In that this test relies on knowing which pairs to test, pair information has to be tracked separately. Whenever an object A is stored in a grid cell containing another object B, a counter corresponding to the pair (A, B) is incremented, and the pair is added to the tracking data structure. When either object moves out of one of their common cells, the counter is decremented. When the counter reaches zero, the pair is removed from the tracking data structure because they are no longer close. Only pairs in the tracking data structure undergo narrow-phase collision detection checking. The cost of moving an object is now much larger, but thanks to spatial coherence the larger grid cells have to be updated only infrequently. Overall, this scheme trades the cost of updating cells for the benefit of having to test fewer grid cells.

7.2.3 Other Hierarchical Grids

As presented, a drawback with the hierarchical grid is that it is not very appropriate for handling world geometry, which often consists of large amounts of unevenly distributed small static data. Representing the lowest-level grid as a dense grid will be extremely expensive is terms of storage. Similarly, a sparse representation using hashing is likely to cause slowdown when a large number of cells is intersected due to cache misses and the extra calculations involved. For this particular type of problem, other types of hierarchical grids are more suited. Many of these have their origins in ray tracing.

Of these, the most straightforward alternative is the *recursive grid* [Jevans89]. Here, a uniform grid is constructed over the initial set of primitives. For each grid cell containing more than some k primitives, a new, finer, uniform grid is recursively constructed for that cell. The process is stopped when a cell contains less than some m objects or when the depth is greater than n (usually 2 or 3). Compared to the original hgrid, the lowest-level grids no longer span the world and they will require less memory for a dense grid representation.

Two other grid alternatives are *hierarchy of uniform grids* [Cazals95] and *adaptive grids* [Klimaszewski97]. Both of these use object clustering methods to identify groups of primitives that should be contained within their own subgrids inside a larger uniform grid. More information and a thorough comparison of these two and other methods can be found in [Havran99].

7.3 Trees

Trees were discussed in the previous chapter in the context of bounding volume hierarchies. Trees also form good representations for spatial partitioning. This section explores two tree approaches to spatial partitioning: the octree and the k-d tree.

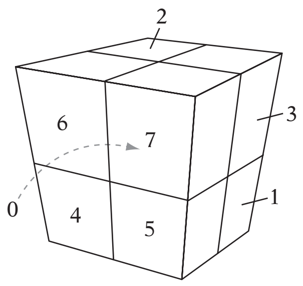

Figure 7.10 Numbering of the eight child nodes of the root of an octree.

7.3.1 Octrees (and Quadtrees)

The archetypal tree-based spatial partitioning method is the *octree*. It is an axis-aligned hierarchical partitioning of a volume of 3D world space. As the name suggests, each parent node in the octree has eight children. Additionally, each node also has a finite volume associated with it. The root node volume is generally taken to be the smallest axis-aligned cube fully enclosing the world. This volume is then subdivided into eight smaller equal-size subcubes (also called cells, octants, or cubelets) by simultaneously dividing the cube in half along each of the x, y, and z axes (Figure 7.10). These subcubes form the child nodes of the root node. The child nodes are, in turn, recursively subdivided in the same fashion. Typical criteria for stopping the recursive creation of the octree include the tree reaching a maximum depth or the cubes getting smaller than some preset minimum size. By virtue of construction, the associated volume of space for each node of the tree fully contains all descendant nodes.

Even though the geometric boundaries of a node can, for instance, be passed down and refined as the octree is traversed, the node structure is often enhanced to contain both node position and extent. This simplifies tree operations at the cost of extra memory. In addition, if the requirement for splitting into eight child volumes of equal size is relaxed to allow off-center splitting independently on all three axes, this volume information must be stored in the nodes.

```
// Octree node data structure
struct Node {
    Point center;      // Center point of octree node (not strictly needed)
    float halfWidth;   // Half the width of the node volume (not strictly needed)
    Node *pChild[8];   // Pointers to the eight children nodes
    Object *pObjList;  // Linked list of objects contained at this node
};
```

The analogous structure to the octree in two dimensions is known as a *quadtree*. The world is now enclosed in an axis-aligned bounding square, and is subdivided into four smaller squares at each recursive step.

Just as binary trees can be represented in a flat array without using pointers, so can both quadtrees and octrees. If the tree nodes are stored in the zero-based array **`node[N]`**, then being at some parent node **`node[i]`**, the children would for a quadtree be **`node[4*i+1]`** through **`node[4*i+4]`** and for an octree **`node[8*i+1]`** through **`node[8*i+8]`**. This representation requires the tree being stored as a complete tree.

Where a complete binary tree of n levels has $2^n - 1$ nodes, a complete d-ary tree of n levels has $(d^n - 1) / (d - 1)$ nodes. Using this type of representation, a complete seven-level octree would require just short of 300,000 nodes, limiting most trees to five or possibly six levels in practice.

As the tree must be preallocated to a specific depth and cannot easily grow further, the array representation is more suitable for static scenes and static octrees. The pointer-based representation is more useful for dynamic scenes in which the octree is constantly updated.

7.3.2 Octree Object Assignment

Octrees may be built to contain either static or dynamic data. In the former case, the data may be the primitives forming the world environment. In the latter case, the data is the moving entities in the world.

In the static scenario, it is straightforward to form the octree using top-down construction. Initially, all primitives in the world are associated with the root volume. As the root volume is split, the set of primitives is reassigned to the child cells it overlaps, duplicating the primitives when they overlap more than one cell. The procedure is repeated recursively, with the stopping criteria of not subdividing cells containing fewer than some fixed number of primitives or when primitives are assigned to all of the children (obviating further subdivision).

Determining to which child volumes to assign a primitive can be done by first dividing the primitives by the octree x-axis partitioning plane, splitting any straddling primitives into two parts, one for each side. Both sets are then similarly split for the y-axis partitioning plane, and the resulting four sets are split once more by the z-axis plane. The eight final sets are then assigned to the corresponding child node. Whether a primitive must be split by a given partitioning plane is determined by testing to see if it has vertices on both sides of the plane. Splitting of polygon primitives is discussed in detail in Section 8.3.4.

An alternative assignment approach is not to split the primitives. Instead, for each child volume and each primitive a full AABB-primitive overlap test is performed to see if the primitive should be assigned to that volume. Although perhaps a conceptually simpler method, this test is more expensive.

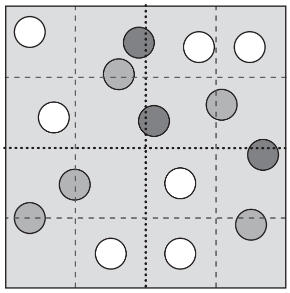

Figure 7.11 A quadtree node with the first level of subdivision shown in black dotted lines, and the following level of subdivision in gray dashed lines. Dark gray objects overlap the first-level dividing planes and become stuck at the current level. Medium gray objects propagate one level down before becoming stuck. Here, only the white objects descend two levels.

To avoid duplication of geometry between multiple octree nodes, the original set of primitives is stored in an array beforehand and a duplicate set of primitives is passed to the construction function. Each of the duplicate primitives is given a reference to the original primitive. When primitives are split, both parts retain the reference to the original primitive. As tree leaf nodes are generated, rather than storing the split-up duplicate geometry the references to the original set of primitives are stored. As the tree is static, this work is best done in a preprocessing step.

Duplicating the primitives across overlapped octants works great for a static environment. However, this approach is inefficient when the octree holds dynamic objects because the tree must be updated as objects move. For the dynamic scenario, a better approach is to restrict placement of objects to the lowest octree cell that contains the object in its interior. By associating the object with a single cell only, a minimum amount of pointer information must be updated when the object moves. A drawback of restricting objects to lie within a single octree cell is that objects may be higher in the tree than their size justifies. Specifically, any object straddling a partitioning plane will become"stuck"at that level, regardless of the relative size between the object and the node volume (Figure 7.11). This problem is addressed later. From here on, it is assumed the octree is of the dynamic variety.

The octree can begin as an empty tree grown on demand as objects are inserted. It can also be preallocated to alleviate the cost of dynamically growing it. Given the previous octree node definition, preallocating a pointer-based octree down to a specified depth can be done as:

```
// Preallocates an octree down to a specific depth
Node *BuildOctree(Point center, float halfWidth, int stopDepth)
{
    if (stopDepth < 0) return NULL;
    else {
        // Construct and fill in 'root' of this subtree
        Node *pNode = new Node;
        pNode->center = center;
        pNode->halfWidth = halfWidth;
        pNode->pObjList = NULL;

        // Recursively construct the eight children of the subtree
        Point offset;
        float step = halfWidth * 0.5f;
        for (int i = 0; i < 8; i++) {
            offset.x = ((i & 1) ? step : -step);
            offset.y = ((i & 2) ? step : -step);
            offset.z = ((i & 4) ? step : -step);
            pNode->pChild[i] = BuildOctree(center + offset, step, stopDepth - 1);
        }
        return pNode;
    }
}
```

Because objects have been restricted to reside in a single cell, the simplest linked-list solution for handling multiple objects in a cell is to embed the linked-list next pointer in the object. For the sample code given here, the object is assumed to be structured as follows.

```
struct Object {
    Point center;         // Center point for object
    float radius;         // Radius of object bounding sphere
    ...
    Object *pNextObject;  // Pointer to next object when linked into list
};
```

The code for inserting an object now becomes:

```
void InsertObject(Node *pTree, Object *pObject)
{
    int index = 0, straddle = 0;
    // Compute the octant number [0..7] the object sphere center is in
    // If straddling any of the dividing x, y, or z planes, exit directly
    for (int i = 0; i < 3; i++) {
        float delta = pObject->center[i] - pTree->center[i];
        if (Abs(delta) < pTree->halfWidth + pObject->radius) {
            straddle = 1;
            break;
        }
        if (delta > 0.0f) index |= (1 << i);  // ZYX
    }
    if (!straddle && pTree->pChild[index]) {
        // Fully contained in existing child node; insert in that subtree
        InsertObject(pTree->pChild[index], pObject);
    } else {
        // Straddling, or no child node to descend into, so
        // link object into linked list at this node
        pObject->pNextObject = pTree->pObjList;
        pTree->pObjList = pObject;
    }
}
```

InsertObject() can be changed so that whenever a child node pointer is NULL a new node is allocated and linked into the tree, with the object inserted into this new node.

```
if (!straddle) {
    if (pTree->pChild[index] == NULL) {
        pTree->pChild[index] = new Node;
        ...initialize node contents here...
    }
    InsertObject(pTree->pChild[index], pObject);
} else {
    ...same as before...
}
```

If nodes should be deleted in a corresponding manner whenever they no longer contain any objects, a pointer (or similar mechanism) is required in the node structure to be able to access the parent node for updating. The creation of new nodes can be delayed until a node contains *n* objects. When this happens, the node is split into eight child nodes, and the objects are reassigned among the children. When creating child nodes dynamically, all that is needed to build an octree incrementally is to create the initial root node; all other nodes are created as needed. Testing a dynamic octree for collisions can be done as a recursive top-down procedure in which the objects of each nonempty node are checked against all objects in the subtree below the node.

```
// Tests all objects that could possibly overlap due to cell ancestry and coexistence
// in the same cell. Assumes objects exist in a single cell only, and fully inside it
void TestAllCollisions(Node *pTree)
{
    // Keep track of all ancestor object lists in a stack
    const int MAX_DEPTH = 40;
    static Node *ancestorStack[MAX_DEPTH];
    static int depth = 0; // 'Depth == 0' is invariant over calls

    // Check collision between all objects on this level and all
    // ancestor objects. The current level is included as its own
    // ancestor so all necessary pairwise tests are done
    ancestorStack[depth++] = pTree;
    for (int n = 0; n < depth; n++) {
        Object *pA, *pB;
        for (pA = ancestorStack[n]->pObjList; pA; pA = pA->pNextObject) {
            for (pB = pTree->pObjList; pB; pB = pB->pNextObject) {
                // Avoid testing both A->B and B->A
                if (pA == pB) break;
                // Now perform the collision test between pA and pB in some manner
                TestCollision(pA, pB);
            }
        }
    }

    // Recursively visit all existing children
    for (int i = 0; i < 8; i++)
        if (pTree->pChild[i])
            TestAllCollisions(pTree->pChild[i]);

    // Remove current node from ancestor stack before returning
    depth--;
}
```

7.3.3 Locational Codes and Finding the Octant for a Point

Consider a query point somewhere inside the space occupied by an octree. The leaf node octant in which the point lies can be found without having to perform direct

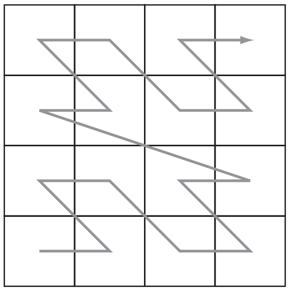

Figure 7.12 The cells of a 4 × 4 grid given in Morton order.

comparisons with the coordinates of the octree nodes. Let the point's coordinates be given as three floats x, y, and z. Remap each of these into binary fractions over the range $[0 \ldots 1]$. For instance, if the octree's x range spans $[10 \ldots 50]$, then an x coordinate of 25 would map into the fraction $(25 - 10)/(50 - 10) = 0.375 = 0.01100000$. Note that this does not require either all dimensions to be the same or that they be powers of 2. By virtue of construction, it should be clear that the most significant bit of the binary fraction for the x coordinate specifies whether the point lies to the left of the octree yz plane through the octree center point (when the bit is 0) or to the right of it (when the bit is 1). Taking the three most significant bits of the binary fractions for x, y, and z together as a unit, they form the index number (0–7) of the root's child node, in which the point lies. Similarly, the three next-most significant bits specify the subnode within that node in which the point lies, and so on until a leaf node has been reached.

The resulting binary string of interleaved bits of the three binary fractions can be seen as a key, a *locational code*, directly describing a node position within the tree. Nodes output in sorted order based on this locational code are said to be in *Morton order* (Figure 7.12).

Given the locational code for the parent node, a new locational code for one of its child nodes is easily constructed by left-shifting the parent key by 3 and adding (or ORing) in the parent's child index number (0–7) for the child node **`childKey = (parentKey << 3) + childIndex`**. Locational codes can be used in a very memory-efficient octree representation, explored next.

7.3.4 Linear Octrees (Hash-based)

Although the pointer-based representation is likely to save memory over the flat array-based representation for average trees, the former still requires up to eight child pointers to be stored in the octree nodes. These likely constitute a major part of the memory required to hold the tree.

A clever nonpointer-based representation alternative is the *linear octree*. A linear octree consists of just the octree nodes containing data, in which each node has been enhanced to contain its own locational code. The locational codes for both the parent node and all children nodes can be computed from the stored locational code. As such, the octree nodes no longer need explicit pointers to the children, thereby becoming smaller.

```
// Octree node data structure (hashed)
struct Node {
    Point center;     // Center point of octree node (not strictly needed)
    int key;          // The location (Morton) code for this node
    int8 hasChildK;   // Bitmask indicating which eight children exist (optional)
    Object *pObjList; // Linked list of objects contained at this node
};
```

The size of a node can be explicitly stored or it can be derived from the "depth" of the locational code. In the latter case, a sentinel bit is required in the locational code to be able to distinguish between, say, 011 and 000000011, turning these into 1011 and 1000000011, respectively. The code for this follows.

```
int NodeDepth(unsigned int key)
{
    // Keep shifting off three bits at a time, increasing depth counter
    for (int d = 0; key; d++) {
        // If only sentinel bit remains, exit with node depth
        if (key == 1) return d;
        key >>= 3;
    }
    assert(0);    // Bad key
}
```

To be able to access the octree nodes quickly given just a locational code, the nodes are stored in a hash table using the node's locational code as the hash key. This hashed storage representation provides $O(1)$ access to any node, whereas locating an arbitrary node in a pointer-based tree takes $O(\log n)$ operations.

To avoid a failed hash table lookup, the node is often enhanced (as previously) with a bitmask indicating which of the eight children exist. The following code for visiting all existing nodes in a linear octree illustrates both how child nodes are computed and how the bitmask is used.

```
void VisitLinearOctree(Node *pTree)
{
    // For all eight possible children
    for (int i = 0; i < 8; i++) {
        // See if the ith child exist
        if (pTree->hasChildK & (1 << i)) {
            // Compute new Morton key for the child
            int key = (pTree->key << 3) + i;
            // Using key, look child up in hash table and recursively visit subtree
            Node *pChild = HashTableLookup(gHashTable, key);
            VisitLinearOctree(pChild);
        }
    }
}
```

A hashed octree similar to this is described in [Warren93].

7.3.5 Computing the Morton Key

Given three integer values representing the *x*, *y*, and *z* coordinates of an octree leaf, the corresponding Morton key can be computed by the following function.

```
// Takes three 10-bit numbers and bit-interleaves them into one number
uint32 Morton3(uint32 x, uint32 y, uint32 z)
{
    // z--z--z--z--z--z--z--z--z--z-- : Part1By2(z) << 2
    // -y--y--y--y--y--y--y--y--y--y- : Part1By2(y) << 1
    // --x--x--x--x--x--x--x--x--x--x : Part1By2(x)
    // zyxzyxzyxzyxzyxzyxzyxzyxzyxzyx : Final result
    return (Part1By2(z) << 2) + (Part1By2(y) << 1) + Part1By2(x);
}
```

The support function **Part1By2()** splits up the bits of each coordinate value, inserting two zero bits between each original data bit. Interleaving the intermediate results is then done by shifting and adding the bits. The function **Part1By2()** can be implemented as:

```
// Separates low 10 bits of input by two bits
uint32 Part1By2(uint32 n)
{
    // n = ----------------------9876543210 : Bits initially
    // n = ------98----------------76543210 : After (1)
    // n = ------98--------7654--------3210 : After (2)
    // n = ------98----76----54----32----10 : After (3)
    // n = ----9--8--7--6--5--4--3--2--1--0 : After (4)
    n = (n ^ (n << 16)) & 0xff0000ff; // (1)
    n = (n ^ (n <<  8)) & 0x0300f00f; // (2)
    n = (n ^ (n <<  4)) & 0x030c30c3; // (3)
    n = (n ^ (n <<  2)) & 0x09249249; // (4)
    return n;
}
```

The equivalent code for interleaving 2D coordinates (for a quadtree) follows.

```
// Takes two 16-bit numbers and bit-interleaves them into one number
uint32 Morton2(uint32 x, uint32 y)
{
    return (Part1By1(y) << 1) + Part1By1(x);
}

// Separates low 16 bits of input by one bit
uint32 Part1By1(uint32 n)
{
    // n = ----------------fedcba9876543210 : Bits initially
    // n = --------fedcba98--------76543210 : After (1)
    // n = ----fedc----ba98----7654----3210 : After (2)
    // n = --fe--dc--ba--98--76--54--32--10 : After (3)
    // n = -f-e-d-c-b-a-9-8-7-6-5-4-3-2-1-0 : After (4)
    n = (n ^ (n <<  8)) & 0x00ff00ff; // (1)
    n = (n ^ (n <<  4)) & 0x0f0f0f0f; // (2)
    n = (n ^ (n <<  2)) & 0x33333333; // (3)
    n = (n ^ (n <<  1)) & 0x55555555; // (4)
    return n;
}
```

If sufficient bits are available in the CPU registers, the calls to the support function can be done in parallel. For example, using a modified **Part1By1()** function that operates on 32-bit numbers giving a 64-bit result, the **Morton2()** function presented previously can be written as:

```
uint32 Morton2(uint32 x, uint32 y)
{
    // Merge the two 16-bit inputs into one 32-bit value
    uint32 xy = (y << 16) + x;
    // Separate bits of 32-bit value by one, giving 64-bit value
    uint64 t = Part1By1_64BitOutput(xy);
    // Interleave the top bits (y) with the bottom bits (x)
    return (uint32)((t >> 31) + (t & 0x0ffffffff));
}
```

In general, for the 2D case just presented, handling n-bit inputs would require $4n$-bit numbers being available in the intermediate calculations.

7.3.6 Loose Octrees

Consider again the problem of objects becoming stuck high up in a dynamic octree due to straddling of the partitioning planes. An effective way of dealing with this problem is to expand the node volumes by some amount to make them partially overlapping (Figure 7.13). The resulting relaxed octrees have been dubbed *loose octrees* [Ulrich00]. The loose nodes are commonly extended by half the side width in all six directions (but may be extended by any amount). This effectively makes their volume eight times larger.

Assuming spherical objects, the objects are propagated into the child node in which the sphere center is contained. Only when the sphere extends outside the loose cell would it remain at the current level.

With these larger nodes, objects now descend deeper into the tree, giving the partitioning more discriminatory power. Unfortunately positioned but otherwise pairwise distant objects that previously became stuck high up in the tree are less likely to have to be tested against other objects. Another benefit of the loose nodes is that the

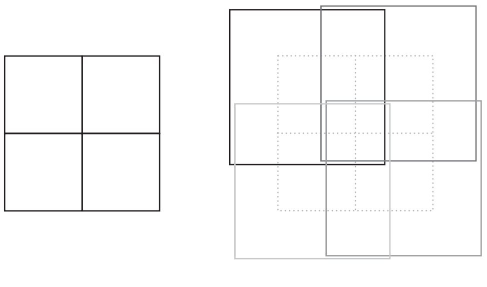

Figure 7.13 (a) The cross section of a regular octree, shown as a quadtree. (b) Expanding the nodes of the octree, here by half the node width in all directions, turns the tree into a loose octree. (The loose nodes are offset and shown in different shades of gray to better show their boundaries. The original octree nodes are shown as dashed lines.)

depth at which an object will be inserted can be computed from the size of the object, allowing for $O(1)$ insertions.

This power does not come for free, however. The larger nodes overlap more neighboring nodes, and more nodes have to be examined to determine whether any two objects must be tested against each other. Still, in most cases the reduction in pairwise tests makes loose octrees attractive for collision detection purposes.

Loose octrees are similar in structure to hierarchical grids. The key difference is that octrees are rooted. This means that time is spent traversing the (largely) empty top of the tree to get at the lower-level nodes where the objects are, due to objects generally being much smaller than the world. In comparison, the hgrid is a shallower, rootless structure providing faster access to the nodes where the objects are stored.

In addition, hgrid levels can be dynamically deactivated so that no empty levels are processed and processing stops when no objects exist on the remaining levels. A linearized or array-based octree allows similar optimizations. The corresponding optimization for a pointer-based octree is to maintain a subtree object-occupancy count for each node. The cost of updating these occupancy counts is proportional to the height of the tree, whereas the hgrid level occupancy counters are updated in $O(1)$ time. Thanks to the automatic infinite tiling performed by the hash mapping, a hashed hgrid does not need to know the world size. The octree cannot easily accommodate an unbounded world without growing in height. Overall, hgrids seem to perform somewhat better than loose octrees for collision detection applications.

7.3.7 *k*-d Trees

A generalization of octrees and quadtrees can be found in the *k*-dimensional tree, or *k-d tree* [Bentley75], [Friedman77]. Here, *k* represents the number of dimensions subdivided, which does not have to match the dimensionality of the space used.

Instead of simultaneously dividing space in two (quadtree) or three (octree) dimensions, the *k*-d tree divides space along one dimension at a time. Traditionally, *k*-d trees are split along *x*, then *y*, then *z*, then *x* again, and so on, in a cyclic fashion. However, often the splitting axis is freely selected among the *k* dimensions. Because the split is allowed to be arbitrarily positioned along the axis, this results in both axis and splitting value being stored in the *k*-d tree nodes. An efficient way of storing the two bits needed to represent the splitting axis is presented in Section 13.4.1. One level of an octree can be seen as a three-level *k*-d tree split along *x*, then *y*, then *z* (all splits selected to divide the space in half). Figure 7.14 shows a 2D spatial *k*-d tree decomposition and the corresponding *k*-d tree layout.

Splitting along one axis at a time often accommodates simpler code for operating on the tree, in that only the intersection with a single plane must be considered. Furthermore, because this plane is axis aligned, tests against the plane are numerically robust.

For collision detection, *k*-d trees can be used wherever quadtrees or octrees are used. Other typical uses for *k*-d trees include *point location* (given a point, locate the