BSP Tree

Gerard Martin Teixidor

May 11, 2021

Overview

A binary space partitioning (BSP) tree, is a data structure which subdivides space into convex subspaces by using hyperplanes [1]. This data structure can be seen as a generalization of a k-d tree which, unlike a k-d tree, the orientation and position of the spiting planes are arbitrary.

In a BSP tree, each inner node represents an hyperplane that splits the space into two half-spaces. The right child represents the positive half-space, while the left child represents negative half-space. The leaf nodes, unlike the inner nodes, nodes does contain any information about the hyperplane and represent a convex subspace.

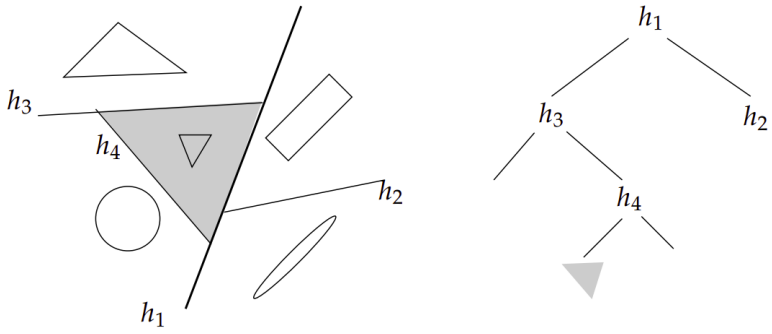

Figure 1: Example of a BSP tree.

Because of the freedom of choosing the position and orientation of the hyperplanes, a common approach is to use the planes defined by the input data. Given a set of polygons as the input data, when all the hyperplanes are polygons from the input data, it is called an auto-partition BSP tree.

In the computer graphics field, BSP trees are used in multiple applications. The most common applications are:

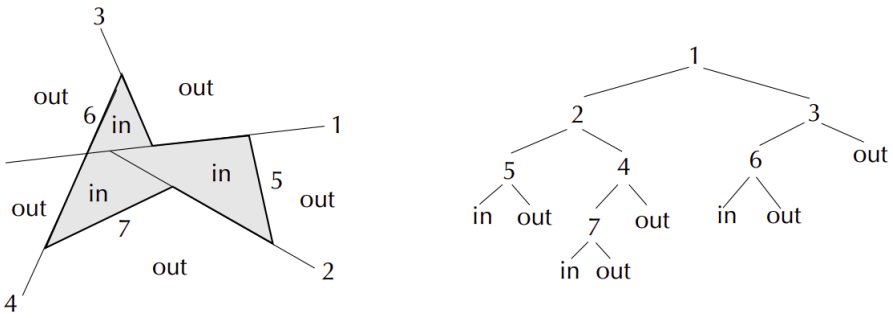

Figure 2: Auto-partition BSP tree which represents a closed object.

Constructive solid geometry Auto-partition BSP trees allow to represent volumetric polygonal closed objects by defining if a space space region is inside or outside the object. This representation also allows to check very efficiently if a point belongs inside or outside the object. Figure 2 shows an example of a 2D BSP tree representing an object.

Boolean operations Another interesting property of an object represented by a BSP tree is the ability to perform very efficiently boolean operation between objects. Given two objects, A and B, being able to perform the boolean operations $\{\cup, \cap, \setminus, \ominus\}$. As a simple overview, the main idea behind this operations is to split one of the two BSP tree by the planes of the other BSP tree, and merging them with the corresponding boolean operation.

Visibility priority When rendering the polygons on the screen, an auto-partition BSP tree can be used to determine the drawing priority of those polygons [2].

Visibility priority

In computer graphics, the visibility problem of a 3D scene consists on defining which parts of the geometry are going to we visible on screen. The painter's algorithm allows to solve this problem by painting those polygons from back to front by overdrawing those portions of the screen which are occluded by the polygons painted on top.

In static scenes, an auto-partition BSP tree can be used to solve this problem since it stores implicitly the polygon painting order from any view point.

Generation

Given a set of polygons as the input data, to generate an auto-partition BSP the following algorithm is performed:

1. A polygon is chosen from the list.
2. For each other polygon in the list:
   (a) If the polygon is completely in front of the hyperplane, it is added to the right child.
   (b) If the polygon is completely behind the hyperplane, it is added to the left child
   (c) If the polygon intersects with the hyperplane, the polygon is split and each new fragment is added to the corresponding child.
   (d) If the polygon lies in the plane, is added tot the list of polygons of that node.
3. This algorithm is applied recursively to each child until there is only one polygon on each node.

It is important to note that the selection of the polygon which specifies the hyperplane (step 1) is going to impact the number of fragments generated down the tree. There exist different heuristics which try to minimize the number of splits.

Image generation

Once the BSP tree has been generated, the tree can be traversed in linear time in order to determine the visibility priorities of each polygon.

From the root node:

1. If the view location is in front of the hyperplane
   - Paint the left child (behind) polygons recursively.
   - Paint the current node polygons.
   - Paint the right child (int front) polygons recursively.
2. If the view location is behind the the hyperplane
   - Paint the right child (int front) polygons recursively.
   - Paint the current node polygons.
   - Paint the left child (behind) polygons recursively.

To check if the view is in front or behind the plane, is just a matter com computing the singed distance of a point with respect of the hyperplane.

Personal opinion

Nowadays the BSP trees are not that relevant in the computer graphics field and seams to be taken over by other more advanced data structures. This is

Polygons	$Fragments/Polygons^2$
Monkey	0.004%
Sphere	0.028%
Teapod	0.017%
Random	1.633%

Table 1: Table showing the number of fragments generated by the BSP tree.

obvious in the case of the visibility problem. Nowadays the default solution to this problem is the use of the Z-Buffer. This alternative is not a data structure by itself, but instead is a hardware accelerated GPU buffer. This GPU buffer allows to perform the priority sorting directly on the GPU, and at pixel level, with a minimum performance penalty.

Also, in the case of solid modeling, the rendering of those objects does not require them to be represented by BSP trees. Like with the visibility problem, this rendering can be perform with object primitives and the help of another hardware buffer called stencil buffer.

Experiments

One of the main problem with BSP trees is the size of the tree. In the case of a BSP tree constructed from a set of polygons, this size of the tree can be much larger than the number of polygons n due to the hyperplanes splitting those polygons. While the number of fragments in an 2D auto-partition BSP tree is bounded $O(n^2 \log n)$, in higher dimensions the known bound is much wider. In the case of 3D BSP trees they can be as many as $O(n^2)$ fragments.

Even thought there can be as many as a quadratic number of fragments, the reality is that, due to the principle of locality, BSP trees contain much fever fragments. The principle of locality states that polygons are smaller compared to the whole space.

The implemented experiment tries to show this reality by constructing multiple BSP trees and comparing the number of polygons with respect to the number of fragments. The BSP trees generated are auto-partitions BSP trees from real models and from a set of random polygons.

As we can see in the results (Table 1) we can see that in reality, the number of fragments is far from the upper bound. Also this experiment shows the principle of locality as we can see that, when using random polygons, the number of fragments increase with respect to the models. This is due to the models being composed by small polygons, while the size of the random polygons is variable.

Conclusions

Even being forgotten nowadays, BSP trees are an interesting very data structure which still are being used. It seams that one of the causes of its decrease in popularity is that many applications now uses more advanced an efficient techniques which allow to parallelize the computation (e.g. Z-Buffer for the visibility problem).

References

[1] H. Futchs, Zvi Kedem, and Bruce Naylor. On visible surface generation by a priori tree structures. volume 14, pages 39–48, 12 1988.

[2] Gabriel Zachmann and Elmar Langetepe. Geometric data structures for computer graphics. In *Eurographics (Tutorials)*, 2002.