Simple and Efficient Traversal Methods for Quadtrees and Octrees

Sarah F. Frisken and Ronald N. Perry
Mitsubishi Electric Research Laboratories

Abstract. Quadtrees and octrees are used extensively throughout computer graphics and in many other diverse fields such as computer vision, robotics, and pattern recognition. Managing information stored in quadtrees and octrees requires basic tree traversal operations such as point location, region location, and neighbor searches. This paper presents simple and efficient methods for performing these operations that are inherently non-recursive and reduce the number of comparisons with poor predictive behavior. The methods are table-free, thereby reducing memory accesses, and generalize easily to higher dimensions.

1. Introduction

Quadtrees and octrees are spatial data structures that successively partition a region of space into 4 or 8 equally sized quadrants or octants (i.e., cells). Starting from a root cell, cells are successively subdivided into smaller cells under certain conditions, such as when a cell contains an object boundary (e.g., region quadtree) or when a cell contains more than a specified number of objects (e.g., point quadtree). Compared to methods that do not partition space or that partition space uniformly, quadtrees and octrees can reduce the amount of memory required to represent objects (e.g., an image) and improve execution times for querying and processing data (e.g., collision detection).

Quadtrees and octrees can be represented in a hierarchical tree structure composed of a root cell, intermediate cells, and leaf cells, or in a pointerless representation such as a linear quadtree, which stores only leaf cells in a cell list. In tree-based representations, each cell stores pointers to its parent and child cells. Operations such as point location (finding the leaf cell containing a given point) and neighbor searches (finding a cell in a specified direction that touches a given cell) traverse the tree by following these pointers. In linear quadtrees, each cell has an associated *locational code*, which acts as a search key to locate the cell in the cell list. Traditionally, these locational codes interleave bits that comprise values of the cell's minimum (x, y, z, etc.) coordinates such that quadtrees use locational codes of base 4 (or 5 if a "don't care" directional code is used) and octrees use locational codes of base 8 (or 9) (see [Samet 90b]). In general, linear quadtrees are more compact than tree-based representations at the expense of more costly or complicated processing methods.

Managing information stored in a quadtree or octree requires basic operations such as point location, region location (finding the smallest cell or set of cells that encloses a specified region), and neighbor searches. Traditional methods for point location in a tree-based representation require a downward branching through the tree from the root node, where each branch is made by comparing a point's position to the midplane positions of the current enclosing cell. In linear quadtrees, point location is performed by encoding a point's position into a locational code and then searching the (ordered) cell list to find the cell whose locational code best matches that of the point position. Traditional neighbor searching in a tree-based representation (e.g., [Samet 90b]) requires a recursive upward branching from the given cell to the smallest common ancestor of the cell and its neighbor and then a recursive downward branching to locate the neighbor. Each branch in the recursion relies on comparing values (derived from tables) that depend on the current cell and its parent. Neighbor searches in linear quadtrees can be performed in a similar manner [Samet 90b].

As discussed in [Knuth 98, Knuth 99], the comparisons required for point location, region location, and neighbor searching can be costly because the predictive branching strategies used for modern CPUs will stall the instruction pipeline whenever branch instructions are incorrectly predicted. Such mispredictions occur frequently during tree traversal because, unlike most loops, the branch chosen in a previous iteration has no relevance to the likelihood of it being taken again [Pritchard 2001]. In addition,

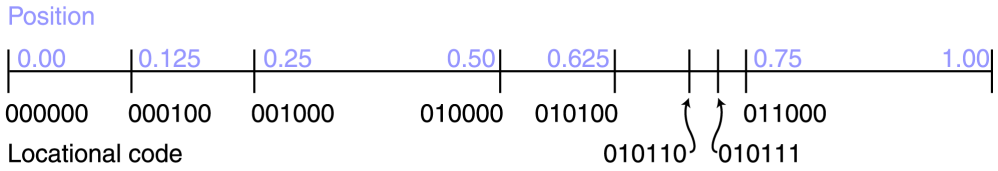

Figure 1. A 1D spatial partitioning over [0,1] showing the relationship between the position of the cell's left corner and the cell's locational code.

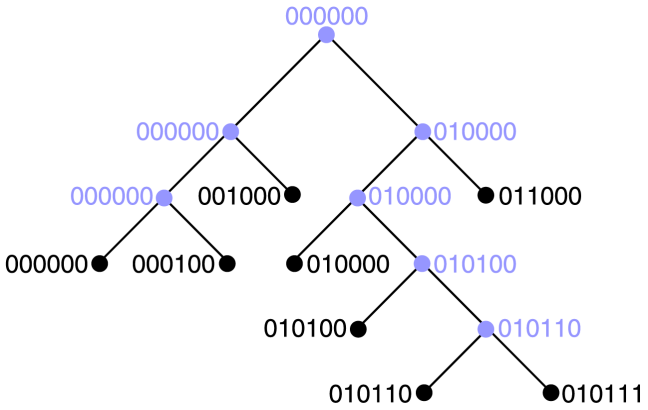

Figure 2. The same spatial partitioning of Figure 1 represented as a binary tree showing the relationship between a cell's locational code and the branching pattern from the root cell to the cell.

traditional neighbor searching methods are recursive (thus introducing overhead from maintaining stack frames and making function calls), table-based (thus requiring costly memory accesses in typical applications), and difficult to extend to higher dimensions. Here we present simple and efficient tree traversal methods for tree-based representations of quadtrees, octrees and their higher dimensional counterparts that use locational codes and which are inherently non-recursive, reduce or eliminate the number of comparisons with poor predictive behavior, and are table-free. We provide details of the representation, algorithms for point location, region location, and neighbor searching, and C code that performs these operations. While we focus on quadtrees, extension to octrees and higher dimensional trees is trivial because these methods treat each dimension independently.

2. Representation

We assume a standard tree-based quadtree representation of a cell hierarchy branching from a root cell, through intermediate cells, to leaf cells. Each cell contains a pointer to its parent cell (null for the root cell), a pointer to the first of 4 consecutively stored child cells (null for leaf cells), and (typically) an application-specific field for cell data (e.g., a cell type for region quadtrees or object indices for point quadtrees). The depth of the quadtree is limited to *N_LEVELS* for practical purposes. Similar to [Samet 90a], we define the level of the root cell to be $ROOT\_LEVEL \equiv (N\_LEVELS - 1)$, and the level of the smallest possible cell in the tree to be 0. The quadtree is defined over *[0,1]x[0,1]*, which can be achieved for a general rectangular region by applying an affine transformation of the coordinate space.

A locational code is associated with each coordinate of a cell's minimum vertex and is represented in binary form in a data field with bit size greater than or equal to *N_LEVELS*. Unlike traditional locational codes, we do not interleave bits for x and y coordinates into a single value and instead use independent locational codes for each coordinate. The bits in a locational code are numbered from right (LSB) to left starting from 0. Each bit in a locational code indicates the branching pattern at the corresponding level of the tree, i.e., bit *k* represents the branching pattern at level *k* in the tree.

Figures 1 and 2 illustrate how locational codes are determined for a "bi-tree", the 1D equivalent of a quadtree. Figure 1 is a spatial representation of the bi-tree which is defined over *[0,1]*. The bi-tree has 6 levels (i.e., *N_LEVELS = 6*) and the level of the root cell is 5 (i.e., *ROOT_LEVEL = 5*). Figure 2 illustrates the same spatial partitioning with a tree representation.

The locational code for each cell can be determined in two ways. The first method multiplies the position of the cell's left corner by 2^{ROOT_LEVEL} (i.e., $2^5 = 32$) and then represents the product in binary form. For example, the cell *[0.25,0.5)* has locational code *binary(0.25 * 32) = binary(8) = 001000*. The second method follows the branching pattern from the root cell to the cell in question, setting each bit according to the branching pattern of the corresponding level. Starting by setting bit *ROOT_LEVEL* to 0, the method then sets each subsequent bit, *k*, to 0 if the branching pattern from level *k + 1* to *k* branches to the left and to 1 if it branches to the right. All lower order bits that are not set because a leaf cell is reached are set to 0. In quadtrees (octrees, etc.), locational codes for each dimension are computed independently from the x, y, (z, etc.) positions of the left, bottom, (back, etc.) cell corner (first method) or from the left-right, bottom-top, (back-front, etc.) branching patterns used to reach the cell in question from the root cell (second method).

Several properties of these locational codes can be used to devise simple and efficient tree traversal methods. First, just as locational codes can be determined from branching patterns, branching patterns can be determined from locational codes. That is, a cell's locational code can be used to traverse the tree (without costly comparisons) from the root cell to the cell by using the appropriate bit in the x, y, (z, etc.) locational codes to index the corresponding child of each intermediate cell. Second, the position of any point in *[0,1)x[0,1)* ($[0,1)^3$, etc.) can be converted into 2 (3, etc.) locational codes by applying the first method for determining cell locational codes to each coordinate of the point. These first two properties enable efficient point and region location as described in Sections 3 and 4. Finally, the locational codes of any neighbor of a cell can be determined by adding and subtracting bit patterns to the cell's locational codes. This property is used in Section 5 to provide simple, non-recursive, table-free neighbor searches that generalize easily to higher dimensions. This property is also used in Section 6 to provide an efficient, bottom-up method for determining leaf cells along a ray intersecting an octree (an operation required for ray tracing objects stored in an octree).

3. Point location

For point location (i.e., determining a leaf cell that contains a given point) we assume that the quadtree represents the region *[0,1]x[0,1]* and the point lies in *[0,1)x[0,1)*. The first step converts the point's x and y coordinate values to x and y locational codes by multiplying each coordinate value by 2^{ROOT_LEVEL}, truncating the resultant products to integer types, and representing the integers in binary form. Assuming that floating point numbers are represented using the IEEE standard, this conversion to integers can be performed very efficiently [King 2001]. The second step starts at the root cell, and at each level *k* in the tree, uses the *(k-1)st* bit from each of the x, y, (z, etc.) locational codes to determine an index to the appropriate child cell. Note that either all of the cell's 4 (8, etc.) children or all of the pointers to these children are stored consecutively and are consistently ordered to enable this indexing. When the indexed child cell has no children, the desired leaf cell has been reached and the operation is complete.

As an example, in the bi-tree of Figures 1 and 2, the point at *x = 0.55* has locational code *binary(trunc(0.55 * 32)) = binary(17) = 010001*. Following the branching pattern of the point's locational code from the root cell in Figure 2 (i.e., branching first right, then left, then left where a leaf cell is reached), the point is located in the leaf cell *[0.5,0.625)* which has locational code *010000*. C code for point location in quadtrees using this method is provided in Appendix A.

Like traditional methods for point location, this method requires a comparison at each level of the tree to test if the current cell is a leaf cell. For example, in order to locate a point in a level 0 cell of an 8-level octree, both methods require 8 of these simple leaf-cell tests. However, unlike traditional methods, comparisons between the point position and midplane positions of each cell are *not* needed at each branching point. This saves *d* comparisons at each level in the tree traversal, where *d* is the dimension of the tree (e.g., 2 for quadtrees, 3 for octrees, etc.). Therefore, in order to locate a point in a level 0 cell of an

8-level octree, traditional methods require an additional *24 (= 3 * 8)* comparisons to branch to the appropriate children of intermediate cells. These additional comparisons in traditional methods exhibit the poor predictive behavior described by [Pritchard 2001].

4. Region Location

Region location involves finding the smallest cell or set of cells that encloses a given region. Here we assume that the region is a rectangular, axis-aligned bounding box and present a method for finding the single smallest cell entirely enclosing the region.

[Pritchard 2001] recently presented a method for region location in quadtrees that uses binary representations (i.e., locational codes) of the *x* and *y* boundaries of the region bounding box. However, his method assumes a quadtree representation that is not truly tree-based. Instead, his quadtrees are represented by a hierarchy of regular arrays of cells, where each level is fully subdivided and contains 4 times as many cells as the previous level. This representation alleviates the need to store pointers in each cell and allows simple indexing for cell access at each level. However, it requires significantly more memory than tree-based quadtrees (and even more for octrees) and is hence impractical for many applications. Pritchard's method has two steps: it first uses the locational codes of the left and right x boundaries and the top and bottom y boundaries of the region bounding box to determine the level of the enclosing cell and then uses a scaled version of the position of the bottom-left vertex of the region bounding box to index into the regular array at this level. We provide a method for region location in tree-based quadtrees, octrees, and their higher dimensional counterparts which first determines the size of the smallest possible enclosing cell using a method similar to Pritchard's first step, and then uses a variation of the point location method presented above to traverse the tree from the root cell to the smallest enclosing cell.

Similar to Pritchard, we determine the size (i.e., level) of the smallest possible enclosing cell by *XOR'ing* the left and right *x* locational codes and the top and bottom *y* locational codes of the left-right and top-bottom vertices of the region. These two *XOR'ed* codes are then searched from the left (MSB) to find the first 1 bit of the two codes, indicating the first level below the root level where at least one of the pairs of locational codes differ. The level of the smallest possible enclosing cell is then equal to the bit number of the 0 bit immediately preceding this 1 bit. Given this level, our method then traverses the tree downward from the root cell following the bit pattern of the locational codes of any point in the region (e.g., the bottom-left corner) until a leaf cell is encountered or a cell of the determined size is reached. This yields the desired enclosing cell. Note that there are several methods for identifying the highest order 1 bit in the *XOR'ed* values ranging from a simple shift loop to platform specific single instructions which bit-scan a value, thereby eliminating the loop and subsequent comparisons.

As a first example, using the bi-tree of Figures 1 and 2, the region *[0.31,0.65)* has left and right locational codes *001001* and *010101*, respectively. *XOR'ing* these locational codes yields *011100*, with the first 1 bit from the left (MSB) encountered at bit 4, so that the level of the smallest possible enclosing cell is 5, i.e., the enclosing cell of the region *[0.31,0.65)* is the root cell. As a second example, the region *[0.31, 0.36)* has locational codes *001001* and *001010*. The *XOR* step yields *000011*, with the first 1 bit from the left encountered at bit 1, so that the level of the smallest possible enclosing cell is 2. The smallest enclosing cell can then be found by traversing the bi-tree downward from the root cell following the left locational code, *001001*, until a leaf cell is encountered or a level 2 cell is reached. For the spatial partitioning of Figure 1, the enclosing cell is the level 3 leaf cell with locational code *001000*. C code for this region location method applied to quadtrees is presented in Appendix A.

5. Neighbor searches

Neighbor searching finds a cell in a specified direction (e.g., left, top, top-left, etc.) that touches a given cell. Several variations exist, including finding vertex, or edge, (or face, etc.) neighbors, finding neighbors of the same size or larger than the given cell, or finding all of the leaf cell neighbors of the given cell.

In order to determine neighbors of a given cell, we first note that the bit patterns of the locational codes of two neighboring cells differ by the binary distance between the two cells. For example, we know that the

left boundary of every right neighbor of a cell (including intermediate and leaf cells) is offset from the cell's left boundary by the size of the cell. Hence, the x locational code of every right neighbor of a cell can be determined by adding the binary form of the cell's size to the cell's x locational code. Fortunately, the binary form of a cell's size is easily determined from its level, i.e., $cellSize \equiv binary(2^{cellLevel})$. Hence, the x locational code for a cell's right neighbor is the sum of the cell's x locational code and *cellSize*.

As an example, the cell *[0.25,0.5)* of the bi-tree in Figures 1 and 2 has locational code *001000* and level 3. Hence, the x locational code of a neighbor touching its right boundary is $001000 + binary(2^3) = 001000 + 001000 = 010000$ which can be verified by inspection.

Determining the x locational codes of a cell's left neighbors is more complicated. Because we don't know the sizes of the cell's left neighbors, we don't know the correct binary offset between the cell's x locational code and the x locational codes of its left neighbors. However, we first observe that the smallest possible left neighbor has level 0 and hence the difference between the x locational code of a cell and the x locational code of the cell's smallest possible left neighbor is $binary(2^0)$ (i.e., *smallest possible left neighbor's x locational code = cell's x locational code – binary(1)*). Second, we observe that the left boundary of this smallest possible left neighbor is located between the left and right boundaries of every left neighbor of the cell (including intermediate cells). Hence, a cell's left neighbors can be located by traversing the tree downward from the root cell using the x locational code of this smallest possible left neighbor and stopping when a neighbor cell of a specified level is reached or a leaf cell is encountered.

As an example, referring to Figure 1, the smallest possible left neighbor of the cell *[0.25,0.5)* has locational code $001000 – 000001 = 000111$. Traversing the tree downwards from the root cell using this locational code and stopping when a leaf cell is reached yields the cell *[0.125,0.25)* with locational code *000100* as the cell's left neighbor.

For quadtrees (octrees, etc.), a neighbor is located by following the branching patterns of the pair (triplet, etc.) of x, y, (z, etc.) locational codes to the neighbor until a leaf cell is encountered or a specified maximum tree traversal level is reached. Each of these locational codes is determined from the appropriate cell boundary (which is determined from the specified direction to the neighbor). In a quadtree, the x locational code of a right edge neighbor is determined from the cell's right boundary and the x and y locational codes of a top-right vertex neighbor are determined from the cell's top and right boundaries.

For example, the right edge neighbor of size greater than or equal to a cell is located by traversing downward from the root cell using the x locational code of the cell's right boundary and the y locational code of the cell until either a leaf cell or a cell of the same level as the given cell is reached. As a second example, a cell's bottom-left leaf cell vertex neighbor is located by traversing the tree using the x locational code of the cell's smallest possible left neighbor and the y locational code of the cell's smallest possible bottom neighbor until a leaf cell is encountered.

Once the locational codes of the desired neighbor have been determined, the desired neighbor can be found by traversing the tree downward from the root cell. However, as described in [Samet 90b], it can be more efficient to first traverse the tree upward from the given cell to the smallest common ancestor of the cell and its neighbor and then traverse the tree downward to the neighbor. Fortunately, locational codes also provide an efficient means for determining this smallest common ancestor. Assuming a bi-tree, the neighbor's locational code is determined, as described above, from the given cell's locational code and the specified direction to the neighbor. The given cell's locational code is then *XOR'ed* with the neighbor's locational code to generate a difference code. The bi-tree is then traversed upward from the given cell to the first level where the corresponding bit in the difference code is a 0 bit. This 0 bit indicates the smallest level in the bi-tree where the two locational codes are the same. The cell reached by traversing upward to this level is the smallest common ancestor.

In quadtrees (octrees, etc.), the 2 (3, etc.) locational codes of the given cell are *XOR'ed* with the corresponding locational codes of the neighbor to produce 2 (3, etc.) difference codes. The cell with the highest level reached by the upward traversal for each of these difference codes is the smallest common ancestor.

As an example, referring to Figure 1, the difference code for the level 3 cell *[0.25,0.5)* and its right neighbor is $001000 \wedge 010000 = 011000$. Traversing the tree upward from level 3 considers bits in this difference code to the left of bit 3. The first 0 bit is reached at *ROOT_LEVEL*, so the smallest common ancestor of *[0.25,0.5)* and its right neighbor is the root cell. As a second example, the difference code for

the level 3 cell *[0.75,1)* and its left neighbor is *011000 ^ 010111 = 001111*. Examining bits to the left of bit 3 yields the first 0 at bit 4, corresponding to a level 4 cell. Hence, the smallest common ancestor of the cell *[0.75,1)* and its left neighbor is the cell's parent cell *[0.5,1)* which has locational code *010000*.

Depending on the application, several different variations of neighbor searches might be required. C code is provided in Appendix A for some typical examples, i.e., finding the smallest left and right neighbors of size at least as large as the given cell and finding the three leaf cell neighbors touching the given cell's bottom-right vertex.

The main advantage of this neighbor finding method over traditional methods is its simplicity: it generalizes easily to higher dimensions and it can be applied to many variations of neighbor searches. In contrast, traditional neighbor searches require different methods for face, edge, and vertex neighbors and "vertex neighbors are considerably more complex" [Samet 90b]. Our method trades off traditional table lookups for simple register-based computations in the form of bit manipulations. This is advantageous in current architectures where CPU speeds far exceed memory speeds. Note that if the working set of an application is fortunate enough to be cache resident, table lookups are virtually free. However, in many practical applications, the application data and the table data compete for the cache, forcing frequent reloading of the table data from memory. In addition, while [Samet 90b] provides tables for some varieties of neighbor searching, neither he nor the literature provides them for all common variations. Furthermore, tables are specialized for a given cell enumeration and must be re-determined for different cell labeling conventions. Our experience has been that determining these tables requires good spatial intuition and hence they can be error prone, are tedious to verify, and are extremely difficult to generate in 4 dimensions and beyond (where there are numerous applications in fields such as computer vision, scientific visualization, and color science). Finally, our neighbor searching method is inherently non-recursive and requires fewer comparisons than traditional methods. In contrast, traditional methods for neighbor searching are inherently recursive and unraveling the recursion is non-trivial. [Bhattacharya 2001] recently presented a non-recursive neighbor searching method for quadtrees and octrees limited to finding neighbors of the same size or larger than a cell. However, like Samet's, his method requires table-based tree traversal to determine the appropriate neighbor, and both methods require a similar number of comparisons.

6. Ray Traversal

Ray tracing methods often make use of octrees to accelerate tracing rays through large empty regions of space and to minimize the time spent computing ray-surface intersections. These methods determine non-empty leaf cells along a ray passing through the octree and then process ray-surface intersections within these cells. There are two basic approaches for following a ray through an octree: bottom-up methods which start at the first leaf cell encountered by the ray and then use neighbor finding techniques to find each subsequent leaf cell along the ray, and top-down methods that start from the root cell and use a recursive procedure to find offspring leaf cells that intersect the ray. An extensive summary of the research in this area is presented in [Havran 99].

Both the top-down and bottom-up approaches can exploit the tree traversal methods described in this paper. In fact, [Stolte, Caubet 95] presents a top-down approach that uses a method similar to the point location method described in Section 3. They first locate a leaf cell containing the point where a ray enters the octree. Then, for each leaf cell without a ray-surface intersection, a 3D DDA is used to incrementally step along the ray (in increments proportional to the size of the smallest possible leaf cell) until a boundary of the leaf cell is crossed, providing a sample point in the next leaf cell along the ray. The next leaf cell is then found by popping cells from a recursion stack to locate a common ancestor of the leaf cell and the next leaf cell and then traversing down the octree using their point location method.

Stolte and Caubet's top-down method can be converted to a non-recursive bottom-up approach by using the neighbor searching techniques presented in Section 5. In addition, their method can be improved in two ways. First, the three (dimension-dependent) comparisons in their point location method can be replaced by three multiplies. Second, the use of the 3D DDA can be avoided by using the current leaf cell's size and locational codes to determine one or more of the new locational codes of the next leaf cell directly (from which the entry point to the next leaf cell can be determined).

7. Conclusions

We have provided methods for point location, region location, and neighbor searching in quadtrees and octrees that are simple, efficient, inherently non-recursive, and reduce the number of comparisons with poor predictive behavior. The methods are table-free, thereby reducing memory accesses, and generalize easily to higher dimensions. In addition, we have provided C code for point location, region location, and 3 variations of neighbor searching. In our real-time sculpting application [Perry, Frisken 2001] we have found that these methods significantly reduce computation times during triangulation of an octree-based volumetric representation (which requires neighbor searching) and in a time-critical edit-octree/refresh-display loop (which requires point and region location).

References

[Bhattacharya 2001] P. Bhattacharya, "Efficient Neighbor Finding Algorithms in Quadtree and Octree", M.T. Thesis, Dept. Comp. Science and Eng., India Inst. Technology, Kanpur, 2001.

[Havran 99] V. Havran, "A Summary of Octree Ray Traversal Algorithms", Ray Tracing News, 12(2), pp. 11-23, 1999.

[King 2001] Y. King, "Floating-Point Tricks: Improving Performance with IEEE Floating Point", in *Game Programming Gems 2*, ed. M. DeLoura, Charles River Media, Hingham, MA., pp. 167-181, 2001.

[Knuth 98] D. Knuth, *The Art of Computer Programming*, Volume 1, Addison-Wesley, 1998.

[Knuth 99] D. Knuth, *MMIXware: A RISC Computer for the Third Millennium*, Springer-Verlag, 1999.

[Perry, Frisken 2001] R. Perry and S. Frisken, "Kizamu: A System for Sculpting Digital Characters", Proc. SIGGRAPH 2001, pp. 47-56, 2001.

[Pritchard 2001] M. Pritchard, "Direct Access Quadtree Lookup", in *Game Programming Gems 2*, ed. M. DeLoura, Charles River Media, Hingham, MA, pp. 394-401, 2001.

[Samet 90a] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[Samet 90b] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, GIS*, Addison-Wesley, Reading, MA, 1990.

[Stolte, Caubet 95] N. Stolte and R. Caubet, "Discrete Ray-Tracing of Huge Voxel Spaces", Computer Graphics Forum, 14(3), pp. 383-394, 1995.

Appendix A


```
//--------------------------------------------------------------------------
// Copyright 2002 Mitsubishi Electric Research Laboratories.
// All Rights Reserved.
//
// Permission to use, copy, modify and distribute this software and its
// documentation for educational, research and non-profit purposes, without fee,
// and without a written agreement is hereby granted, provided that the above
// copyright notice and the following three paragraphs appear in all copies.
//
// To request permission to incorporate this software into commercial products
// contact MERL - Mitsubishi Electric Research Laboratories, 201 Broadway,
// Cambridge, MA 02139.
//
// IN NO EVENT SHALL MERL BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL,
// INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING LOST PROFITS, ARISING OUT OF
// THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF MERL HAS BEEN ADVISED
// OF THE POSSIBILITY OF SUCH DAMAGES.
```

```
//
//  MERL SPECIFICALLY DISCLAIMS ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
//  THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//  THE SOFTWARE PROVIDED HEREUNDER IS ON AN "AS IS" BASIS, AND MERL HAS NO
//  OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
//  MODIFICATIONS.
//-----------------------------------------------------------------------------
//-----------------------------------------------------------------------------
// Generic quadtree cell. Note that the locational codes and the cell level are
// only used in neighbor searching; they are not necessary for point or region
// location.
//-----------------------------------------------------------------------------
typedef struct _qtCell {
   unsigned int     xLocCode; // X locational code
   unsigned int     yLocCode; // Y locational code
   unsigned int     level;    // Cell level in hierarchy (smallest cell has level 0)
   struct _qtCell   *parent;  // Pointer to parent cell
   struct _qtCell   *children;// Pointer to first of 4 contiguous child cells
   void             *data;    // Application specific cell data
}  qtCell;


//-----------------------------------------------------------------------------
// Maximum quadtree depth and related constants
//-----------------------------------------------------------------------------
#define QT_N_LEVELS     16        // Number of possible levels in the quadtree
#define QT_ROOT_LEVEL   15        // Level of root cell (QT_N_LEVELS - 1)
#define QT_MAX_VAL      32768.0f  // For converting positions to locational codes
                                  // (QT_MAX_VAL = 2^QT_ROOT_LEVEL)


//-----------------------------------------------------------------------------
// Macro to traverse a quadtree from a specified cell (typically the root cell)
// to a leaf cell by following the x and y locational codes, xLocCode and
// yLocCode. Upon entering, cell is the specified cell and nextLevel is one less
// than the level of the specified cell. Upon termination, cell is the leaf cell
// and nextLevel is one less than the level of the leaf cell.
//-----------------------------------------------------------------------------
#define QT_TRAVERSE(cell,nextLevel,xLocCode,yLocCode)                           \
{                                                                               \
   while ((cell)->children) {                                                   \
      unsigned int childBranchBit = 1 << (nextLevel);                           \
      unsigned int childIndex = ((((xLocCode) & childBranchBit) >> (nextLevel))  \
      + (((yLocCode) & childBranchBit) >> (--(nextLevel))));                    \
      (cell) = &(((cell)->children)[childIndex]);                               \
   }                                                                            \
}


//-----------------------------------------------------------------------------
// Macro to traverse a quadtree from a specified cell to an offspring cell by
// following the x and y locational codes, xLocCode and yLocCode. The offpring
// cell is either at a specified level or is a leaf cell if a leaf cell is
// reached before the specified level. Upon entering, cell is the specified
// cell and nextLevel is one less than the level of the specified cell. Upon
// termination, cell is the offspring cell and nextLevel is one less than the
// level of the offspring cell.
//-----------------------------------------------------------------------------
#define QT_TRAVERSE_TO_LEVEL(cell,nextLevel,xLocCode,yLocCode,level)            \
{                                                                               \
   unsigned int n = (nextLevel) - (level) + 1;                                  \
   while (n--) {                                                                \
      unsigned int childBranchBit = 1 << (nextLevel);                           \
      unsigned int childIndex = ((((xLocCode) & childBranchBit) >> (nextLevel))  \
      + (((yLocCode) & childBranchBit) >> (--(nextLevel))));                    \
      (cell) = &(((cell)->children)[childIndex]);                               \
      if (!(cell)->children) break;                                             \
   }                                                                            \
}


//-----------------------------------------------------------------------------
// Macro for traversing a quadtree to a common ancestor of a specified cell
// and its neighbor, whose x or y locational code differs from the cell's
// corresponding x or y locational code by binaryDiff (determined by XOR'ing the
// appropriate pair of x or y locational codes). Upon entering, cell is the
// specified cell and cellLevel is the cell's level. Upon termination, cell is
// the common ancestor and cellLevel is the common ancestor's level.
//-----------------------------------------------------------------------------
#define QT_GET_COMMON_ANCESTOR(cell,cellLevel,binaryDiff)                       \
{                                                                               \
   while ((binaryDiff) & (1 << (cellLevel))) {                                  \
      (cell) = (cell)->parent;                                                  \
      (cellLevel)++;                                                            \
   }                                                                            \
}
```

```
//--------------------------------------------------------------------------------
// Locate the leaf cell containing the specified point p, where p lies in
//  [0,1)x[0,1).
//--------------------------------------------------------------------------------
qtCell *qtLocateCell (qtCell *root, float p[2])
{
   //----Determine the x and y locational codes of the point's position. Refer
   //----to [King2001] for more efficient methods for converting floating point
   //----numbers to integers.
   unsigned int xLocCode = (unsigned int) (p[0] * QT_MAX_VAL);
   unsigned int yLocCode = (unsigned int) (p[1] * QT_MAX_VAL);


   //----Follow the branching patterns of the locational codes from the root cell
   //----to locate the leaf cell containing p
   qtCell *cell = root;
   unsigned int nextLevel = QT_ROOT_LEVEL - 1;
   QT_TRAVERSE(cell,nextLevel,xLocCode,yLocCode);
   return(cell);
}


//--------------------------------------------------------------------------------
// Locate the smallest cell that entirely contains a rectangular region defined
// by its bottom-left vertex v0 and its top-right vertex v1, where v0 and v1
// lie in [0,1)x[0,1).
//--------------------------------------------------------------------------------
qtCell *qtLocateRegion (qtCell *root, float v0[2], float v1[2])
{
   //----Determine the x and y locational codes of the region boundaries. Refer
   //----to [King2001] for more efficient methods for converting floating point
   //----numbers to integers.
   unsigned int x0LocCode = (unsigned int) (v0[0] * QT_MAX_VAL);
   unsigned int y0LocCode = (unsigned int) (v0[1] * QT_MAX_VAL);
   unsigned int x1LocCode = (unsigned int) (v1[0] * QT_MAX_VAL);
   unsigned int y1LocCode = (unsigned int) (v1[1] * QT_MAX_VAL);


   //----Determine the XOR'ed pairs of locational codes of the region boundaries
   unsigned int xDiff = x0LocCode ^ x1LocCode;
   unsigned int yDiff = y0LocCode ^ y1LocCode;


   //----Determine the level of the smallest possible cell entirely containing
   //----the region
   qtCell *cell = root;
   unsigned int level = QT_ROOT_LEVEL;
   unsigned int minLevel = QT_ROOT_LEVEL;
   while (!(xDiff & (1 << level)) && level) level--;
   while (!(yDiff & (1 << minLevel)) && (minLevel > level)) minLevel--;
   minLevel++;


   //----Follow the branching patterns of the locational codes of v0 from the
   //----root cell to the smallest cell entirely containing the region
   level = QT_ROOT_LEVEL - 1;
   QT_TRAVERSE_TO_LEVEL(cell,level,x0LocCode,y0LocCode,minLevel);
   return(cell);
}


//--------------------------------------------------------------------------------
// Locate the left edge neighbor of the same size or larger than a specified
// cell. A null pointer is returned if no such neighbor exists.
//--------------------------------------------------------------------------------
qtCell *qtLocateLeftNeighbor (qtCell *cell)
{
   //----No left neighbor if this is the left side of the quadtree
   if (cell->xLocCode == 0) return(0);
   else {
      //----Get cell's x and y locational codes and the x locational code of the
      //----cell's smallest possible left neighbor
      unsigned int xLocCode = cell->xLocCode;
      unsigned int yLocCode = cell->yLocCode;
      unsigned int xLeftLocCode = xLocCode - 0x00000001;


      //----Determine the smallest common ancestor of the cell and the cell's
      //----smallest possible left neighbor
      unsigned int cellLevel, nextLevel;
      unsigned int diff = xLocCode ^ xLeftLocCode;
      qtCell *pCell = cell;
      cellLevel = nextLevel = cell->level;
      QT_GET_COMMON_ANCESTOR(pCell,nextLevel,diff);


      //----Start from the smallest common ancestor and follow the branching
```

```
        //----patterns of the locational codes downward to the smallest left
        //----neighbor of size greater than or equal to cell
        nextLevel--;
        QT_TRAVERSE_TO_LEVEL(pCell,nextLevel,xLeftLocCode,yLocCode,cellLevel);
        return(pCell);
    }
}


//------------------------------------------------------------------------------
// Locate the right edge neighbor of the same size or larger than a specified
// cell. A null pointer is returned if no such neighbor exists.
//------------------------------------------------------------------------------
qtCell *qtLocateRightNeighbor (qtCell *cell)
{
    //----No right neighbor if this is the right side of the quadtree
    unsigned int binaryCellSize = 1 << cell->level;
    if ((cell->xLocCode + binaryCellSize) >= (1 << QT_ROOT_LEVEL)) return(0);
    else {
        //----Get cell's x and y locational codes and the x locational code of the
        //----cell's right neighbors
        unsigned int xLocCode = cell->xLocCode;
        unsigned int yLocCode = cell->yLocCode;
        unsigned int xRightLocCode = xLocCode  + binaryCellSize;


        //----Determine the smallest common ancestor of the cell and the cell's
        //----right neighbors
        unsigned int cellLevel, nextLevel;
        unsigned int diff = xLocCode ^ xRightLocCode;
        qtCell *pCell = cell;
        cellLevel = nextLevel = cell->level;
        QT_GET_COMMON_ANCESTOR(pCell,nextLevel,diff);


        //----Start from the smallest common ancestor and follow the branching
        //----patterns of the locational codes downward to the smallest right
        //----neighbor of size greater than or equal to cell
        nextLevel--;
        QT_TRAVERSE_TO_LEVEL(pCell,nextLevel,xRightLocCode,yLocCode,cellLevel);
        return(pCell);
    }
}


//------------------------------------------------------------------------------
// Locate the three leaf cell vertex neighbors touching the right-bottom vertex
// of a specified cell. bVtxNbr, rVtxNbr, and rbVtxNbr are set to null if the
// corresponding neighbor does not exist.
//------------------------------------------------------------------------------
void qtLocateRBVertexNeighbors (qtCell *cell, qtCell **bVtxNbr, qtCell **rVtxNbr,
qtCell **rbVtxNbr)
{
    //----There are no right neighbors if this is the right side of the quadtree and
    //----no bottom neighbors if this is the bottom of the quadtree
    unsigned int binCellSize = 1 << cell->level;
    unsigned int noRight = ((cell->xLocCode + binCellSize) >= (1 << QT_ROOT_LEVEL)) ? 1 : 0;
    unsigned int noBottom = (cell->yLocCode == 0) ? 1 : 0;


    //----Get cell's x and y locational codes and the x and y locational codes of
    //----the cell's right and bottom vertex neighbors
    unsigned int xRightLocCode = cell->xLocCode + binCellSize;
    unsigned int xLocCode = xRightLocCode - 0x00000001;
    unsigned int yLocCode = cell->yLocCode;
    unsigned int yBottomLocCode = yLocCode - 0x00000001;
    unsigned int rightLevel, bottomLevel;
    unsigned int diff;
    qtCell *commonRight, *commonBottom;


    //----Determine the right leaf cell vertex neighbor
    if (noRight) *rVtxNbr = 0;
    else {
        //----Determine the smallest common ancestor of the cell and the cell's
        //----right neighbor. Save this right common ancestor and its level for
        //----determining the right-bottom vertex.
        unsigned int level = cell->level;
        diff = xLocCode ^ xRightLocCode;
        commonRight = cell;
        QT_GET_COMMON_ANCESTOR(commonRight,level,diff);
        rightLevel = level;


        //----Follow the branching patterns of the locational codes downward from
        //----the smallest common ancestor to the right leaf cell vertex neighbor
        *rVtxNbr = commonRight;
```

```
        level--;
        QT_TRAVERSE_TO_LEVEL(*rVtxNbr,level,xRightLocCode,cell->yLocCode,0);
    }


    //----Determine the bottom leaf cell vertex neighbor
    if (noBottom) *bVtxNbr = 0;
    else {
        //----Determine the smallest common ancestor of the cell and the cell's
        //----bottom neighbor. Save this bottom common ancestor and its level for
        //----determining the right-bottom vertex.
        unsigned int level = cell->level;
        diff = yLocCode ^ yBottomLocCode;
        commonBottom = cell;
        QT_GET_COMMON_ANCESTOR(commonBottom,level,diff);
        bottomLevel = level;


        //----Follow the branching patterns of the locational codes downward from
        //----the smallest common ancestor to the bottom leaf cell vertex neighbor
        *bVtxNbr = commonBottom;
        level--;
        QT_TRAVERSE_TO_LEVEL(*bVtxNbr,level,xLocCode,yBottomLocCode,0);
    }


    //----Determine the right-bottom leaf cell vertex neighbor
    if (noRight || noBottom) *rbVtxNbr = 0;
    else {
        //----Follow the branching patterns of the locational codes downward from
        //----the smallest common ancestor (the larger of the right common ancestor
        //----and the bottom common ancestor) to the right-bottom leaf cell vertex
        //----neighbor
        if (rightLevel >= bottomLevel) {
            *rbVtxNbr = commonRight;
            rightLevel--;
            QT_TRAVERSE_TO_LEVEL(*rbVtxNbr,rightLevel,xRightLocCode,yBottomLocCode,0);

        } else {
            *rbVtxNbr = commonBottom;
            bottomLevel--;
            QT_TRAVERSE_TO_LEVEL(*rbVtxNbr,bottomLevel,xRightLocCode,yBottomLocCode,0);
        }
    }
}
```