some GEOS library functions which are shown in red boxes while the functions provided by PostGIS are shown in blue boxes. As can be seen, many optimizations are carried out to avoid exhaustive algorithm which is to find intersection for all pairs of edges of two given geometries. So for most of the input tuples, short-circuit tests give the result and we don't have to do exhaustive test thus saving time.

4.2 Graphics intersection test

The paper [4] talks about a simple intersection test using graphics where two polygons are drawn on screen with some color with color blending enabled. If the two polygons intersect at some point, that point will have the color which is the addition of color components of two polygons. We read whole pixel buffer and look for this color value and if found one, conclude that geometries intersect. An example is shown in figure 4 which shows two intersecting rectangles. The overlapping region shows white color while geometries are drawn with grey color. The paper suggests to put this test after MBR intersection test and before actual software intersection test.

Though the test looks simple, there are some issues in deciding resolution of the image to ensure that we don't miss out on some intersections. To give an example, consider that the two polygons range from $(0, 0)$ to $(10^4, 10^4)$ and the

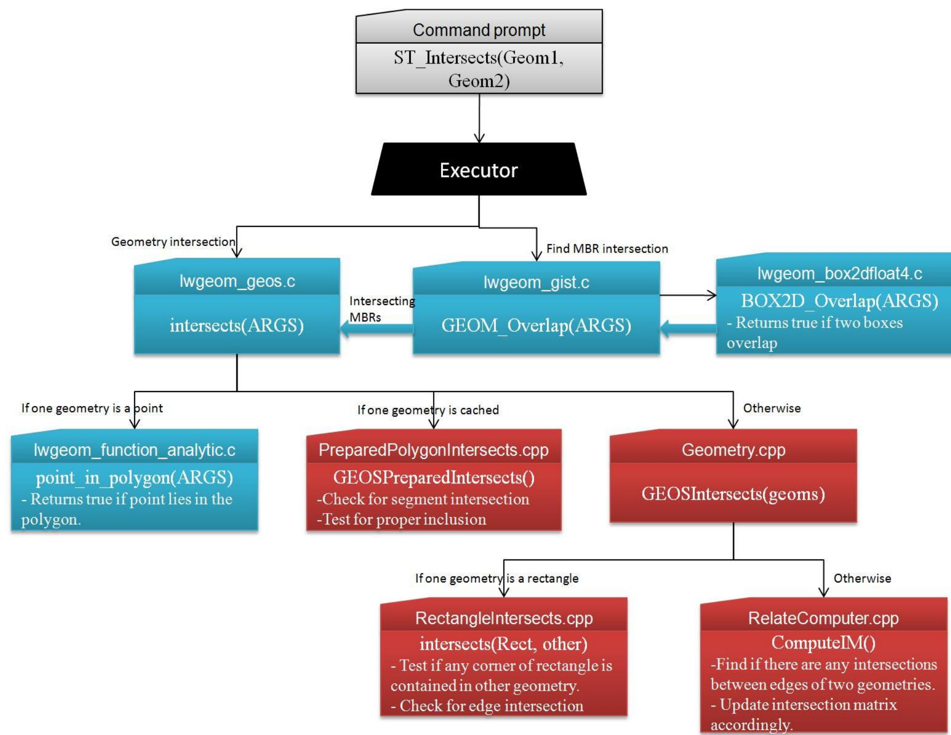


Figure 3: Geometry intersection flow diagram

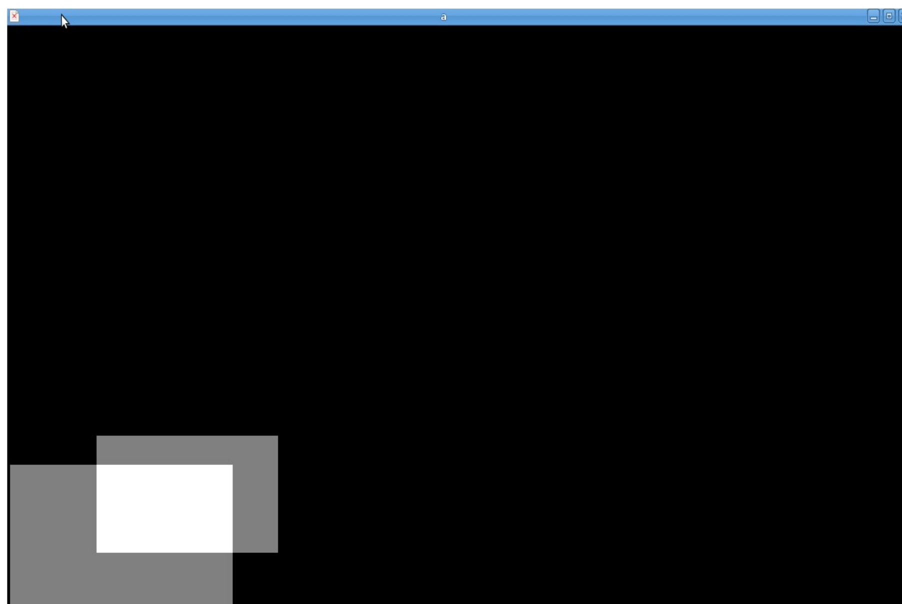
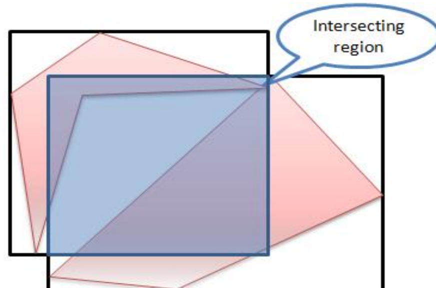


Figure 4: Graphics intersection test

intersecting part is only of area 10^{-3}, you probably won't see the intersecting region rendered on the screen. We suggest a small improvement in this part. Since we only get geometries for this test only after finding that their MBRs overlap, we are sure that bounding rectangles overlap in some part. So we can render only this overlapping part on screen because if the geometries are intersecting they can intersect only in this part and other part of geometries is of no interest to us. We can find almost all the intersections with this improvement. In one of our test queries where MBRs of one geometry were larger by the order of 4 approximately than MBRs of other geometry, we were only able to find 7 out of 124 intersections with earlier test. But with this improvement, we are able to find all 124 intersections.

4.3 A problem

Though we improve on the suggested algorithm by only rendering MBR overlap region, we may not still find intersection in some cases. These are the cases where polygons are concave and have very high MBR overlap but very small actual intersecting region. An example is shown in figure below. But these cases are very rare and we haven't come across such situation on the datasets we have worked upon. In most of the cases, most of the MBR overlap region contains the geometries and thus our test goes through. But to be on safer side, we follow this test with software intersection tests provided by GEOS library.



4.4 Implementation details

To implement graphics intersection test, we have to do off-screen rendering of the polygons. Off-screen rendering is not a core part of OpenGL. So we use the off-screen rendering provided by *GLX* window system interface. We use *pbuffers* which allow hardware accelerated rendering to an off-screen buffer [8]. We first need to establish a connection to X server to use pbuffers. We do this when we establish the connection to PostgreSQL server. The code for which is inserted in *PostmasterMain*() routine which is the main entry point of the database server. The resource freeing is done in the routine *ExitPostmaster*() which is called when the connection to server is terminated. The PostgreSQL database engine is linked with graphics libraries to use these graphics routines.

The main intersection test is implemented in PostGIS. After MBR intersection test is carried out, the routine *intersects*() of file *lwgeom_geos.c* is called as shown in figure 3 which in turn calls some routines of GEOS library to carry

	Q1(time in ms)	Q2(time in ms)
without graphics test	31	44
pbuffer 500×500	30496	3264
pbuffer 200×200	4910	577
pbuffer 100×100	1261	189

Table 1: Comparison of performance of join with/without graphics test and the effect of changing pbuffer size

out actual intersection. We put our intersection test in between these two tests. We draw the two geometries on the window with color $(0.5, 0.5, 0.5)$, the context for the window was set initially in $PostmasterMain()$. The blending is enabled so that the colors get added in overlapping region. Remember, we only draw overlapping regions of the MBRs as discussed in previous subsection. So we map this overlapping region to the size of the window. After drawing these geometries, we read the $pixel\ buffer$. While this buffer is read from GPU memory to main memory, the minimum and maximum color values of the buffer are found out in hardware which we read by using function $glGetMinmax()$ function. Here we only read *red* component from pixel buffer to reduce the transfer. We check whether maximum value is 1. If found, we declare that intersection is found.

4.5 Results

Our goal of doing this exercise of putting graphics intersection test was to enhance the performance of the spatial join queries. So we tried to execute some queries with this test enabled and without the test and compare the results. We used a dataset depicting hydrological boundaries of U.S. [11]. We found that graphics intersection test is very slow compared to software intersection tests and the main bottleneck is reading of the frame buffer for each intersection test which involves transfer of the data from GPU memory to main memory. As can be seen from results given below in table 1, when we use pixel buffer of size 500×500 we get degradation as much as of the order 3. To verify that pixel buffer is the bottleneck, we carried out some experiments by changing pixel buffer size. As can be seen in table, as we reduce the size of pixel buffer, the processing gets faster. But reducing pixel buffer also means that we are compromising on the accuracy of the algorithm as we are reducing the screen space to draw the geometries. These results show that there is no use of adding the graphics intersection test unless we use some faster ways of GPU processing.

5 Conclusion and future work

In our work, we implemented R-Tree search algorithm using GPGPU. The algorithm uses linear space as opposed to quadratic space used by [3]. A visualization tool is developed to show the working of R-Tree in join processing. We also implemented a simple hardware intersection test in PostgreSQL database which is an improvement over the one suggested in [4]. The performance results of this test show that because of the bottleneck of the slow transfer of data from

GPU memory to main memory, we get huge degradation. The test is written for only specific join queries which test whether given two geometries intersect or not. It can be extended to some other queries like containment easily. To reduce the overhead of data transfer from GPU to main memory, we can use CUDA functions to parallelize the function to find maximum color value from pixel buffer. Also in future, we would like to implement R-Tree search algorithm using CUDA in PostgreSQL database engine.

References

[1] A. Guttman, "Rtrees: a dynamic index structure for spatial searching", in Proceedings of ACM SIGMOD International Conference on Management of Data, pp. 47-57, 1984.

[2] Joseph M. Hellerstein , Jeffrey F. Naughton , Avi Pfeffer, Generalized Search Trees for Database Systems, Proceedings of the 21th International Conference on Very Large Data Bases, p.562-573, September 11-15, 1995.

[3] Xiao X., Shi T., Vaidya P., Lee J., R-Tree: A Hardware Implementation, In 2008, 3-9. Proc. of International Conf. on Computer Design.

[4] Chengyu Sun, Divyakant Agrawal, Amr El Abbadi, Hardware Acceleration for Spatial Selection and Join, In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data.

[5] CUDA home `http://www.nvidia.com/object/cuda_home.html`

[6] PostGIS home `http://postgis.refractions.net/`

[7] GEOS library `http://trac.osgeo.org/geos/`

[8] Off-screen rendering `http://www.mesa3d.org/brianp/sig97/offscrn.htm`

[9] R-Tree base source code `http://www2.research.att.com/~marioh/spatialindex/`

[10] PGCon 2009 `http://www.pgcon.org/2009/schedule/events/174.en.html`

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

[12] OpenGIS standard `http://www.opengeospatial.org/standards/sfs`