Multiple Spectator Problem

Zachary Kieda (zkieda@andrew.cmu.edu)

May 31, 2015

Abstract

Here, we present an algorithm that computes the items in the field of view of a spectator on a 2-Dimensional plane. Our goal is to make the algorithm efficient, and easy to change when the spectators move.

Keywords. Computational Geometry, Artificial Intelligence, Algorithms

1 Problem Statement

Consider a 2-Dimensional plane with width w and height h. We consider this plane the field. There are n 'spectators' on this field, each with a different (x, y) position, Θ_d direction, and Θ_s field of view. Every spectator has the same, fixed radius. The field of view is the space defined by these parameters, such that other spectators are in its field of view only if it's above the planes that passes through the position of the spectator with angle $\Theta_d \pm \Theta_s$.

We want to find the field of view range for each spectator. A field of view range is a sequence of intervals that each spectator can see in its field of view sorted with respect to Θ, the left-to-right angle from the view of the spectator. Each interval consists of the left hand side and right hand side points of another spectator that is viewable from this spectator, the other spectator, and the distance from this spectator to the other. Gaining this information is useful in an artificial intelligence that wants to gain a higher level description of the objects in its point of view.

We also have to consider the runtime for updating data structures for when the positions or directions of spectators change.

A naive algorithm would take every spectator individually. Then, obtain the list L of other spectators that are in our field of view. Finally, we sort each spectator with respect to distance, and insert them from back to front into our data structure. We have n spectators, and process $n-1$ to compute L. Let $k = |L|$. Sorting L is in $k \log k$, and finding the intervals is in $O(L)$. In total, the output sensitive runtime is in $O(n^2 k \log k)$, with a maximum of $O(n^3 \log n)$. This algorithm does no preprocessing/stores no data structures between frames, so the runtime to update the positions and directions of spectators is $O(1)$.

We present a different output-sensitive algorithm that runs in $O(kn)$ to compute the field of views, and $n \log n$ to update each frame.

2 Proposed Solution

We store a Quad-tree data structure that maps the (x, y) position to a spectator. We traverse this quad-tree to give us all of the other spectators in sight. We assume the quad-tree is balanced, and we can get the nearest neighbors of an object in the tree in constant time.

The algorithm requires a valid quad-tree that maps the coordinate location of each spectator to the spectator itself. Collect a set of nodes L by the following DFS algorithm. Initially L is empty.

- For each spectator S, let $i \in Q$ represent the node that S is stored. We proceed as follows
    - Add all of the neighbors of i to L that (1) are not in L or (2) are not outside the field of vision. Immediately recursively process the neighbors after they have been added to L.
    - Terminate when all neighbors are already in L or are outside the field of vision.

Note that we can check if a neighbor is in our field of vision in $O(1)$ since we have a constant number of constraints. We can also find all adjacent neighbors in $O(1)$, since a quad-tree has a constant number of neighbors. So, creating L is $O(k)$.

Initially, we preprocess L, the nodes in our field of view, in $O(k)$ time. This is done by storing the minimum distance node to this spectator at every tree node in the quad-tree. We only store the minimum distance node out of all of our children. With this preprocessing, obtaining the nearest neighbor nodes is in $O(1)$. Since there are only at most four children per node, we can find the minimum node easily in constant time.

Then, we traverse and add the nearest neighbor in one at a time. We can find the next minimum node in $O(1)$ time, and we can add it to R also in constant time (See: Splitting Interval Table). So, the time this takes is the number of nodes we traverse in the quad-tree, which is in $O(k)$.

This might seem odd that we are getting a sorted list of nodes in merely $O(k)$ time. However, we are exploiting the euclidean distance property of the quad tree to give us this increased runtime. This technique is a rough analogy to storing the minimum values at each node of a BST.

Since we perform this for all n spectators, the total runtime is $O(kn)$.

To update a frame, at most, we have to move around all n spectators. The direction they are facing does not change any data structure, so we only need to update the quad-tree when a spectator moves out of its box. In the worst case, we reconstruct an entire quad-tree with n items. Since insertion into a balanced quad-tree is $O(\log n)$, we have to expend at most $O(n \log n)$ to update each frame. However, since a spectator generally does not move rapidly in a game, we can re-insert much less often.