# October: a combined structure of octree and oriented bounding box

Yu Wang
Department of Computer Science
University of Maryland
ywang185@terpmail.umd.edu

## Goals of this project

In this project, I created a combination of octree and oriented bounding box. My initial purpose was to upgrade OBB-tree so that it takes less time to construct the tree and traverse the tree. But it has been updated in the process of implementing it.

For now, my purpose is more stressed on the efficiency of traversing the tree. Since construction only happens once in the beginning. Another thing I'd emphasize is to control the error by introducing an evaluation called "space per vertex" (spv). I will describe this idea in later paragraph.

## Prior work

Octree: in 1980, Professor Donald Meagher at Rensselaer Polytechnic Institute presented an idea of using octrees for 3D computer graphics in a report "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer". He also holds a patent of which called "High-speed image generation of complex solid objects using octree encoding". In this project, I am just using the data structure since it is natural to use for partitioning 3-D space.

Oriented bounding box: there are many types of bounding boxes. Unlike axis-aligned bounding boxes (AABB), an oriented bounding box can have arbitrary orientation, which can bring us tighter bounding boundary comparing with AABB.

OBB-tree: this idea was brought up in an article called "OBBTree A Hierarchical Structure for Rapid Interference Detection" by S. Gottschalk, M. C. Lin and D. Manoch. This idea has inspired me because this structure only does space partitioning on the longest axis. It is obvious that if more space partitionings are done in one level, the tree structure will be shallower and easier to traverse/access. Personally speaking, I think OBB-tree structure has its advantage on elongated 3-D objects such as a pole, but on a more uniform-shaped 3-D object, October has its advantage of efficiency. In addition, October is able to limit the error of distance. This error-controlling protocol starts when the October structure is built.

## The approach I pursued

To make it more efficiency to construct the tree, I thought keeping only one copy of the geometric information (like the coordinates of points) should be faster than making copies for each related tree node. But then, labeling them with different indices of corresponding tree nodes would be the extra part of work to build the tree. So, I finally decide to send a copy into the construct method and use the copy to calculate corresponding features, but use the copy as a temporary variable so that it won't largely increase the size of the data structure. After discussing the idea of this project with Professor Manoch, I stopped trying more potential ways to shorten the construction period.

To make it more efficient to traverse the tree, in this structure, a node can find the desired child node (the closest one to a given coordinate) directly by projecting the given coordinate into a local coordinate system and check the octant of this corresponding new coordinate. I call this process "reflection". This reflection can be done recursively till we find the proper leaf child.

To control the error (the error of distance that triggers in a collision of bounding boxes when the objects don't actually has one), spv is used to control how the October is built. When a certain spv is given, the construction procedure will only be terminated when all the leaf nodes are qualified with the limitation of spv.

## Results

A short demo:

A jupyter notebook file called "OctoberDemo" is uploaded together with this report. The functionality of October is shown inside in a naive way. I haven't construct a specific data set for this, probably I will make one if I figure out how the dataset should be represented properly.

Basic idea:

pseudo code:

```
class Object
{
        October october         //the structure I create in this project
        Matrix tmatrix          //the transform matrix

        func update(newmatrix):         //a function to update the stored matrix
        self.tmatrix=newmatrix*self.tmatrix
        return
}
class October
{
        func init(points, spv):
        self.boundingbox=boundingbox(points)
```

```
        if self.boundingbox.spv<spv:
            return
        else:
            partition space according to pca
            split points according to pca
            foreach child in self.children:
                child=October(corresponding points, spv)
            return
}
func collisiondetection(object1, object2):
    c1=nearest leaf node to object2 in object1.october
    c2=nearest leaf node to object1 in object2.october
    return c1.collides(c2)
```

*The final version of code can be different after continuous improvements. But the key ideas are the same.

Add-on:

for the case that a bounding box's spv is too high, after enough rounds, the bounding box will be a singleton that only contains one point.

Math descriptions of space per vertex and error-controlling details:

Space per vertex, spv, is a virtual cube. To describe this idea, we need to abstract the bounding box as a cube (Big cube), and each vertex inside this bounding box takes the space of a smaller cube (Small cube) that has a volume that splitting the volume of "big cube" evenly for each vertex. The volume of a "small cube" is exactly spv. To control the error, a threshold of spv is set in the construction part. And the error can be bounded by as the following:

$$Error(obj1, obj2) = \sqrt{3} * (\sqrt[3]{spv(obj1)} + \sqrt[3]{spv(obj2)})$$

This idea is depending on a hypothesis (Uniform distribution hypothesis): after certain rounds of tree-construction, the vertex in each current-leaf nodes will be more likely to be distributed uniformly in the corresponding bounding box.

To work around this hypothesis, there are two ways. First of all, we can use this method with dense meshes only. In this case, it already has the feature we want: uniformly distributed vertices; but the density of the mesh may conflicts with spv: for an object with real uniformly distributed vertices, it is not possible to get a bounding box in which the vertices are significantly denser than the object's density, unless the meshes are broken into edges and/or singletons. In my implementation, edges and/or singletons can be handled either by interpolation or treating it as singleton. So this won't be a problem if the spv value is reasonable.

Another way is to prove it. After I struggled on this for a pretty long while, I couldn't figure out a way to prove it directly. So I would describe it in a more intuitive way: since October does spatial partitioning on the longest axis (also on the other two), it is separating vertices into smaller groups, bounded by smaller and smaller bounding boxes. If we continue

this procedure, the leaf node of October either ends up as a singleton, or ends up with a tight enough bounding box. When spv is set to a critical value, October can “kick” a far-from-other vertex outside from a group since the critical value is between the spv without this vertex and the spv with this vertex. As I assume that the vertices come from a tight enough mesh, we can observe the mesh first to figure out a reasonable spv. Or use brutal-force methods to find a perfect spv among a reasonable range.

## Extensions for future work

With a limited amount of time, I am not able to finish a large and complicated demo of how October works and how it really performs. For the uniform distributed hypothesis, I haven’t figured out a way to prove it properly. If I have an inspiration someday, I would like to add a nice proof.

I think this data structure can also be used for computer graphics such as accelerating rendering. I will compare the performance of October to the one of normal Octree in Nori (open-sourced render) when I have time.