Dynamic Binary Space Partitioning for Hidden Surface Removal

Vivek Kwatra, Alok Kumar, Balpreet Singh, Sanjiv Kapoor
Department of Computer Science & Engineering
Indian Institute of Technology
New Delhi

Abstract

This paper describes a new technique for dynamic binary space partitioning with applications to the dynamic hidden surface removal problem of polygons. The dynamic hidden surface removal problem is to determine and maintain visible surfaces in 3-D set of polygons from a given viewpoint under changes to the scene and it allows for efficient insertions and deletions of polygons in a 3-D scene. The polygons are maintained in a space partitioning structure called the separating plane partition tree which is a variation of the traditional binary space partition (BSP) tree. In this structure , separating planes are the fundamental objects which are used to partition the space. The polygons are fragmented with each of the fragments being stored at the leaves. This allows for more easy access of polygon fragments for insertion and deletion. Efficient hidden surface removal of the polygons is then achieved by painting the fragments in an order obtained from the space partitioning tree.

1 Introduction

We consider the problem of dynamic hidden surface removal problem (DHSR). Given a set of polygons, $P_1, P_2 \ldots P_n$ in 3-D and a viewpoint v the DHSR problem is to determine and maintain the surfaces viewable from v under the operations -

- Insertion of a polygon
- Deletion of a polygon
- Change of viewpoint.

Applications of the dynamic hidden surface removal include simulating walk-through's and modeling of moving structures [1,10], Virtual Reality environments, assistance in design of architectural structures and solid modeling.

We solve the dynamic hidden surface removal problem using a variation of the traditional Binary Space Partition (BSP) Trees. The BSP-tree is a versatile structure commonly used for hidden surface removal and solid modeling [4,5,8]. Other uses of the BSP-Trees are in shadow generation, collision detection, implementing set operations on objects etc [3,4,5,6].

A BSP tree is a binary tree used to partition n-dimensional space and essentially to sort polytopes. The tree taken as a whole represents the entire space, and each subtree represents a convex subspace. Each node stores a **hyperplane** which divides the space it represents into two parts, and has as its two children, nodes which represent each part. In addition, each node may store one or more polytopes. A **hyperplane** in n-dimensional space is an n-1 dimensional object which can be used to divide the space into two half-spaces. For example, in three dimensional space, the **hyperplane** is a plane. In two dimensional space, a line is used. In this paper we will restrict ourselves to 3-D space though our arguments will carry over to higher dimensions also.

In 3-D, the polygons representing the scene are stored at the various nodes of the BSP tree. Consequently, the BSP-Tree creates fragments of inserted polygons. Using the BSP-tree directly for dynamic operations on a set of polyhedral objects creates complications since the fragments of a polygon are distributed all over the tree. Nevertheless, this static structure can be extended for use in a dynamic environment, as described in [2]. In [2], where the traditional static BSP structure is used, the deletion of a polygon causes regeneration of entire subtrees occuring below the fragments of the affected polygon, increasing the complexity. It may be useful when only a single polygon undergoes many successive insertions and deletions, which may not happen in the general case.

Hence, for an efficient implementation of dynamic operations our aim was to force the fragments of the polygons to appear at the leaves of our space partitioning tree. This allows for easy access to all fragments of a polygon and its removal. Note that removal of fragments from an internal node of the BSP-Tree would involve restructuring of the tree. We modify the traditional BSP-tree by introducing a separating plane as a partitioning plane. A separating plane is a plane that separates two objects, which in our case are two polygons. This separating plane is stored at the internal nodes of the tree and is not part of the set of input polygons. It is used simply to partition the space. The same effect may be achieved simply by constructing an oct-tree: however the separating plane allows two separable polygons to be inserted *without* any fragmentation whatsoever. Separating planes have also been used by Torres [10]. But as [2] notes, the approach uses separating planes between compound objects using 6-level tree structures which becomes very complex and loses elegance. Our approach uses separating planes at the lowest level in the object structure.

In the next section we detail the separating plane partitioning method. In the last section some results of the implementation of the above approach are described. This work may find extensions in providing shading and shadows to 3-d views, which will make view much more realistic and to achieve hidden surface removal for curved surface.

2 The Separating Plane Algorithm

In a dynamic environment since we are concerned with the problem of relative motion, the configuration of the polygons would keep changing in space continuously. Therefore, we need a data structure which is able to provide an efficient means of updating it with the change in the position of one of its elements, namely a polygon in our case. The previous data structure, i.e. the traditional BSP, in which we use the plane containing the polygon to divide the space into two halves is not very useful for a dynamic environment because when a polygon is inserted the plane containing the polygon is used to divide the space into smaller regions. Hence, when a polygon shifts, the entire BSP tree needs to be regenerated, and the overhead of doing such a thing is very large. So we need to use a data structure which need not be regenerated every time a polygon moves. The BSP tree with a separating plane as described below is such a data structure.

2.1 Data Structure for Partition Tree

The data structure used to store the polygons in our algorithm is a binary tree **B** which essentially represents the BSP tree we are discussing. It has the following properties :-

- All the nodes of **B** either contain planes or polygons.
- Only leaf nodes of **B** contain polygons.
- All other nodes of **B** contain planes which partition the space.
- Any node **T** has a particular region, say **R** in space associated with it.
- If **T** contains a plane, then this plane again divides **R** into two regions, with which the left and the right child of **T** are associated.
- If **T** contains a polygon then **T** should be a leaf node and the polygon contained in **T** should lie completely within the region associated with **T** and no other polygon should lie in this region.
- The root of **B** node is associated with the entire 3-D space.

2.2 Insertion :

Now we describe how to insert a polygon **P** into a given BSP tree **B**.The polygon has to lie at the leaf node(s) associated with the appropriate region(s) in which the polygon lies. If we can find a region in which the entire polygon lies, we plac e it there, or else it is fragmented into smaller polygons which can fit into single regions. The algorithm goes as follows :-

Let **T** be the root node of **B**. **T** would either contain a plane or a polygon(if it is a leaf node as well).The polygon **P** lies completely within the region associated with **T**.Consider the following cases:-

1. **T** contains a plane **L**. Then **T** is not a leaf node and has a left child node $\mathbf{T_1}$ (region left of **L** and a right child node $\mathbf{T_2}$ (region right of **L**).
   (a) If **L** fragments **P** into $\mathbf{P_1}$ and $\mathbf{P_2}$ with $\mathbf{P_1}$ lying in the region associated with $\mathbf{T_1}$ and $\mathbf{P_2}$ lying in the region associated with $\mathbf{T_2}$, then we insert $\mathbf{P_1}$ into the tree with $\mathbf{T_1}$ as root and $\mathbf{P_2}$ into the tree with $\mathbf{T_2}$ as root node.
   (b) If **P** lies completely to the left of **L**, we insert **P** into the tree with $\mathbf{T_1}$ as root node, else if **P** lies completely to the right of **L**, we insert **P** into the tree with $\mathbf{T_2}$ as root node.
2. **T** contains a polygon **P'** and is a leaf node. Now we need to find a separating plane between the two polygons **P** and **P'**. Consider the following cases:-

(a) **Case 1:** **P** *and* **P'** *do not intersect.* Note that two polygons having a common edge, or two polygons with an edge of one polygon lying in the plane of the other do not intersect explicitly but they would not be considered by us in this category.Here, we find a plane **L** which separates the region associated with **T** into two regions, which completely contain **P** and **P'**.We replace **T** with a node containing the plane **L** and having two children as nodes containing **P** and **P'**, with the polygon to the left of **L** as the left child and the one to the right of **L** as the right child. The algorithm for finding **L** is described at the end of this insertion algorithm.

(b) **Case 2:** **P** *and* **P'** *intersect.* Here, we find two planes $\mathbf{L_1}$ and $\mathbf{L_2}$ which are the bisectors of the angles between the planes containing the two polygons. Note that these planes pass through the line of intersecti on of the two polygons. Now the region associated with **T** is divided into four regions, each containing one or less of fragments of **P** and **P'**. Thus, we are able to find regions which contain exclusively one polygon (a fragment of **P** o r **P'**), and hence we can update **B** as below:

If **P** *and* **P'** *both fragment each other*, generating a total of four polygons, then we replace the node **T** with a node containing $\mathbf{L_1}$. The left and right child of this node, $\mathbf{T_1}$ and $\mathbf{T_2}$ both contain the plane $\mathbf{L_2}$. The four fragments formed above are placed as the children of $\mathbf{T_1}$ and $\mathbf{T_2}$ as follows:-

The children of $\mathbf{T_1}$ are fragments which lie on the left of $\mathbf{L_1}$ and on the left and right of $\mathbf{L_2}$ appropriately.

The children of $\mathbf{T_2}$ are fragments which lie on the right of $\mathbf{L_1}$ and on the left and right of $\mathbf{L_2}$ appropriately.

If the line of intersection of **P** *and* **P'** *is a common edge*, then one of the bisecting planes would divide the region associated with **T** into two regions such that **P** lies in one such region (say, the left one) and **P'** lies in the other one (the right one). If $\mathbf{L_1}$ is this plane then we replace **T** with a node containing the plane $\mathbf{L_1}$ which has nodes containing**P** and **P'**, as its left and right child respectively.

If the line of intersection of one of the polygons lies in the plane of the other, say **P'** has an edge in the plane of **P**, which is not an edge of **P** of-course, then only three of the four regions formed by $\mathbf{L_1}$ and $\mathbf{L_2}$ are occupied because **P** fragments into two, occupying two regions and **P'** occupies a third one. We replace **T** with a node containing $\mathbf{L_1}$. We have the following scenario:-

P breaks into two fragments, $\mathbf{P_1}$ and $\mathbf{P_2}$. Now, $\mathbf{P_1}$ and **P'** lie on one side of $\mathbf{L_1}$ and $\mathbf{P_2}$ lies on the other side of $\mathbf{L_1}$. If $\mathbf{P_2}$ lies to the left of $\mathbf{L_1}$, it becomes the left child of **T** else its right child and the other child is a node containing $\mathbf{L_2}$. Now, $\mathbf{L_2}$ divides this region into two, with **P'** and $\mathbf{P_1}$ in each one, which accordingly become the left and right child of this node containing $\underline{f}L_2$.

The above algorithm described for inserting a polygon into **B** checks at each node, to which side of the plane does it lie, and then moves to the corresponding child. At a leaf node it does some computation for the separating plane and adds the polygo n appropriately.

Thus, if **B** contains n nodes then the time taken for insertion is O(d) where d is the depth of the tree, which we believe to be O(log n) on an average if the polygons are inserted in an order such that they lie on either side of the existing separating planes with an almost equal probability.

2.3 Algorithm to Determine the Separating Plane :

A very important part of the above algorithm is the determination of a separating plane between two polygons, say $\mathbf{P_1}$ and $\mathbf{P_2}$. Let $\mathbf{L_1}$ and $\mathbf{L_2}$ be the planes containing these polygons.

1. In case of the polygons having a common line of intersection (i.e. intersecting polygons), we need two planes which are simply the bisector planes of $\mathbf{L_1}$ and $\mathbf{L_2}$ and are obtained by rotating either $\mathbf{L_1}$ or $\mathbf{L_2}$ by half the angle between them on either side.

2. In case of the polygons having no common line of intersection (non-intersecting polygons), we follow the algorithm described below for the respective cases (the separating planes are shown in fig **2**) :-

   (a) Neither of $\mathbf{L_1}$ or $\mathbf{L_2}$ intersects $\mathbf{P_2}$ or $\mathbf{P_1}$ respectively. In this case the line of intersection of $\mathbf{L_1}$ and $\mathbf{L_2}$ does not intersect

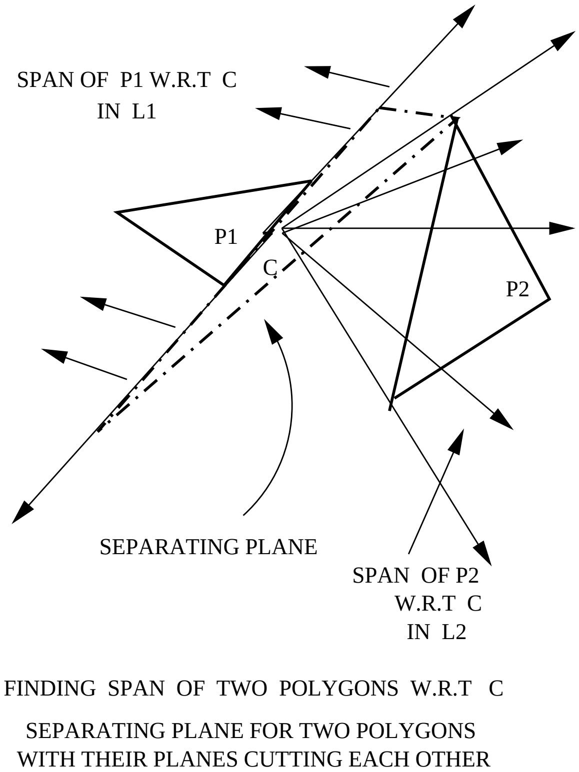

Figure 1: **Spans a polygon wrt a point**

either of $\mathbf{P_2}$ or $\mathbf{P_1}$ and the separating plane is obtain ed by just rotating one of $\mathbf{L_1}$ or $\mathbf{L_2}$ by half the angle between them about this line of intersection.

(b) Only one of $\mathbf{L_1}$ or $\mathbf{L_2}$ intersects $\mathbf{P_2}$ or $\mathbf{P_1}$ respectively, say $\mathbf{L_1}$ intersects $\mathbf{P_2}$. In this case, we just shift $\mathbf{L_2}$, by half the shortest distance between $\mathbf{L_2}$ and $\mathbf{P_1}$, towards $\mathbf{P_1}$, to obtain the separating plane.

We can immediately see in the above two cases that the plane obtained actually separates the two polygons.

(c) Both $\mathbf{L_1}$ and $\mathbf{L_2}$ intersect $\mathbf{P_2}$ and $\mathbf{P_1}$ respectively. In this case we find the edge of $\mathbf{P_1}$ which is cut by $\mathbf{L_2}$ and which faces $\mathbf{P_2}$. Keeping this edge as hinge, we form planes with the third point as one of the vertices of $\mathbf{P_2}$ and test if this plane forms a separating plane.

Claim : At least one of these planes forms a separating plane.

Proof : Consider a point $\mathbf{C}$ in space. We define a *span* of a polygon w.r.t $\mathbf{C}$ as the region in space enclosed by drawing rays from $\mathbf{C}$ to the various vertices of the polygon. Now if the span of two polygons do not intersect then we can find a plane which separates the two polygons and passes through $\mathbf{C}$.

For the above case, let $\mathbf{C}$ be defined as the point of intersection of $\mathbf{L_2}$ with $\mathbf{P_1}$,lying on the edge of $\mathbf{P_1}$ which intersects $\mathbf{L_2}$ and is closest to $\mathbf{P_2}$. Now the span of $\mathbf{P_1}$ w.r.t $\mathbf{C}$ is the entire half plane containing $\mathbf{P_1}$ and not intersecting $\mathbf{P_2}$ because $\mathbf{C}$ lies on the edge facing $\mathbf{P_2}$. The spa n of $\mathbf{P_2}$ w.r.t $\mathbf{C}$ is the angle between the most diverging vertices of $\mathbf{P_2}$ and $\mathbf{C}$, lying in the plane $\mathbf{L_2}$, and away from the span of $\mathbf{P_1}$. Since the two spans do not intersect, therefore there exists a separating pl ane passing through $\mathbf{C}$. Any separating plane passing through $\mathbf{C}$ would be hinged at the edge containing $\mathbf{C}$ for not intersecting $\mathbf{P_1}$. If we keep rotating this plane, we would finally touch one of the vertices of $\mathbf{P_2}$. Thus, our claim above for the existence of a plane as defined is true.

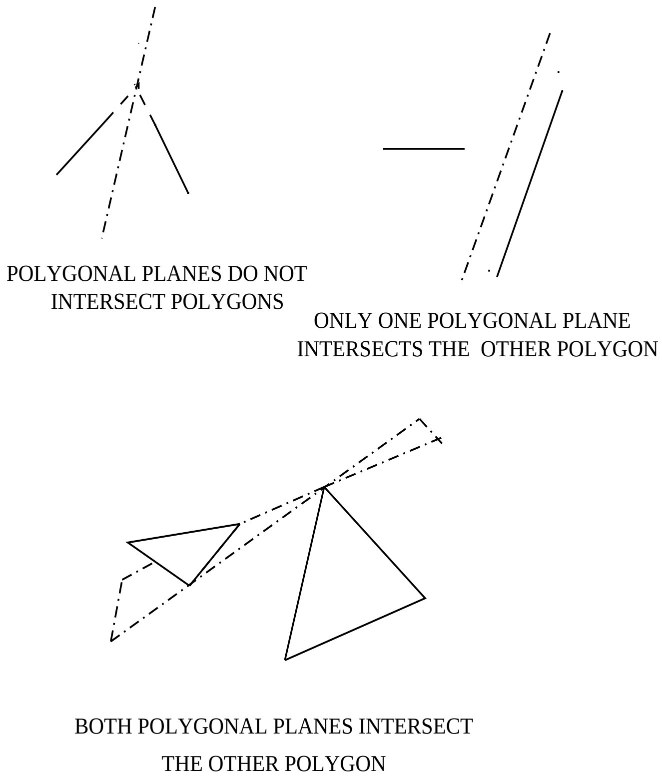

Figure 2: **Separating planes for non-intersecting polygons**

2.4 Deletion of a Polygon :

Since a polygon aimed for deletion may be fragmented while insertion, we maintain a thread of fragments of the same polygon within the BSP.

To delete a polygon

- move to the first fragment of the polygon
- delete the node containing it
- replace the parent of this node by the node containing the sibling of this node
- repeat this for all fragments of the polygon

Note that since the parent of this node contains a separating plane between it and its sibling we do not require the plane any more after the deletion of this node. Hence we just leave the sibling.

The time for deleting a fragment is O(1) and for deleting an entire polygon the time taken is O(f), where f is the no. of fragments of the polygon

2.5 Simulating motion :

The algorithm which uses this BSP tree to simulate motion of a polygon would do so by deleting the polygon from the tree and then inserting a polygon corresponding to the new position of the moved polygon.Here, the separating planes in the tree would almo st remain the same as computed initially, except for the ones which are the immediate parents of the fragments of the moving polygon.

In the BSP with the plane containing the polygon as the separating plane, such an approach would imply that at each deletion, we regenerate the entire sub-tree lying below the separating plane formed by the moving polygon, and do so for all fragments of the polygon. In our BSP, since all polygons lie at the leaf nodes, deleting a fragment becomes a constant time operation, and is thus much more efficient.

3 Conclusion

The algorithm was implemented by us and we experimented on it by testing the execution time obtained by varying various parameters. The graphs which were obtained are shown in this section.

The graph showing **Insertion time vs No of polygons inserted** (fig **4**) shows the characteristic of order $o(n^2)$. The graph of **No of fragments vs No of polygons** (fig **3**) again shows that the order of number of fragments is of $o(n*(log(n))^2)$. **Deletion** takes linear time (fig **5**), which has a great signifigance in scoring over the scheme in [2]. Comparison with that scheme of updating the traditional BSP tree when a polygon is deleted was not explicitly experimentally performed, but since the update there would be equivalent to reinserting, the tme complexity of that would be extremely large. One possible scheme in that framework would be to delete fragments but mark nodes as dirty. This would increase the space complexity. Moreover, periodically, large costs would be incurred to reconstruct the tree.

References

[1] Arya, K., Processes in a Functional Animation System, IBM T.J. Watson Reseach Center Yorktown Heights, NY 10598.

[2] Chrysanthou, Y., and Slater, M., Computing dynamic changes to BSP trees, Computer Graphics Forum (EUROGRAPHICS '92 Proceedings), 11(3), 321–332, sep 1992.

[3] Naylor, B., Amanatides, J., and Thibault, W., Merging BSP Trees Yields Polyhedral Set Operations, Computer Graphics (SIGGRAPH '90 Proceedings), 24(4), 115–124, aug 1990.

[4] Naylor, B., Interactive solid geometry via partitioning trees, Proceedings of Graphics Interface '92, 11–18, may 1992.

[5] Naylor, B., Partitioning tree image representation and generation from 3D geometric models, Proceedings of Graphics Interface '92, 201–212, may 1992.

[6] Naylor, B., SCULPT An Interactive Solid Modeling Tool, Proceedings of Graphics Interface '90, 138–148, may 1990.

[7] Ihm, I., and Naylor, B., Piecewise linear approximations of digitized space curves with applications, Scientific Visualization of Physical Phenomena (Proceedings of CG International '91), 545–569, 1991.

[8] Paterson, M., and Yao, F., Efficient Binary Space Partitions for Hidden-Surface Removal and Solid Modeling, Discrete and Computational Geometry, 5(5), 485–503, 1990.

[9] Torres, E., Optimization of the Binary Space Partition Algorithm (BSP) for Visualization of Dynamic scenes. Computer Graphics Forum(EUROGRAPHICS '90 proceedings), 9(3):507-518, 1990.

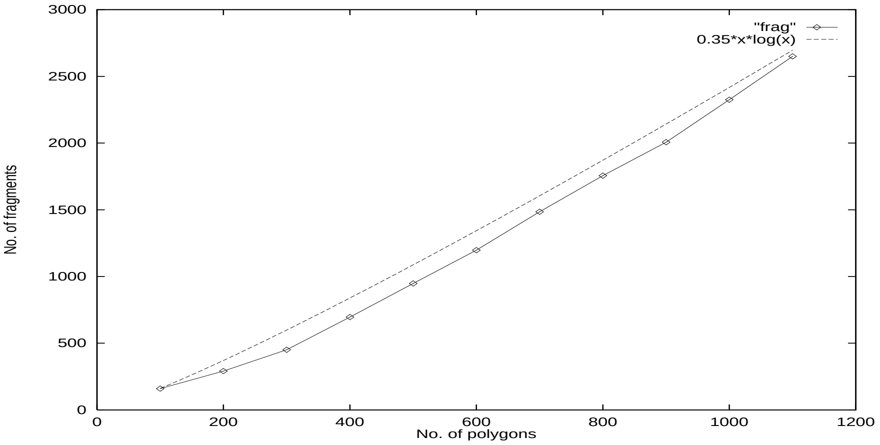

Figure 3: **No of Fragments vs No of Polygons inserted in the tree**

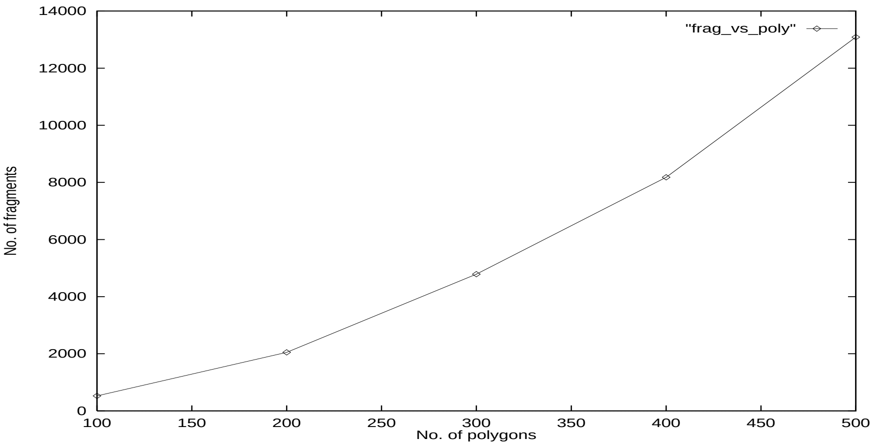

Figure 4: **Time taken for insertion vs No of polygons inserted**

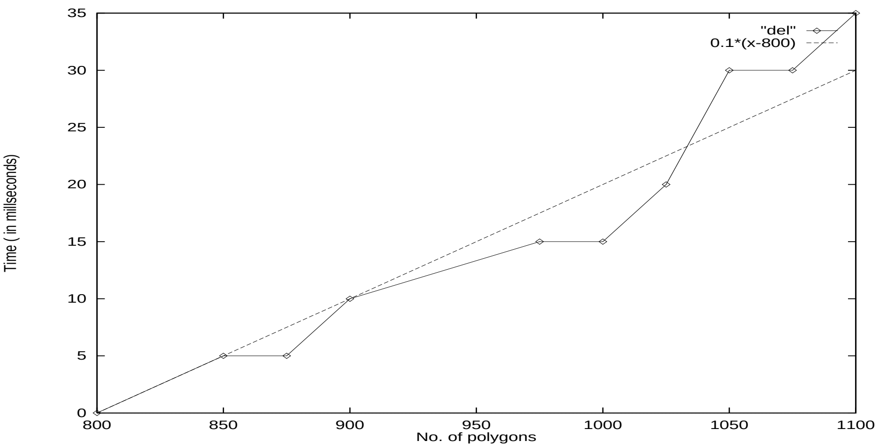

Figure 5: **Time taken for deletion vs No of polygons deleted**