Exploring Spatial Data Management in 3D Graphics and Game Engines

Joshua Beam
Auburn University
josh@joshbeam.com

Abstract

3D graphics applications and video games must manage and use a large amount of spatial data for a variety of tasks, including rendering, visibility determination, collision detection, and more. A variety of data structures and algorithms have been utilized in game engines, each with their own strengths and weaknesses depending on the types of spatial objects to be managed (such as moving and static objects).

This paper will describe several data structures for managing spatial data in real-time 3D graphics applications, and will present results from a simulation program developed to aid in measuring and comparing the performance of such data structures.

1. Introduction

3D graphics applications, and particularly video games, often involve a large amount of spatial data. The types of spatial data that may be found in video games include the polygons/geometry that make up 3D models and objects, the locations of objects, and bounding regions for the scene and objects (including both detailed bounding regions and approximations based on boxes, spheres, and cylinders).

In real-time 3D graphics applications, in which the scene may be rendered more than 60 times per second, efficiently managing and utilizing spatial data is an integral part of the development of multiple subsystems. Spatial data is used for tasks such as visibility determination, collision detection, rendering, and more.

Consider the task of collision detection. For a scene with thousands or millions of objects, it is not feasible to check each individual polygon against every other polygon to test for collisions. Instead, a filter/refine process that is commonly applied in spatial databases [1] can be used. Approximate bounding regions, such as minimum bounding boxes, are utilized to quickly rule out collisions between two objects if their bounding regions do not intersect, and, in the event that they do intersect, more complex comparisons between the two objects can then be performed [1]. A spatial data structure such as an octree or Binary Space Partitioning (BSP) tree can also be used to group objects based on their location in 3D space, allowing the application to quickly eliminate the possibility of a collision between two objects that are clearly not near each other [2].

The management of spatial data in video games is complicated by the fact that some objects move, while others are stationary. It may be necessary to use different techniques for managing moving objects versus static objects, depending on the strengths and weaknesses of the data structures and algorithms involved. For example, if the algorithm for setting up a particular spatial data structure performs poorly but its algorithms for utilizing spatial data perform very well, it may be good to use for static objects, because the set up process only needs to be performed once and the resulting data structure can be precomputed and stored for later use [2]. Such an algorithm would not be optimal for moving objects, however, because the data structure must be adjusted in real-time as objects move throughout the scene, and the structure may not perform efficiently as changes are made [2].

Given this reliance on spatial data, it's no surprise that game engines often apply well-known spatial data structures and algorithms (similar to those used in spatial database management systems) in order to achieve optimal performance. This report will explore several spatial data structures that are suitable for use in game development, discussing the strengths and weaknesses of each for such applications. Simulations will be used to determine the relative performance of some of these data structures when managing large amounts of both moving and static objects.

2. Data Structures

Data structures traditionally used in spatial databases, such as the R-Tree [3], may be used in 3D graphics applications, but several other data structures are more frequently used. In this section, we will discuss several of the spatial data structures that are suitable for use in real-time 3D graphics applications.

As mentioned previously, a 3D graphics application may need to adjust its spatial data structures in real-time as objects move across the scene; therefore, we are concerned not only with the speed of traversing the data structure and accessing the objects contained within it, but also the time required to set up the data structure and modify it as necessary. Note also that, although it may be possible to adjust a data structure in real-time, it may not be possible to adjust it in an optimal manner in real-time, and so the data structure may not perform efficiently as changes are made while running the application [2].

2.1. Bounding Volume Hierarchy

The Bounding Volume Hierarchy (BVH) is a simple variation of the tree data structure that is familiar to most computer scientists and software engineers, where each node is associated with a bounding volume, which may be represented by a sphere, a box, a cylinder, or any other sort of three-dimensional object that is appropriate [2, 4]. The bounding volume for a node must simply enclose the bounding volume for every one of its child nodes (or, in the case of a leaf node, its bounding volume must enclose the objects in the node).

Figure 1 provides a visual representation of such a hierarchy utilizing a box and two spheres as bounding volumes. The root node of the hierarchy is a box that has two child nodes that are spheres, each containing two objects. The goal is to obtain a balanced tree by grouping a similar number of objects together based on their proximity to each other. In this example, the two objects in the upper left are reasonably close to each other, so they are placed in one child node. Similarly, the two objects in the lower right of the figure are close to each other and placed in another child node.

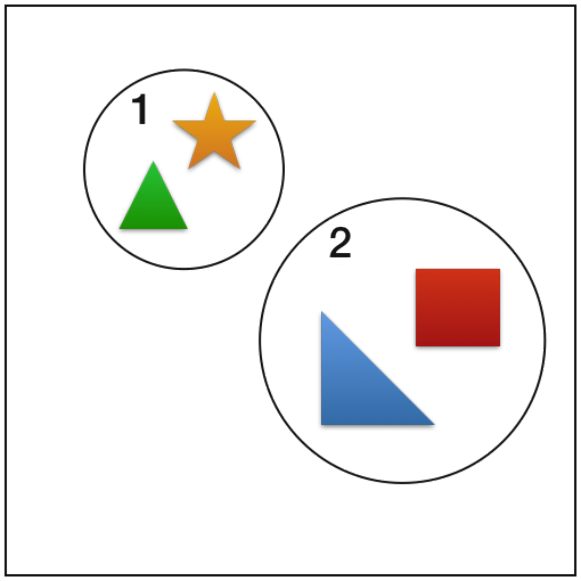


Figure 1. Bounding Volume Hierarchy

The purpose of the hierarchy is to speed up the process of querying for objects based on point data (for example, to determine whether an object is visible to the user or to detect collisions between objects). Imagine that we have been given a point that happens to lie within the green triangle in the upper left portion of the figure; we first check to see whether or not the point is within the root node of the hierarchy. It is, so we next check to see which (if any) of the root's child nodes the point is within; we find that it is within child node 1, so we can then perform the appropriate checks to see whether or not the point is inside any object in the child node. With this hierarchy, we did not need to check the objects in child node 2 at all.

The Bounding Volume Hierarchy provides a tremendous amount of flexibility, as there is no restriction on the types of bounding volumes that may be used. This data structure can lead to good performance for static objects, for which as much preprocessing time as is needed can be devoted to constructing an optimal hierarchy, but is not ideal for moving objects; when objects move, the hierarchy will need to be reconstructed such that objects that are near each other are grouped together in an optimal fashion, which may require a lot of processing time when there are many objects [2]. This required processing time may make the data structure infeasible for managing moving objects in real-time.

2.2. Octree

The octree is the natural three-dimensional extension of the quadtree (a popular data structure for organizing two-dimensional spatial data), in which each node is represented by a box and may be subdivided into eight equally sized child nodes [2, 5].

The root node is a box that encloses the entire space of the scene. This node is split in the middle of each of the three dimensions, resulting in eight child nodes. This process may be repeated recursively, until a node contains no more than one object, or until the leaf nodes reach a minimum size, which can prevent excessive splitting of nodes when objects are clustered together.

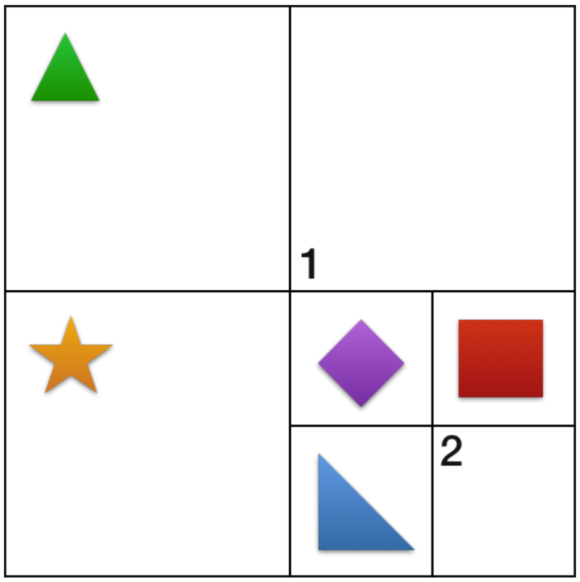


Figure 2. Octree (non-uniform leaf nodes)

Figure 2 shows a top-down graphical representation of an octree (which, in this two-dimensional representation, appears the same as a quadtree would). The outer-most square represents the root node of the octree, which is the first node to be subdivided into child nodes. Of these child nodes, only the one in the bottom right contains more than one object and needs to be subdivided further, so it is the second node to be split.

At this point, no node in Figure 2 contains more than one object, so splitting can stop there [2]. Ending the splitting process this way results in leaf nodes with non-uniform sizes. Another option is to keep splitting until each leaf node reaches a certain minimum size, resulting in leaf nodes with uniform sizes (Figure 3). The latter method is advantageous when there are moving objects because the entire octree can be built upfront without nodes having to be added and/or removed as objects are added, removed, or repositioned throughout the scene.

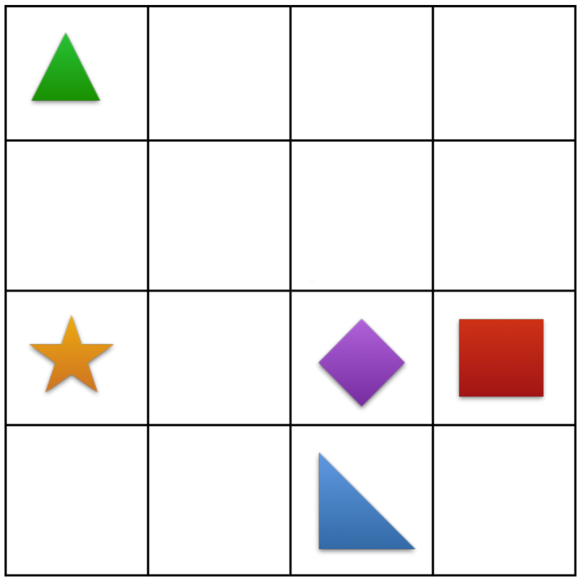


Figure 3. Octree (uniform leaf nodes)

The key advantage of the octree compared to the Bounding Volume Hierarchy is its predictable structure [2]. There is no need for a complex mechanism to determine the optimal way to divide each node; either a node needs to be divided or it doesn't, depending on the number of objects that it encloses and/or its size. This makes the data structure effective for managing moving objects as well as static objects.

2.3. Binary Space Partitioning (BSP) Tree

The Binary Space Partitioning (or BSP) Tree is a binary tree structure in which each node divides space along a plane, resulting in two child nodes that represent space on either side of the plane, which may be boxes in the case of an axis-aligned BSP tree [2, 6].

Typically, this dividing continues recursively until each leaf node of the tree contains only one object.

A graphical representation of a BSP tree can be seen in Figure 4. In this example, the enclosing square represents the root node of a BSP tree containing five objects. The root node is first divided along the plane represented by line 1, resulting in two child nodes, one of which contains the star at the bottom, and the other containing the other four objects. The latter node is split along the plane represented by line 2, resulting in two child nodes containing two objects each, and these child nodes are finally split along the planes represented by lines 3 and 4 to obtain the final BSP tree containing just one object per leaf node.

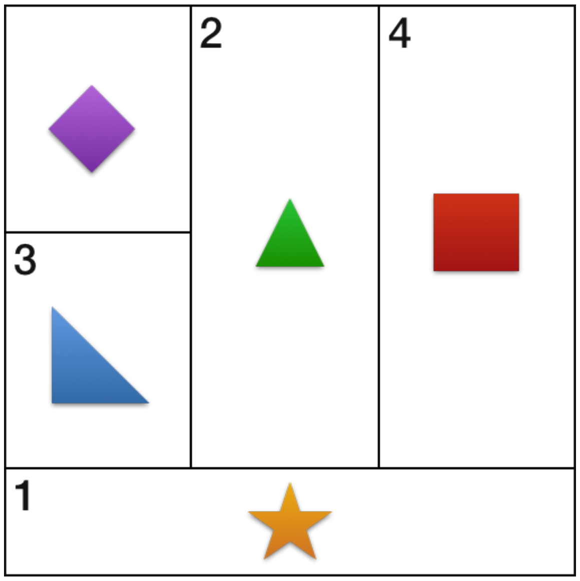


Figure 4. Binary Space Partitioning Tree

While the BSP tree described above is valid, it is not optimal, because one child of the root node contains just one object while the other contains four; it is preferable to achieve a more balanced BSP tree for performance reasons, and coming up with such an optimal tree may require extensive processing if there are many objects [2, 6]. For this reason, BSP trees are better suited for managing static objects, rather than moving objects.

2.4. k-d Tree

The last data structure that we will discuss is the k-d Tree, a binary tree structure that stores *k*-dimensional points [7, 8]. In our case, *k* would be equal to three, for three dimensions.

Each node in a k-d Tree represents a point, and has two child nodes, which can be called left and right. Each node also specifies a dimension *d* such that $1 \leq d \leq k$. This dimension is used to divide space between the left and right child nodes of a node; for any point *p* stored beneath a node, if that point has a lesser value in dimension *d* than the point that the node represents, then *p* should be stored somewhere in the left child of the node, otherwise it should be stored in the right child of the node [7, 8].

This is illustrated in Figure 5. In this example, point 1 is the root node of the tree, and its *d* value is 2 to specify the Y dimension. The point below the horizontal line intersecting point 1 becomes the left child of point 1, and one of the points above can be chosen as the right child. Point 2 is chosen as the right child, and its *d* value is 1 to specify the X dimension, so the remaining point to the left of the vertical line intersecting point 2 becomes the left child of point 2. There is no right child for point 2 because there are no points to the right of the vertical line.

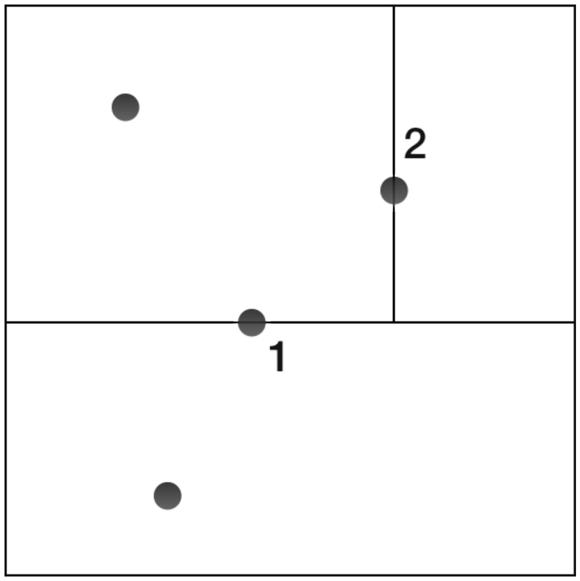


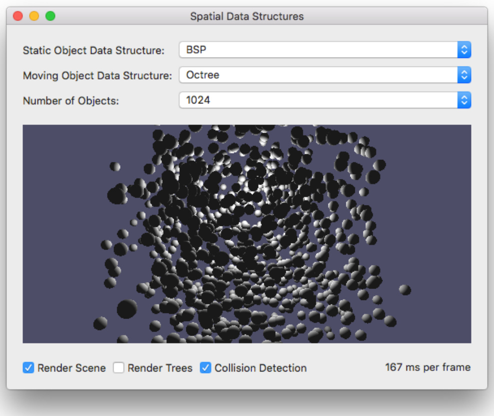
Figure 5. k-d Tree

This data structure has traditionally been used for static data, but there has been research into generating k-d trees dynamically [8].

3. Evaluating the Data Structures

Now that we understand some of the available options for spatial data structures that can be utilized in 3D graphics applications, we can compare them to determine which ones give the best performance in a given scenario.

One way to accomplish this is to set up a scene containing many objects, both static and moving, and to measure the performance of the scene when utilizing different combinations of data structures. For this paper, that task was accomplished with the aid of a custom program that was developed to simulate such a scene [9]. The program features a scene with many objects, and allows the user to select different data structures to use for static and moving objects in order to see how each combination performs. A screenshot of the program can be seen in Figure 6.



Static Object Data Structure: BSP
Moving Object Data Structure: Octree
Number of Objects: 1024

✓ Render Scene ☐ Render Trees ✓ Collision Detection 167 ms per frame

Figure 6. Simulation Program

3.1. Program Functionality

The program is a Mac OS X application written in the Swift programming language. It uses OpenGL for graphics rendering. At the top of the program's user interface are three dropdown lists, for selecting a static object data structure, a moving object data structure, and the number of objects to include in the scene (half of the selected number are created as static objects, while the other half consists of moving objects).

The bottom of the window includes checkboxes for enabling/disabling scene rendering, tree rendering, and collision detection. Disabling scene rendering can be used to ensure that graphics rendering is not the bottleneck when using the program to evaluate data structures. Tree rendering can be used to visualize the data structures that the program generates. Enabling and disabling collision detection can be used to verify that querying the data structure for intersecting objects has the biggest impact on performance. The counter in the lower right corner of the window shows the number of milliseconds spent processing each frame that is rendered; for example, when rendering a scene containing many objects, the program may spend 300 milliseconds per frame if collision detection is enabled, but only 100 milliseconds per frame if it is disabled. This shows that the performance of the spatial data structures containing objects is critical to the overall performance of the simulation.

The program also includes the ability to automatically measure the performance of each combination of static object data structure, moving object data structure, and number of objects, exporting the results to a Comma-Separated Values (CSV) file that can then be imported into a spreadsheet and used to generate charts. This functionality was used to generate the charts that appear later in this paper.

3.2. Design and Implementation

The program's source code was designed to allow for easy modification and addition of data structures. Each data structure implemented in the program conforms to a Swift protocol named *SpatialTree*, which is defined as follows:

```
protocol SpatialTree {
    var box: Box { get }
    var subtrees: [SpatialTree] { get }
    var objects: Set<SpatialObject>{ get }
    func optimize()
    func addObject(obj: SpatialObject)
        -> Bool
    func removeObject(obj: SpatialObject)
        -> Bool
}
```

Every class that implements a data structure conforms to the above protocol, allowing one data structure to easily be replaced with another. This interface also simplifies the process of implementing new data structures.

An instance of a class implementing the *SpatialTree* protocol represents a node in a tree-based data structure, such as those presented earlier in this paper. Each node has a bounding box, any number of subtrees (each node can be considered the root of its own tree, hence child nodes are referred to as *subtrees* in the code), and a set of objects contained in the node, represented by instances of the *SpatialObject* class.

As discussed earlier, some data structures, such as the BSP tree, may be optimized for performance reasons. Data structures may implement the *SpatialTree* protocol's *optimize* function in order to optimize their structure. The simulation program calls this function after generating trees of static objects, but not for trees of dynamic objects, since dynamic objects move throughout the scene as the program runs and the optimization code may be too costly to keep running as objects move. For data structures that do not require optimization code (such as the octree), this function may do nothing.

The *addObject* and *removeObject* functions simply add an object to or remove an object from the data structure. Additional functionality that is common to each data structure is implemented in a Swift protocol extension. For example, the *updateObject* function will remove an object and add it back to the data structure taking its current position into account, which may have changed since the object was previously added to the data structure if it is a moving object. The *findIntersectingObjects* function will return a set of objects contained in the data structure that intersect the given object.

The functions defined by the *SpatialTree* protocol operate on objects represented by the *SpatialObject* class. The most important property of this class is *boundingVolume*, which represents the object's bounding volume. This property is always a sphere (consisting of a three-dimensional position and a radius) in the simulation program, since all of the static and dynamic objects that are generated are spheres.

3.3. Data Structure Implementation

When it came time to implement data structures for the simulation program, the decision was made to focus on the octree and the axis-aligned BSP tree, as they presented the most different approaches among the data structures discussed in this paper, and the other data structures are similar to them; an octree can be seen as a more rigidly structured Bounding Volume Hierarchy, while the k-d tree is a variant of the axis-aligned BSP tree [2]. For the exact details of the included data structure implementations, refer to the source code for the simulation program [9].

A portion of the *Octree* class definition is included below:

```
class Octree: SpatialTree {
    var box: Box
    var subtrees: [SpatialTree] = []
    var objects = Set<SpatialObject>()

    var minSize: Float

    init(_ box: Box, _ minSize: Float =
10.0) {
        self.box = box
        self.minSize = minSize
    }

    var subBoxes: [Box] {
        let min = box.min
        let max = box.max
        let mid = box.mid

        return [
            Box(min: Vector3(min.x, min.y,
min.z), max: Vector3(mid.x, mid.y,
mid.z)),
            ...
        ]
    }

    private func split() {
        let size = box.size
        if size.x < minSize &&
           size.y < minSize &&
           size.z < minSize {
            return
        }

        subtrees =
subBoxes.flatMap({ Octree($0, minSize) })

        ...
    }

    ...
}
```

As mentioned in Section 2.2, there are two methods that can be utilized for octree construction: either tree nodes can be created and removed as needed when objects are added to or removed from the data structure, or all nodes can be created upfront so that they do not need to be created and removed dynamically. During the development of the simulation program, both methods were tested, and they were found to be similar in performance, so the first method was used in the final implementation of the data structure. The *subBoxes* property in the code above returns an array of the eight bounding boxes for the subtrees of the current octree node (for brevity, the construction of only one such box is shown in the code above; the construction of the other seven boxes is similar). The *split* function then uses these bounding boxes to create the subtrees if the tree has not reached a given minimum node size in each dimension.

The *addObject* function from the *Octree* class is shown below:

```
func addObject(obj: SpatialObject)->Bool {
    if objects.contains(obj) {
        return true
    }

    if !objectInTreeBox(obj) {
        return false
    }

    objects.insert(obj)

    if subtrees.count == 0 {
        if objects.count > 1 {
            split()
        }
    } else {
        for subtree in subtrees {
            subtree.addObject(obj)
        }
    }

    return true
}
```

When an object is added to a leaf node of the tree (that is, the node has zero subtrees), the node will be split into eight subtrees if there are now multiple objects inside of it; otherwise, if the node has already been split, then the object is recursively added to any subtrees that it intersects.

The *removeObject* function includes logic to merge a node by removing its subtrees:

```
func removeObject(obj: SpatialObject)
        -> Bool {
    if objects.remove(obj) == nil {
        return false
    }

    if subtrees.count != 0 {
        if objects.count < 2 {
            merge()
        } else {
            for subtree in subtrees {
                subtree.removeObject(obj)
            }
        }
    }

    return true
}
```

If removing an object from a node results in it having fewer than two objects, then the node is merged; otherwise, the object is recursively removed from the node's subtrees.

The implementations of the *addObject* and *removeObject* functions in the *BSPTree* class are similar, although the method of splitting a BSP node is different. When a BSP leaf node has a second object added to it, the node is split in the dimension which has the greatest distance between the two objects.

It should be noted that splitting and merging BSP nodes in real-time as objects are added to and removed from the tree will result in the tree becoming unbalanced over time [2]. The *BSPTree* class implements the *optimize* function to alleviate this for static objects; it attempts to optimize the tree such that each of the two subtrees of a node containing multiple objects will contain roughly half of the objects contained in the node. Rebuilding the tree in this way results in optimal performance when accessing it.

4. Simulation Results

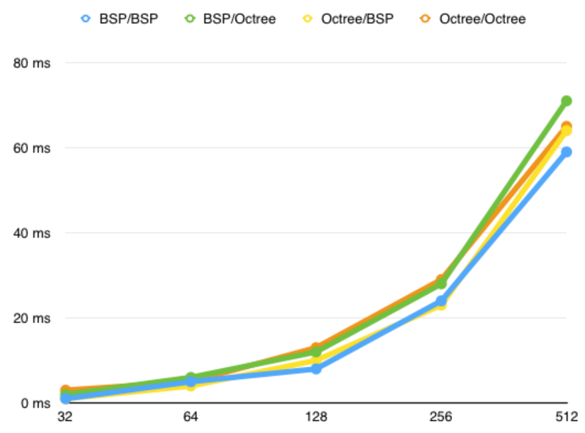


Figure 7. Results with 32 to 512 objects

We will now take a look at some results from the simulation program. These results were obtained by running the program on an Apple MacBook Pro computer containing a 2.3 GHz Intel Core i5 processor and 8 GB of memory. Figure 7 shows the performance of different data structure combinations with 32 to 512 objects.

The legend at the top of the figure shows different combinations of data structures represented in the chart, where the static object data structure is given before the slash and the moving object data structure is after. The horizontal axis shows the number of objects along the bottom of the figure, while the vertical axis shows the number of milliseconds required for processing each frame along the left side of the figure, so smaller values are better. For smaller numbers of objects, as represented in this chart, using a BSP tree for both static and moving objects gives the best results.

For larger numbers of objects, however, we see different results. Figure 8 shows the performance of the data structures for 8,192 objects.

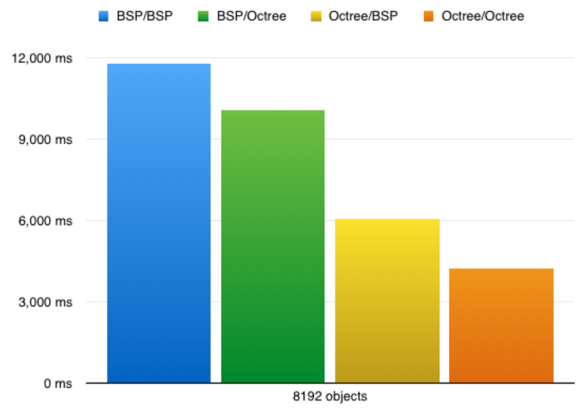


Figure 8. Results with 8,192 objects

With a large number of objects, the octree performs better, with the best performance seen when using an octree for both static and moving objects.

A possible explanation for this is the fact that there is more overhead involved in checking for intersections between objects and tree node bounding volumes in an octree. Each node that has subtrees has exactly eight subtrees which all must be checked when checking for intersecting objects, which makes a large impact when there are fewer objects.

As the number of objects increases, this overhead has a lesser impact, and the octree's ability to eliminate a larger fraction of space from consideration at each level of the tree results in improved performance. The predictable structure of the octree may also contribute to this improved performance as the number of objects increases, because there is less work involved in determining how best to split each node.

As shown in the code for the *Octree* class, one of the parameters for the data structure is a minimum node size, which causes the class to stop subdividing once a node has reached that size in each dimension. For the results shown in Figure 7 and Figure 8, a value of 25 was used. Another test was run to compare octrees with different values for this size; Figure 9 shows the results, again when processing a scene with 8,192 objects (the minimum node size is shown in parentheses in the legend above the chart; the same size was used for both static and moving objects).

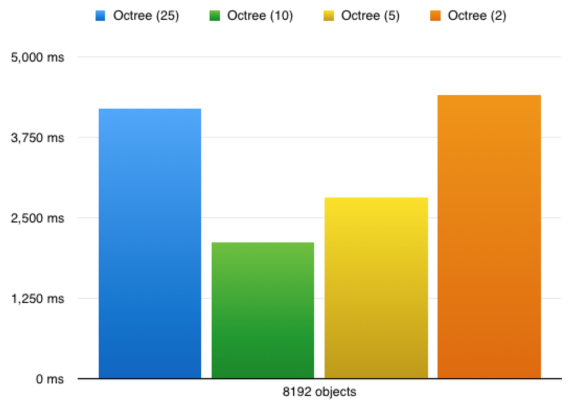


Figure 9. Comparison of octrees with different minimum node sizes

The optimal minimum node size depends primarily on the proximity of the objects in the scene. Of the minimum node sizes tested with the simulation program, the best performance is seen with a value of 10, which results in processing times about twice as fast as with values of 25 or 2.

The minimum node size also helps explain why octrees perform better than BSP trees with a large number of objects in the simulation program: the BSP tree is subdivided until each object has its own leaf node, whereas the octree can end up with leaf nodes that contain multiple objects. As mentioned earlier, the minimum node size prevents excessive splitting of octree nodes, limiting the depth of the tree. At a certain point, the time involved in traversing a deep tree, checking for intersections with the bounding volumes of tree nodes along the way, can become more expensive than simply checking for intersections with individual objects.

5. Conclusion

There are a number of choices for spatial data structures to use in 3D graphics applications, and it is not obvious which ones perform best. As the results in the previous section show, you may see different results depending on the number of objects being managed by the data structures, or on the values of parameters such as the octree's minimum node size. Other factors may also have an impact, and the developer of a real-time 3D application is encouraged to measure the performance of multiple data structures using methods such as those presented in this paper in order to find the best data structures to use for a particular application.

The source code for the simulation program has been posted online, and its development will continue as an open source project. Future work will involve implementing more data structures, continuing to optimize the data structures in order to achieve greater performance, and porting the program to iOS, so that the performance of these data structures can be measured on a mobile platform. Readers who are interested in this topic are encouraged to follow the project [9].

6. References

[1] S. Shekhar and S. Chawla, *Spatial Databases: A Tour*, 1st ed. Prentice Hall, 2003, ch. 1.

[2] T. Akenine-Möller and E. Haines, *Real-Time Rendering*, 2nd ed. Wellesley, MA: A K Peters, 2002, ch. 9, pp. 345-355.

[3] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *ACM SIGMOD*, vol. 14, no. 2, pp. 47-57, Jun. 1984.

[4] J. T. Klosowski *et al.*, "Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 1, pp. 21-36, Jan.-Mar. 1998.

[5] Z. Tang, "Octree Representation and Its Applications in CAD," *Journal of Computer Science & Technology*, vol. 7, no. 1, pp. 29-38, 1992.

[6] M. S. Paterson and F. F. Yao, "Binary Partitions with Applications to Hidden-Surface Removal and Solid Modelling," *ACM Proceedings of the fifth annual symposium on Computational geometry*, pp. 23-32, 1989.

[7] J. L. Bentley, "Multidimensional Divide-and-Conquer," *Communications of the ACM*, vol. 23, no. 4, pp. 214-229, Apr. 1980.

[8] K. Zhou *et al.*, "Real-Time KD-Tree Construction on Graphics Hardware," *ACM Transactions on Graphics*, vol. 27, no. 5, Dec. 2008.

[9] J. Beam. (2016, Apr. 24). *Spatial Data Structures Simulation Program*. [Online] Available: https://github.com/joshb/SpatialDataStructures