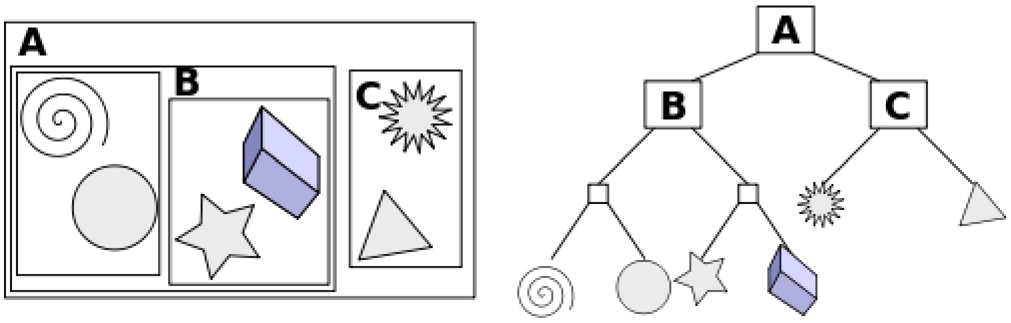
WIKIPEDIA

Bounding volume hierarchy



An example of a bounding volume hierarchy using rectangles as bounding volumes.

A **bounding volume hierarchy** (**BVH**) is a tree structure on a set of geometric objects. All geometric objects are wrapped in bounding volumes that form the leaf nodes of the tree. These nodes are then grouped as small sets and enclosed within larger bounding volumes. These, in turn, are also grouped and enclosed within other larger bounding volumes in a recursive fashion, eventually resulting in a tree structure with a single bounding volume at the top of the tree. Bounding volume hierarchies are used to support several operations on sets of geometric objects efficiently, such as in collision detection and ray tracing.

Although wrapping objects in bounding volumes and performing collision tests on them before testing the object geometry itself simplifies the tests and can result in significant performance improvements, the same number of pairwise tests between bounding volumes are still being performed. By arranging the bounding volumes into a bounding volume hierarchy, the time complexity (the number of tests performed) can be reduced to logarithmic in the number of objects. With such a hierarchy in place, during collision testing, children volumes do not have to be examined if their parent volumes are not intersected.

Contents

BVH design issues

The choice of bounding volume is determined by a trade-off between two objectives. On the one hand, we would like to use bounding volumes that have a very simple shape. Thus, we need only a few bytes to store them, and intersection tests and distance computations are simple and fast. On the other hand, we would like to have bounding volumes that fit the corresponding data objects very tightly. One of the most commonly used bounding volumes is an axis-aligned minimum bounding box. The axis-aligned minimum bounding box for a given set of data objects is easy to compute, needs only few bytes of storage, and robust intersection tests are easy to implement and extremely fast.

There are several desired properties for a BVH that should be taken into consideration when designing one for a specific application:[1]

- The nodes contained in any given sub-tree should be near each other. The lower down the tree, the nearer the nodes should be to each other.
- Each node in the BVH should be of minimum volume.
- The sum of all bounding volumes should be minimal.
- Greater attention should be paid to nodes near the root of the BVH. Pruning a node near the root of the tree removes more objects from further consideration.
- The volume of overlap of sibling nodes should be minimal.
- The BVH should be balanced with respect to both its node structure and its content. Balancing allows as much of the BVH as possible to be pruned whenever a branch is not traversed into.

In terms of the structure of BVH, it has to be decided what degree (the number of children) and height to use in the tree representing the BVH. A tree of a low degree will be of greater height. That increases root-to-leaf traversal time. On the other hand, less work has to be expended at each visited node to check its children for overlap. The opposite holds for a high-degree tree: although the tree will be of smaller height, more work is spent at each node. In practice, binary trees (degree = 2) are by far the most common. One of the main reasons is that binary trees are easier to build.[2]

Construction

There are three primary categories of tree construction methods: top-down, bottom-up, and insertion methods.

Top-down methods proceed by partitioning the input set into two (or more) subsets, bounding them in the chosen bounding volume, then keep partitioning (and bounding) recursively until each subset consists of only a single primitive (leaf nodes are reached). Top-down methods are easy to implement, fast to construct and by far the most popular, but do not result in the best possible trees in general.

Bottom-up methods start with the input set as the leaves of the tree and then group two (or more) of them to form a new (internal) node, proceed in the same manner until everything has been grouped under a single node (the root of the tree). Bottom-up methods are more difficult to implement, but likely to produce better trees in general. Some recent studies (e.g. [3]) indicate that in low-dimensional space, the construction speed can be largely improved (which matches or outperforms the top-down approaches) by sorting objects using space-filling curve and applying approximate clustering based on this sequential order.

Both top-down and bottom-up methods are considered *off-line methods* as they both require all primitives to be available before construction starts. *Insertion methods* build the tree by inserting one object at a time, starting from an empty tree. The insertion location should be chosen that causes the tree to grow as little as possible according to a cost metric. Insertion methods are considered *on-line methods* since they do not require all primitives to be available before construction starts and thus allow updates to be performed at runtime.

Usage

BVHs are often used in ray tracing to eliminate potential intersection candidates within a scene by omitting geometric objects located in bounding volumes which are not intersected by the current ray.[4]. Additionally, as common performance optimization, when only closest intersection of the ray is of interest, as the ray tracing traversal algorithm is descending nodes, and multiple child nodes are intersecting the ray, traversal algorithm will consider the closer volume first, and if it finds intersection there, which is definitively closer than any possible intersection in second (or other) volume (i.e. volumes are non-overlapping), it can safely ignore the second volume. Similar optimizations during BVH traversal can be employed when descending into child volumes of the second volume, to restrict further search space and thus reduce traversal time.

Additionally, many specialized methods were developed for BVHs, especially ones based on AABB (axis-aligned bounding boxes), such as parallel building, SIMD accelerated traversal, good split heuristics (SAH - surface-area heuristic is often used in ray tracing), wide trees (4-ary and 16-ary trees provide some performance benefits, both in build and query performance for practical scenes), and quick structure update (in real time applications objects might be moving or deforming spatially relatively slowly or be still, and same BVH can be updated to be still valid without doing a full rebuild from scratch).

BVHs also naturally support inserting and removing objects without full rebuild, but with resulting BVH having usually worse query performance compared to full rebuild. To solve these problems (as well as quick structure update being sub-optimal), the new BVH could be built asynchronously in parallel or synchronously, after sufficient change is detected (leaf overlap is big, number of insertions and removals crossed the threshold, and other more refined heuristics).

BVHs can also be combined with scene graph methods, and geometry instancing, to reduce memory usage, improve structure update and full rebuild performance, as well as guide better object or primitive splitting.

See also

- Binary space partitioning, octree, *k*-d tree
- R-tree, R+-tree, R*-tree and X-tree
- M-tree
- Scene graph
- Sweep and prune
- Hierarchical clustering
- Optix

References

1. Christer Ericson, Real-Time Collision Detection, Page 236–237
2. Ericson, Christer. *Real-Time Collision Detection*. p. 238. ISBN 1-55860-732-3.
3. Y. Gu, Y. He, K. Fatahalian and G Blelloch. Efficient BVH Construction via Approximate Agglomerative Clustering (http://graphics.cs.cmu.edu/projects/aac/aac_build.pdf). HPG 2013.
4. Johannes Günther, Stefan Popov, Hans-Peter Seidel and Philipp Slusallek, Realtime Ray Tracing on GPU with BVH-based Packet Traversal (https://archive.is/20130412080929/http://www.mpi-inf.mpg.de/~guenther/BVHonGPU/)

External links

- BVH in Javascript (https://github.com/imbcmdth/jsBVH).
- Dynamic BVH in C# (http://www.codeproject.com/Articles/832957/Dynamic-Bounding-Volume-Hiearchy-in-Csharp)
- Intel Embree open source BVH library (https://www.embree.org/)