Modeling Dynamic Surfaces with Octrees

Don Libes

Integrated Systems Group
National Institute of Standards and Technology
Gaithersburg, MD 20899

ABSTRACT

Past uses of octrees have been for representation of static objects. We discuss extensions necessary to model dynamic surfaces. One particularly important aspect of this is the ability to represent expanding surfaces that grow to be arbitrarily large. Our enhanced octree does exactly this, and models contraction as well.

The ability to represent dynamic surfaces allows us to apply octrees to new problems which could not previously have been modeled with static octrees. One such problem is the *Entropy of Random Surfaces*. Using dynamic octrees, we produced a simulation of self-avoiding random surfaces using Monte Carlo techniques.

Keywords: 3D modeling, Random Surface Theory, octrees, data structures.

Introduction

The octree is a data structure for storing information about static 3D surfaces. This paper presents enhancements which allow octrees to be used with dynamic surfaces as well. The ability to represent dynamic surfaces allows octrees to be applied to new problems which could not previously have been modeled with static octrees. For example, an autonomous robot could explore and model an arbitrarily large universe with these techniques.

The first section of this paper is a brief overview of octrees, and may be skipped by readers familiar with the theory. The second section motivates our enhancements by discussing the typical uses and limitations of octrees. The third and fourth section discuss our enhancements in detail. The fifth section describes how the implementation differs from a typical octree system. The sixth section presents our first application using the enhanced octree. Squeamish physicists may wish to read this section first as an incentive.

Originally issued as NISTIR 89-4055, July 18, 1989. Reprinted from *Computers & Graphics*, Pergamon Press, New York, NY, Vol. 15, No. 3, 1991.

1. Octrees – Brief Description and Background

Octrees are a data structure for storing information about a 3D space. While capable of storing arbitrary information, octrees are commonly used for representing volumes or surfaces. Octrees have particular advantages over other representations when the volumes contained are highly connected or *blobby*. This will be discussed further in the next section. Octrees also have disadvantages in certain types of modeling. A complete discussion of the advantages and disadvantages of octrees and run/space-time analyses are discussed by [Meagher]. [Samet] provides a comprehensive study of octrees and related representations.

Briefly, the octree data structure is a tree composed of *octants*, each of which defines a cubical volume. Part of each octant's data structure denotes whether the space in the cube is empty, full or neither (i.e., partially full). Octants that are partially full have eight child octants (hence the prefix "*oct*"). The eight child octants together exactly fill the space modeled by their parent (see figure 1).

Like its parent, each child octant may be empty, full or decomposed into another eight octants. This process is recursively continued until all partially full octants are described in terms of empty or full octants, or a desired degree of resolution is reached.

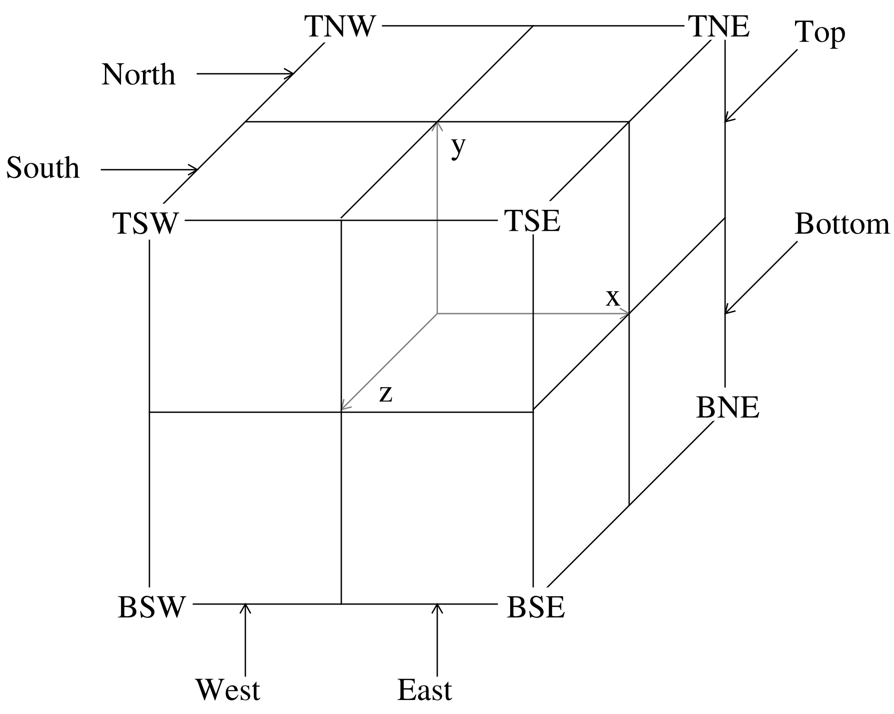

Figure 1: Prototype Octant

This hierarchical representation of a volume is thus described as an *octree*. In octree jargon, *voxel* (for "volume element") refers to an octant of the smallest resolution, while an octant is any size. An octant requires the same amount of computer memory no matter what physical size it is representing. Because of this, it does not necessarily take large amounts of memory to represent large objects.

An example of a four voxel object is shown in figure 2.

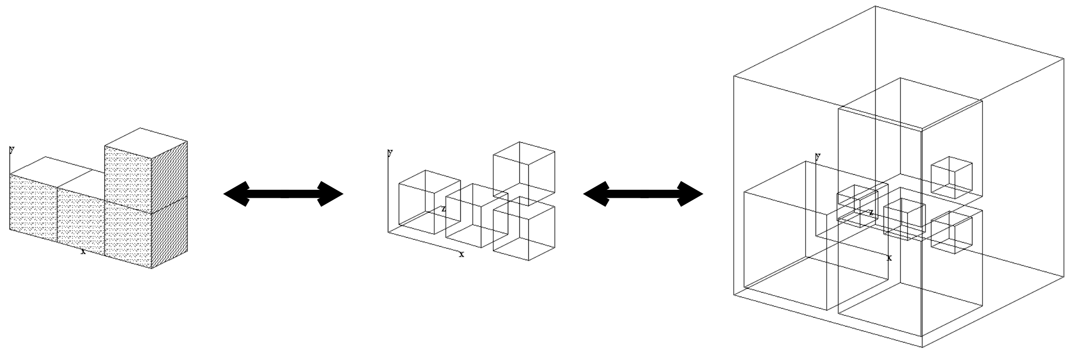

Figure 2: Object of four voxels represented by an octree.

The basic octree is often supplemented with extra information to represent the surface. For example, typical implementations use six bits in each octant to flag whether any of the octant's six faces is on the surface of the volume. More face information (such as color) requires a set of up to six indices or pointers to an auxiliary face structure.

2. Typical uses and limitations

Octrees are excellent at modeling 3D spaces which contain *blobs* or volumes that are highly connected. For example, a human figure is highly connected. The surface area has a low fractal dimension and the space around the human is uninteresting. An octree can capture the essence of the shape without expending any storage modeling the empty space surrounding it or the full space within.

One example of where this is useful is in collision detection, such as might be used when computing robot paths. When planning a path, the robot is only interested in surfaces that it has to avoid, not what is inside or outside the surface. Octrees allow economic storage and efficient probing of the surface to whatever level of resolution is required.

Each of these applications uses octrees as a static model. In particular, the model is derived from another representation such as a camera image or a CSG model. Each of these is static. For example, when the object being modeled changes, the old octree is discarded, a new camera image is taken, and from it, a new octree is generated.

Our objects of interest are not derived from a real object or prior model. Rather, the octree is the only instance of the object being studied. *The object being modeled does not physically exist, nor is it represented by any other model.* Each step of our simulation operates an an old octree description, producing a new octree description. Rather than producing a new octree from scratch, the old octree is modified at each step to be the new octree. This can be done very efficiently, allowing us to modify dynamic surfaces.

One potential problem with the simulation is that surfaces can grow to be arbitrarily large. Unlike octrees that are generated from a static representation, there is no way of knowing the largest bounding box at the start. The ordinary octree implementation begins with a single octant that represents the total working volume. The octant is divided as objects of importance are discovered in the workspace. This is carried out recursively down to whatever resolution is demanded.

Unfortunately, this scheme does not allow for modeling of objects that grow, or objects that appear outside the original working volume. In fact, both of these examples arise frequently. The first appears in our own application, which is described later in the paper. A second example is illustrated by a mobile robot exploring a world, without a priori bounds. This raises the first difference in our octree implementation – modeling objects which grow arbitrarily large. The next section discusses our solution to this problem.

3. Octree Expansion and Contraction

A solution to objects which appear outside the root octant is to enlarge the root octant so that it encompasses the new data. To remain compatible with the octree implementation, a new root octant is created representing a volume eight times larger than the old root octant. The old root octant becomes a child of the new root. A second child is used to store the new data. The remaining six children are marked **empty**. If the new object still falls outside of the octant described by the new root, this process is repeated. When the new root is large enough, a second child is selected to store the new object which fell outside of the old root octant.

Each expansion of the octree causes it to model a space eight times as large as before at the cost of an additional octant. (The six empty children octants take no actual space in memory.) The new object is modeled using the same amount of space it would have taken, had it been modeled statically.

An inverse operation shrinks the working volume of the octree to an eighth. This occurs when the object being modeled fits entirely within one child of the root. Then, the old root is discarded and the remaining root child of interest becomes the new root. This operation is repeated until the root has at least two non-empty children.

It should be apparent that expansion or shrinkage of the octree in the manner described results in a new octree. This is important, as the fact that it is an octree means it can be manipulated as before. In particular, leaf nodes can be split to get better resolution (as in a typical octree) or joined, and the root octant can be expanded or shrunk.

4. Surface Expansion and Contraction

The second difference in our implementation is caused by changes to the surface of the object being represented. In our application, changes were heavily localized. Simplistically, one can imagine a voxel along the surface being either added or deleted. Thus, we needed to maintain the octree data structure in the face of such changes.

Maintenance of the octree volume and surface differs depending upon whether the volume is expanding (by creating a voxel) or shrinking (by deleting a voxel).

Voxel Create

The first step is to create the voxel itself. If the voxel is located outside the root octant, the root is expanded as described above. Then the tree is traversed to find the location of the voxel, splitting the octant as required, as is normally done when growing an octree.

Each voxel on the object surface describes the existence and orientation of its faces. This is done by manipulating a set of six pointers to face objects. Neighboring voxels "give up" their exposed faces. For example, a voxel above the new voxel (being created) gives up its bottom face since it will be covered by the new voxel. This operation can be carried out for each face of the new voxel, however "face passing" (passing the face object pointer around the voxel) allows some improvement in speed. For example, rather than releasing the bottom face of the top voxel, the program first checks if there is a voxel below the new voxel. If there is none, the new voxel may "adopt" the bottom-pointing face. Even if there is a voxel in that position, the new voxel may adopt a face in other positions with slightly higher cost in updating the structure.

Once a neighboring voxel has given up its exposed face, it then checks to see whether it is still part of the surface (i.e., if any other faces are exposed). If no part of the voxel is on the surface, the parent checks if this was the last child on the surface. If so, the parent discards all the children. This process recurses to the root of the tree.

Voxel Delete

When deleting voxels, the first step is to create new faces for the neighbors where they border the old voxel (being deleted). For example, if a voxel exists above the old voxel it acquires a bottom face. Optimizations, similar to that during voxel creation, take place during this process.

Before a neighboring voxel can receive a face, it may have to be fully instantiated. That is, voxels that do not border the surface are not necessarily stored explicitly. In particular, a voxel that is one of eight others that share the same parent will be implicitly stored simply by marking the parent as **full**. All voxels on the surface must be stored explicitly in memory. Creation of these surface voxels proceeds in the same way as with any octree.

The last step is to delete the old voxel itself. Finally, the parent checks if this was its last child. If so, the parent itself is released, and this process recurses to the root of the tree. Another pass is made back down, or until an octant with more than one child is encountered. Each single-child root octant is discarded, causing shrinkage of the octree working volume.

In our simulation, surface change was highly localized. Some researchers have studied how to modify octrees for global changes. In particular, [Hong] describes an algorithm which performs an arbitrary translation and rotation, producing a new octree representing the result of applying the transformation to the entire octree.

5. Implementation

Our octree implementation is not unlike a typical octree system. It is written in 1200 source lines of the C language. Since the program runs on different machines with different data sizes, it is impossible to give a simple answer to the question of how much space an octant takes. (Nonetheless, a more comprehensive description of the program and its performance in actual simulations is described in great detail by [Libes].) Functionally, all an octant contains are pointers to children. For simplicity in coding, we have augmented that with a parent pointer, a single coordinate that anchors one corner of the octant in space, and a count of the number of children in the current octant.

Implementations commonly improve efficiency by having each octant contain a six bit field describing which faces exist. Our problem required the ability to choose a random face on the surface. To perform this, we stored face information in a separate array (which we could randomly choose from) and added (a pointer to) up to six face pointers to an octant. (Hence, the need for the "face passing" mentioned in §4.)

To support the features we mentioned earlier, very few changes to the octree data structures are required. Indeed, the only one is that the root octant and its size must be variable, while normally it can be considered constant.

The remaining changes are in the algorithms. A routine (for convenience, we will refer to it as **oct_find**) takes an xyz location and returns the smallest enclosing octant. **oct_find**'s basic operation is to start at the root of the octree and decide which of eight child octants contains the coordinate. This is carried out recursively, until a leaf octant is encountered.

oct_find actually chooses one of the eight octants by dividing each dimension in half. For example, if the current octant was centered at 2 on the x-axis, we would narrow the choice of eight octants to four, by comparing the given x with 2. Comparing against the other two dimensions would enable the selection of a single child. An additional check in each dimension was added to determine if the value was outside the current octant. If so, we would continue searching in the parent of the current octant. If the parent was the root, we would expand the tree (as described earlier) and continue the search at the new root.

While one might expect that it was sufficient to check for exceeding bounds only at the root, checking at every level allowed us to start searches anywhere in the octree structure. This was essential to the second feature of our implementation.

Localized changes could be made to random parts of the surface of the octree with minimal effort. A surface octant was selected for modification – for example, deletion. It was then deleted as discussed earlier. One important step is that when a voxel is deleted, it causes faces to appear on neighboring voxels. Thus it was necessary to find the six neighbors of a voxel. **oct_find** would find these six neighbors using the same algorithm as before, but searches were started from the deleted voxel's parent rather than the root. Since half of the neighbors share the same parent, these neighbors were found in one step. The remaining neighbors were found in O(n) time, where n is the depth of the octree.

All of these elaborations on the typical octree model take place with minimal overhead. In particular, expansion and contraction of the root octant occur so infrequently as to be negligible. For example, a surface that grows from a single voxel to a long string of 2^{60} voxels will only cause octree expansion to occur sixty times. Managing the faces and neighboring octants upon voxel creation/deletion require O(n) time where n is the depth of the octree. These run-time characteristics provided us with a very fast simulation.

6. Extensions for RS/MC

Octrees lend themselves well to studying Random Surfaces (RS) on the lattice. The structure studied by this problem is a surface topologically equivalent to a sphere, embedded in the lattice. This means that the surface is composed of faces that are each one unit square. Physicists call such faces *plaquettes*. The vertices of a plaquette lay on integral lattice points and are always perpendicular to an axis. Thus, a plaquette is exactly the face of a unit-sized octant. This allowed a convenient bound of one, as the minimal octant size, thereby avoiding any floating-point computations during octree maintenance.

It was, therefore, always possible to generate an exact model using octrees with little effort. This stands in contrast to more typical octree modeling (such as in a CAD system), which virtually never models the surface of interest exactly. We expect to apply octrees to an entire class of related physics problems with similar degrees of success.

Our experiments used the Monte Carlo (MC) technique of randomly choosing a point on the surface either to grow or shrink the surface. In order to perform a "random choice", an array was maintained, each member of which pointed to a face on the surface. This auxiliary data structure combined with the octree, was all that was needed to simulate RS/MC.

One *Monte Carlo step* of the simulation can now be described as follows.

1) Randomly choose a place to grow/delete a voxel on the surface.

2) Retrieve neighbors of the new (for create) or old (for delete) voxel.

3) Verify legality of the operation according to various topological and physical rules.

4) Perform the operation on the new or old voxel.

5) Fix up any neighbors affected.

Each of these can be intuited from other parts of the paper except for step 3. This step consists of rules defining the essence of the object being simulated. For example, two of the rules are as follows:

i) Operations resulting in sharing only one edge by any two voxels are illegal.

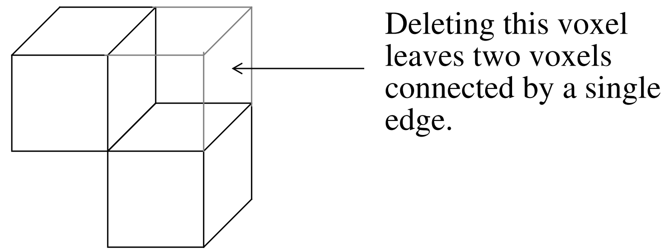

Figure 3: Illegal deletion of old voxel. Restricted by rule i.

This restriction is enforced by examining the twelve sets of adjacent neighbors. For example, if the east and south neighbors do not exist, the east-south neighbor is retrieved from the octree. If it exists, the operation is rejected, otherwise the operation is accepted.

ii) Operations resulting in a hole in the solid are rejected.

This restriction is enforced by examining the six neighbors in several ways. For example, if we are creating a voxel, but only its top and bottom neighbor exists (see figure 4), the operation is rejected because it would create a hole in the volume.

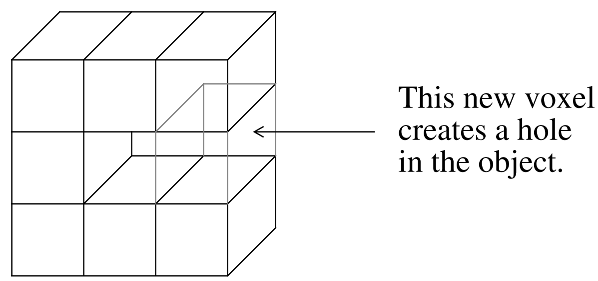

Figure 4: Illegal creation of new voxel. Restricted by rule ii.

Notice that (i) and (ii) can be implemented by boolean operations. The implementation rejects operations as soon as possible, so no logically unnecessary checks are made. The cost of both (i) and (ii) is O(1).

These topological properties can be efficiently verified using octrees. Other disposable representations only allow efficient verification of local properties when neighbors are strongly-connected. For example,determining if a newly created object shares an single edge with another object (violating rule i) with only a boundary representation, potentially requires the entire objected be traversed.

More detail on these rules is given in [Libes]. Figure 5 is a surface generated by running the simulation for 500,000 Monte Carlo steps. The surface size was 3474.

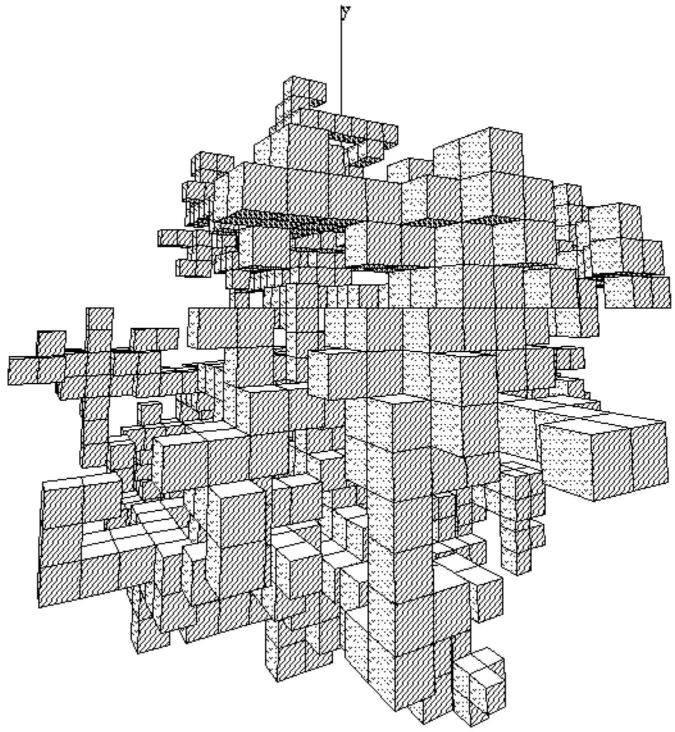

Figure 5: After 500,000 Monte Carlo steps, surface size is 3474 (and still growing).

Prior implementations of RS/MC [Sterling] [Glaus] used a fixed-size array of memory in one-to-one correspondence, with the lattice being modeled. This technique was fast, but drastically limited the size of the simulation that could be performed. By moving to an octree-based representation, the space required was changed from $O(n^3)$ to $O(|S|)$ where n was the size of one dimension of the bounding box and $|S|$ was the number of faces of the surface. The run-time changed from $O(1)$ per step to $O(\log n)$. This was a very favorable trade-off simply because n^3 grew so much faster than $|S|$. This is discussed further in [Libes].

7. Conclusion

We have suggested a new technique for adapting octrees to modeling dynamic surfaces. In particular, octrees can be extended to model arbitrarily large and growing surfaces, which do not have a priori bounds. The salient feature of octrees of $O(\log n)$ performance in time and $O(|S|)$ performance in space, is retained by this enhanced octree.

We expect that this enhancement to octrees will allow them to model objects that they formerly could not, such as autonomous robots exploring unknown worlds. We have also shown the applicability of this data structure to the study of objects embedded in the lattice, for which octrees provide an exact model and efficient verification of topological constraints.

8. Credits

This work was partially supported by funding from the Scientific Computing Division of the U.S. Department of Energy.

9. References

Glaus, U., "Monte Carlo Study of Self-Avoiding Surfaces", Dept. of Physics, Clarkson University, Potsdam, NY 13676.

Hong, Tsai-Hong, and Shneier, Michael O., "Rotation and Translation of Objects Represented by Octrees", *Proceedings of the IEEE International Conference on Robotics and Automation*, Raleigh, NC, March 31 – April 3, 1987.

Libes, Don, and Sullivan, Francis, "Modeling Self-Avoiding Random Surfaces with Octrees", National Institute of Standards and Technology, Gaithersburg, MD, October 1990.

Meagher, Donald, "Octree Generation, Analysis and Manipulation", IPL-TR-027, Image Processing Laboratory, Rensselaer Polytechnic Institute, Troy, New York, April 1982.

Samet, Hanan, "The Quadtree and Related Hierarchical Data Structures", *ACM Computing Surveys 16*, 2(June 1984), 187-260.

Sterling, T., and Greensite, J., "Entropy of Self-Avoiding Surfaces on the Lattice", *Physics Letters*, Volume 121B, Number 5, 10 February 1983.