Visibility Queries among Horizontal Segments — A Dynamic Data Structure

Sergei Bespamyatnikh[1] Matthew J. Katz[2] Frank Nielsen[3] Michael Segal[4]

[1]Department of Computer Science, University of British Columbia
[2]Department of Computer Science, Ben-Gurion University
[3]SONY Computer Science Laboratories Inc., FRL
[4]Department of Communication Systems Engineering, Ben-Gurion University

1 Introduction

Consider the following interactive computer game. A set of blue horizontal segments is drawn in a window. The player must draw in the window, *within a few seconds*, a red segment that is not visible from the bottom edge nor the top edge of the window's boundary. If the red segment drawn by the player is visible, then the player does not get any points for this move; otherwise, the number of points is proportional to the length of the segment. After computing the score for this move, the red segment is removed, and either a new blue segment is added to the scene or an existing blue segment is deleted from the scene. The player must now draw again a red segment for the new scene, get points for this move, and so on.

The main algorithmic difficulty in the implementation of this game, is to determine at each round whether or not the red segment drawn by the player is visible from one of the horizontal edges of the window's boundary. We propose a data structure that overcomes this difficulty. More precisely, we consider the following problem.

Problem: Maintain a set $\mathcal{S}$ of horizontal segments in the plane under insertions and deletions, so that when given a query horizontal segment l, one can efficiently determine whether l is visible from the x-axis, in the sense that one can draw a vertical segment whose endpoints lie on the x-axis and on l, respectively, which does not intersect any of the segments in $\mathcal{S}$.

Our data structure for this problem is quite simple but new and neat. It is based on a segment tree [2]. Its size is $O(n \log n)$, where n is the current size of $\mathcal{S}$, and it supports logarithmic-time queries. An update, i.e., insertion or deletion, costs $O(\log^2 n)$.

2 The data structure

We maintain a segment tree $\mathcal{T}$ for the current set of segments $\mathcal{S}$ (actually, for the projections of the segments in $\mathcal{S}$ on the x-axis). Consider a node v of $\mathcal{T}$. Let $\mathcal{S}_v$ be the subset of $\mathcal{S}$ that is associated with v; $\mathcal{S}_v$ is called the canonical subset of v. The segments in $\mathcal{S}_v$ are stored in the nodes of a binary search tree $\mathcal{T}_v$, according to their y-coordinates.

In addition, we store with v some information concerning the segments that are stored at the descendants of v. Let r_v be the horizontal interval associated with v; r_v is called the canonical interval of v. We store with v a boolean variable c_v whose value is true if and only if the union of the projections of the segments that are stored at the descendants of v completely covers r_v.

We also store with v the y-coordinate y_v of the highest segment among the segments stored at the descendants of v that is visible from the x-axis (ignoring all other segments).

The size of the data structure is $O(n \log n)$, where n is the current size of $\mathcal{S}$. We next describe the query algorithm, and then the algorithms for insertion and deletion.

Query. Let l be a query horizontal segment, for which we wish to determine whether it is visible from the x-axis. As usual, we first partition l into $O(\log n)$ pieces corresponding to canonical intervals. We shall treat each of the pieces separately; if one of them is visible then l is visible, otherwise l is not visible.

Let l_v be a piece corresponding to the canonical interval of node v. If one of the segments that is stored in v is below l_v, then clearly l_v is not visible. We check therefore whether l_v is below the lowest segment in $\mathcal{T}_v$. We repeat this check at each of the $O(\log n)$ ancestors of v.

At this point, we assume that l_v is below all the segments that are stored in v or in one of v's ancestors. We still need to determine whether or not l_v is completely hidden by the segments that are stored at the descendants of v. First we check the value of the boolean variable c_v. If it is false, that is, if the union of the projections of the segments stored at the descendants of v does not cover r_v, then clearly l_v is visible. Otherwise, we compare the height of l_v with y_v, the height that is stored at v. l_v is visible if and only if its height is less than y_v.

The query time is $O(\log n)$, where n is the current size of $\mathcal{S}$, since the number of ancestors visited all together is only $O(\log n)$.

Insertion. Assume we want to insert a new segment s into $\mathcal{T}$. We insert s into the appropriate $O(\log n)$ nodes of $\mathcal{T}$ (see [1]). In each of these nodes v, we insert s into the tree $\mathcal{T}_v$. We now need to update the boolean variables and the heights that are stored in the ancestors of v. This can be done quite easily by walking along the path from v to the root.

Consider the parent w of v. If c_w is already true, then it remains true. Otherwise, c_w is true if and only if either c_u is true or there is at least one segment stored at u, where u is the second child of w. Concerning y_w, if s is the lowest among the segments in $\mathcal{T}_v$, then we perform the second part of the visibility test above for the subsegment s_v (i.e., s restricted to r_v). That is, we determine whether s_v is visible taking into account only the segments in the descendants of v. If it is, then y_w is set to the maximum between its current value and the height of s, otherwise, y_w does not change. We now move to the parent of w, etc., until we reach the root. The total time spent is $O(\log^2 n)$, where n is the current size of $\mathcal{S}$.

Deletion. Assume we need to delete the segment s from $\mathcal{T}$. We delete it from each of the $O(\log n)$ nodes in which it is stored (see [1]). Let v be one of these nodes. We first delete s from $\mathcal{T}_v$. Now, we need to update the variables stored with the nodes along the path from v to the root. Let w be the parent of v. If c_w is false, then it of course remains false. Otherwise, if after removing s from v, v remains without any segments, then c_w is true if and only if c_v is true. Concerning y_w, if s was the lowest among the segments in $\mathcal{T}_v$, then we check for the new lowest segment in $\mathcal{T}_v$ whether it is visible, taking into account only the segments stored at the descendants of v. If it is visible, then y_w is set to the maximum between its current value and the height of this segment. The total time spent is $O(\log^2 n)$, where n is the current size of $\mathcal{S}$.

Theorem. *We can maintain a data structure of size $O(|\mathcal{S}| \log |\mathcal{S}|)$ in $O(\log^2 |\mathcal{S}|)$ time per update, so that given an horizontal query segment l, one can determine whether l is visible from the x-axis in $O(\log |\mathcal{S}|)$ time.*

References

[1] J. Bentley and J. Saxe, Decomposable searching problems I: Static-to-dynamic transformation, *J. Algorithms* 1 (1980), 301–358.

[2] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer, 1997.