Fast BSP tree generation using binary searches
[An informal paper]

Paul Nettle
5/13/2000

Preface

This idea is simply in the concept stages, it has no implementation for proof. The concept, however, *appears* sound. If you find success or failure in practical usage, please notify the author at: midnight@graphicspapers.com.

Introduction

When building a BSP tree, it is most often necessary to find a reasonable balance between “fewest splits” (i.e. smallest tree) and “least depth” (i.e. best balance of nodes.) In most applications fewest splits tend to be more important, unfortunately, this is a most time consuming task.

Many applications that require BSP trees use a heuristic to determine a balance between “fewest splits” and “least depth”. It is in this balance, that we find hope for an efficient solution.

It can be said, that because we’re searching for a balance between depth and splits, there are a certain number of potential polygons (acting as splitters) that will result in the deepest tree, and should not be considered, even if they result in nearly no splits.

Given these criteria, we can find a range of the top contenders for least depth, and from them find the contender that results in the fewest splits. As long as we have enough “least depth” polygons to choose from, we’ll find our target balance.

By doing this, we avoid the computationally expensive (and exhaustive) search for the fewest splits, and focus our energy on the least depth. From this reduced set, we find fewest splits, reducing our computation time.

Finding the top contenders for least depth

Consider a unit sphere. This sphere can be quantized into a set of n points that define the unit sphere. These points, when considered relative to the center of the unit sphere will generate a set of n vectors (this is sometimes referred to as a Gaussian map).

These vectors will be used to determine n lists of polygons (“splitter groups”), one polygon group per vector. To classify each polygon, simply find the vector whose

angle is closest to the polygon's normal (using a dot product) and place it into the appropriate group.

Each group is now sorted by each polygon's D (from the plane equation.)

If you consider a single group, you'll find that it contains a list of polygons that are nearly coplanar, with the error bounded by the difference in angle from its neighbors of quantized vectors. In other words, each group contains nearly coplanar polygons that sequentially (sorted on D) cut through the scene.

Assuming an evenly distributed dataset, it can be said that because each group is sequentially sorted you can visit each polygon in the group, split the entire dataset against that polygon and find that each successive polygon will incrementally raise the polygon count on one side, and decrease the count on the opposite side. If the database were evenly distributed, you would merely need to find the center of the list to find the "least depth" polygon for any particular group. Most datasets are rarely organized so evenly distributed. Because of this, we use a binary search through each group to find the best balance for that group. The polygon within each group is marked as the "least depth polygon."

During the binary search, since we're doing the work, we'll keep track of the number of splits for each polygon. We'll need this later.

With each group categorized, sorted and searched, it's becomes trivial task to find the group whose "least depth polygon" results in the least depth across all groups. This will be the "least depth polygon" for the entire scene (again, bounded by the error of the angle between neighboring quantized vectors of the unit sphere.)

The groups are now sorted according to the "least depth polygon" in each.

Narrowing in on a good balance

If there are n groups of polygons, then we have n "least depth polygons" (in sorted order of efficiency.) If n was chosen based on a heuristic which determines a reasonable threshold between "least depth" and "fewest splits" then there is a good chance that our balance between "least depth" and "fewest splits" can be obtained by one of the "least depth polygons".

To put a finer point on it, our "least depth polygons" will ideally (based on the heuristic) contain a range of polygons that will be acceptable for the "least depth" portion of our balance between "least depth" and "fewest splits."

This is not necessarily true, however, as it is possible that each "least depth polygon" will all end up with a perfect "least depth" leaving the proper balance

between “least depth” and “fewest splits” outside of the range of our list of “least depth polygons”. However, statistically speaking, this would seem to be a rare oddity.

Because our set of “least depth polygons” is sorted, and because each “least depth polygon” contains it’s contribution to splits and depth of the tree, we can linearly search this list for the best balance between depth and splits.

Making it faster

Because a polygon has two sides, which equally subdivide space, we don’t need to use an entire unit-sphere. We can simplify our task by using a unit semi-sphere. When classifying each polygon, we can simply negate the normals of back-facing polygons to the unit semi-sphere.

Polygon classification can be expensive [(polygonCount * n) dot products.] To reduce this, we can implement yet another binary search. The unit semi-sphere can be subdivided much like a quad-tree. Four vectors are centered in each quadrant in the semi-sphere. Each polygon is then classified to one of the four quadrants. Each quadrant is then subdivided, four more vectors are generated for each sub-quadrant, and the list of polygons in each quadrant is classified to a sub-quadrant. This process continues recursively until the subdivision level results n vectors covering the semi-sphere.