The Etree Library:
A System for Manipulating Large Octrees on Disk

Tiankai Tu David R. O'Hallaron Julio C. López

July 2003

CMU-CS-03-174

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This report describes a library, called the *etree library*, that allows C programmers to manipulate large octrees stored on disk. Octrees are stored as a sequence of fixed sized octant records sorted by a locational code order that is equivalent to a preorder traversal of the tree and a Z-order traversal through the domain. The sorted records are indexed by a conventional file-resident B-tree index and queried using fixed-length locational code keys. A schema can be defined to make an etree portable across different platforms. The etree library provides functions for creating, modifying, and searching octrees, including efficient mechanisms for appending octants and iterating over octants in Z-order. The library is the foundation for a larger research effort aimed at enabling scientists and engineers to solve large physical simulations on their desktop systems by recasting the simulation process to work directly on large etrees stored on disk.

Effort sponsored in part by the National Science Foundation under grants 9980063 and 0122464, and in part by a grant from the Intel Corporation. Contact: Tiankai Tu, `tutk@cs.cmu.edu`.

Keywords: octree, etree, locational code, linear octree, spatial data structures, out-of-core computing

1 Introduction

An *octree* recursively subdivides a three-dimensional problem domain into eight equal size *octants* until a desired resolution level is achieved. The octree is an attractive data structure for several reasons [19, 18]. First, it provides a mechanism for aggregating similar regions of spatial data, which can save both space and time. Second, the general concepts of trees and hierarchies are familiar and well-understood. Many basic operations, such as neighbor finding, are implemented in terms of operations on trees such as tree traversal. Third, there exist efficient pointerless octree representations that store octants as sorted sequences. Because of these features, the octree has important applications in areas such as graphics, geographic information system, mesh generation, physical simulation, scientific visualization, and solid modeling.

An octree can be conveniently implemented with in-core pointers. However, the size of the octree is limited by the size of main memory. If a growing octree exhausts all the available memory, the application program will suffer an out-of-memory error and exit. This report describes the *etree library*, a system for manipulating large octrees on disk from C programs. With this approach, the size of an octree is no longer limited by the size of the main memory, but instead, by the size of the disk space. The main memory serves as a disk cache to boost the performance. An application running in a smaller memory might run slower, but it will run nevertheless. This approach rides on a number of long-term technology trends. For example, disk capacity is exploding and price per bit is plummeting, with terabytes of RAID storage available for under five thousand dollars. Further, RAID I/O throughput (128 MB/s) now approaches typical main memory throughput (100 MB/s) [5].

The etree library operates on *etrees*. To the external world, an etree is simply an ordinary UNIX file. Internally, an etree encapsulates the details of an octree and has a disk-resident index structure for fast octant data access. The etree abstraction is similar to the FILE abstraction in several ways. For example, a handle must be acquired before manipulating an etree. The handle must be passed as an argument to all etree function calls. Queries and updates to the octree are only supported by function calls to the etree library. Also, the etree performs resource management (e.g. buffer management, low-level I/O) automatically as does the FILE abstraction. However, there are also fundamental differences between the etree and the FILE abstraction. For one thing, the primitive data objects in the FILE abstraction are bytes. An application can access an arbitrary byte in the file stream by setting the file offset directly. In contrast, octants are the primitive data objects in the etree abstraction. Octants are organized in a database index structure (B-tree) that is transparent to the application. An application accesses octants by their addresses, not its location or offset in the index structure. Address resolution and data location are conducted by the etree library.

Section 2 gives a high-level overview of etrees and the structure of the etree library. Section 3 explains the octant data objects that the etree library works on. Section 4 documents the etree API (Application Program Interface). Section 5 shows several example C programs that use the API. Section 6 illustrates the internal mechanism of the etree library to support the API, which application programmers may safely skip.

The etree library is part of a larger research effort (the Euclid Project) to recast physical simulations such as finite element codes so that they operate on data stored in databases rather than in main memory. With this approach, all steps of the simulation process, including mesh generation, solving, and visualization, work by directly querying and updating etrees. Thus, problem sizes are limited by disk capacity rather than main memory size. The goal is to allow scientists and engineers to run large simulations on their desktop machines, thereby deferring the point at which they must move their codes to a supercomputer center. Section 7 describes these ongoing efforts and outlines some ideas for future research. Section 8 briefly discusses related work.

Finally, the appendix documents the portable etree file format that allows the same etree to be used on multiple platforms with incompatible byte ordering conventions and data alignment requirements.

2 Overview of the Etree Library

The design goal of the etree library is to allow C programs to efficiently access very large octrees created on any platform. In particular, the etree library should provide the following services.

- *Support large octrees.* The etree library should be scalable to support large octrees with billions of octants. The files storing octrees can be as large as hundreds of gigabytes to several terabytes.
- *Provide efficient access.* A scalable and dynamic index structure should be installed to enable efficient data accesses to large octrees. In addition, frequently used data should always be cached in main memory to reduce disk I/O.
- *Provide portability across different platforms.* Data files containing octrees should be self-contained and can be accessed on platforms with different architectures.

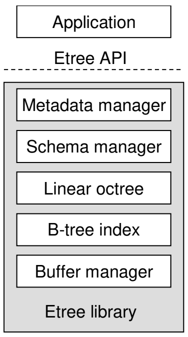


Figure 1: **Etree library components.**

These services are implemented by the components of the etree library summarized in Figure 1.

- *Buffer manager.* The buffer manager performs extensive data caching to reduce disk I/O [10]. It provides systems support for efficient data access of large files.
- *B-tree index.* A B-tree is an well-known indexing structure widely used in commercial database systems [3, 6, 10, 21]. It is a dynamic and scalable disk-resident data structure with $\log n$ complexity for all data access operations such as search and insert.
- *Linear octree.* The linear octree is a technique for assigning unique addresses to octants [8, 9, 1]. When used in conjunction with an indexing technique such as the B-tree, it allows octants to be located quickly, without actually storing the topological structure of the octree on disk.

- *Schema manager.* The schema manager allows an application program to register a schema describing individual fields of the payload of each octant. Once a schema is registered, the schema manager takes care of extracting data from the payload and guarantees that platform-specific data alignment requirements and byte ordering convention are observed. This is the critical component to support portability.
- *Metadata manager.* The metadata manager enables a data file containing an octree to be self-contained. It keeps track of an octree's structural information and records application-specific metadata. Besides, it provides accesses to the metadata.

All of these components are encapsulated in a C library (the etree library), which can be invoked by an application program through a small API. The etree library works on *etrees*. An etree is a database file that stores an octree as an ordered sequence of equal-sized octant record. Each record stores the address of the octant and its associated payload. For a given etree, each record has the same size. Different etrees can have arbitrarily sized records.

At run-time, the etree library performs extensive optimization to improve running time and reduce disk I/O. We defer the discussion of the internal mechanism to Section 6. The next section explains the key concepts of the etree library, which are indispensable to the understanding of the etree API.

3 Octant Data Objects

The primitive data objects the etree library manipulates are octant records. Each octant record consists of two parts: the address and the payload. This section explains how to construct and specify the address of an octant, and how to define and use the payload of an octant.

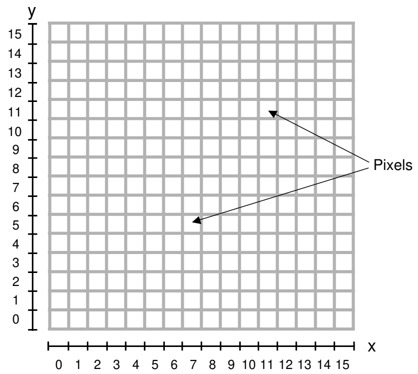


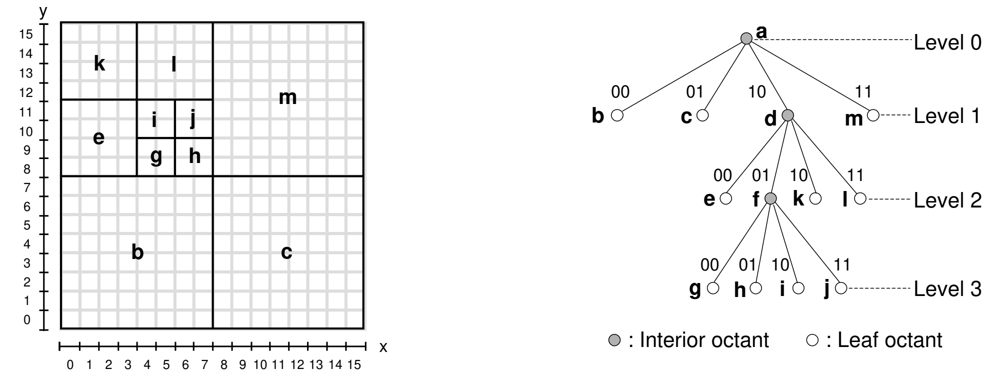
Figure 2: **A $2^4 \times 2^4$ domain of pixels.** Each pixel represents the smallest degree of refinement.

Quadtrees, the 2D counterparts of octrees, are easier to draw and understand than octrees, so we will use them whenever we need to illustrate basic concepts. The techniques we describe generalize to all dimensions. For simplicity, we will refer only to octants and octrees, regardless of the dimensionality.

3.1 Etree Address Space

An octree can be viewed in two equivalent ways: the *domain representation* and the *tree representation*. A *domain* is a Cartesian coordinate space that consists of a uniform grid of $2^n \times 2^n$ indivisible *pixels*. For example, Figure 2 shows the 2D domain where $n = 4$.

Figure 3(a) shows the domain representation of an octree, and Figure 3(b) shows an equivalent tree representation of the same octree. The *root octant* that spans the entire domain is said to be at level 0. Each child octant is one level higher that its parent. Each tree edge in Figure 3(b) is labeled with a *directional code* that distinguishes the children of each parent octant. Figure 4 shows the interpretation of the directional code in the context of the domain representation.



(a) Domain representation (b) Tree representation

Figure 3: **Domain and tree representations of an octree.** The edges of the tree representation are labeled with their corresponding directional codes (see Figure 4).

The etree library operates on a domain of $2^{31} \times 2^{31}$ pixels called the *etree address space*. The largest octant, the root octant that spans the entire domain, is at level 0. The smallest octant, which corresponds to a pixel, is at level 31. We refer to the unit edge length of each pixel as an *etree tick*, or simply a *tick*. In general, an octant at level k has an edge length of $2^{(31-k)}$ ticks. For example, a level-0 octant has an edge length of 2^{31} ticks, a level-1 octant an edge length of 2^{30}, and a level-31 pixel octant an edge length of one.

The 31-bit etree address space appears to provide sufficient spatial resolution for any application we can imagine. For example, if we were to embed a continent-sized volume $5,000$ kilometers on a side into the

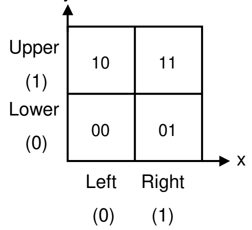


Figure 4: **Two-bit encoding of a directional code.**

etree address space, then each pixel would have an edge length on the order of 2 millimeters:

$$5 \times 10^6 \text{ meters}/2 \times 10^9 \text{ pixels} \approx 2 \times 10^{-3} \text{ meters/pixel}$$

Similarly, a volume the size of the 200 km^3 Los Angeles basin would have pixels on the order of 100 microns per side:

$$200 \times 10^3 \text{ meters}/2 \times 10^9 \text{ pixels} \approx 100 \times 10^{-6} \text{ meters/pixel}$$

If someday we find that 31 bits not sufficient, then there are hooks in place that will allow us to increase the address to 63 bits and still maintain backward compatibility with 31 bit etrees.

Each octant in an etree is uniquely identified by its *etree octant address*, which is a tuple of the form

$$(x, y, z, \textit{level})$$

where (x, y, z) is the location of the pixel in lower left hand corner of the octant (i.e., the pixel closest to the origin) and where *level* is the octant level. We will also associate a type with each octant, which identifies the octant as either a leaf octant with no children or an interior octant with eight children. We indicate the type of an octant by appending an 'I' or an 'L' to the address tuple. For example, assuming a z coordinate of zero, the root octant in Figure 3(a) labeled 'a' has an address of

$$(0, 0, 0, 0)I$$

and the leaf octant labeled 'i' has an address of

$$(4, 10, 0, 3)L$$

libsrc/etree.h

```
typedef enum { ETREE_INTERIOR = 0, ETREE_LEAF } etree_type_t;

typedef struct etree_addr_t {
    etree_tick_t x, y, z;   /* (x, y, z, t) is lower left corner */
    etree_tick_t t;         /* Time dimension for 4D etrees */
    int level;              /* Octant level  */
    etree_type_t type;      /* ETREE_LEAF or ETREE_INTERIOR */
} etree_addr_t;
```

libsrc/etree.h

Figure 5: **The etree address structure.** Applications manipulate structures of this type when they insert and search for octants in etrees. See Section 5 for examples.

Figure 5 shows the address structure exported to application programmers by the etree library Although the octant type (leaf or interior) is not a necessary part of an octant address, we include it in the address structure for those applications that must distinguish between leaf and interior octants.

An interesting question related to the etree octant address is how to represent a point in the domain. Mathematically, a point has no geometric span. However, in the etree library, we follow the convention in computer graphics and approximate a point as a pixel. To be more specific, a point/pixel corresponds to an octant at the lowest level in the tree representation and has an edge length of 1 tick in the etree address space. It is straightforward to see that the more bits we use for the etree address space, the higher resolution we can obtain.

We also note that it is the application programmer's task to establish the mapping between the real-world coordinate system of the problem domain and the discrete etree address space. For example, in earthquake ground motion simulations, the Los Angeles basin area is often specified by the latitude, the longitude and the depth of the origin and the number of kilometers along each of the three dimensions. One way to transform this coordinate system to the etree address space is to calculate the maximum span among the latitude, the longitude and the depth axes (say 100 kilometers) and then map this value to the size of the root octant. That is, the 100 kilometer span corresponds to 2^{31} ticks in the etree address space. Once the root octant is established, the remaining transformation is straightforward.

3.2 Schema

We now explain how to define and use the payload of an octant. In its simplest form, the payload of an octant is an arbitrary byte sequence. The interpretation of the byte sequence is left to the application program. Typically, an application program specifies the size of a structure as the size of the payload and accesses the payload by passing a pointer to a structure instance.

```
typedef struct payload_t {
    int intval;
    double dblval;
} payload_t;
```

Figure 6: **A C structure as payload.** An application specifies the size of the C structure (sizeof(payload_t)) to the etree library and passes a pointer to a structure instance to store and access the payload of an octant.

Obviously, the simplest form of payload imposes two limitations on the use of the etree. First, the etree can only be used on the same platform as it is originally created and populated. Otherwise, different byte ordering conventions and alignment requirements would cripple the interpretation of the payload. For example, Figure 6 shows a C structure name `payload_t`. Due to different default alignment requirements, the size of the example payload is 12 bytes on IA32/Linux (Figure 7(a)) but 16 on Sparc/SunOS (Figure 7(b)). Thus, passing a pointer to a Sparc/SunOS structure to retrieve the payload stored as an IA32/Linux structure garbles the meaning of the payload. The second limitation is that the entire payload must be passed to application programs even if only a single filed is actually needed.

To avoid these limitations, we allow an application to define (register) a *schema* for the payload of octants in an etree. A schema describes the name, type and size of each field of the payload. An application can register a schema with a newly created or truncated etree by providing a definition text string describing individual fields for the payload. The syntax of the definition string is similar to that for the body of a C structure. To guarantee portability, we stipulate that basic C data types (`char`, `int`, `float`, `double`) must be specified as size-explicit data types such as `int32_t`, `int64_t`, and `float32_t`. For the C structure in Figure 6, the schema definition string is:

```
int32_t intval; float64_t dblval;
```

(a) Layout of payload_t on IA32/Linux (b) Layout of payload_t on Sparc/SunOS

Figure 7: **Layout of the same C structure on different platforms.** Layouts produced by GCC on both platforms. Paddings marked by XXXX are inserted by compiler. IA32/Linux requires that any data bytes larger than 2 bytes must have an address that is a multiple of 4. However, Sparc/SunOS requires any double be aligned to have its address be a multiple of 8.

Once a schema is defined, it cannot be modified or deleted unless the etree itself is destroyed. An etree with a schema is portable across different platforms. That is, an etree created and populated on IA32/Linux can be reopened and queried on Sparc/SunOS or Alpha/DUX and vice versa without any problems. All byte ordering and alignment issues are resolved by the etree library automatically. In addition, applications can request a particular field of the schema from the payload without knowing other irrelevant fields.

This concludes our explanation of how to define octant data objects. Next we show how to operate on these octant data objects using etree library functions.

4 The Etree Library Interface

The section describes the Etree library API, which is mainly comprised of two types of functions: *stateless* and *stateful*. Stateless functions leave no state in the runtime system once they terminate. A stateless function call is independent of any other stateless function calls. In other words, an application can interleave stateless function calls in any order. Inserting, deleting, updating, searching, and various helper functions are stateless. Note that a stateless function may change the state of the etree itself. For example, insertion is a stateless function. After inserting an octant, the etree's state is definitely changed but this operation has no effect on the runtime system to perform future operations.

Stateful functions, on the other hand, set or change the state of the runtime system. Appending and cursor operations are stateful functions. Stateful functions can be viewed as the building blocks of transactions. A state is kept until a transaction is completed. During the lifetime of a transaction, no other transactions can be started and no stateless functions can be called. Note that stateful functions do not necessarily change the state of the etree (by adding or removing octants). For example, the cursor functions support preorder traversal of an octree, which do not modify the state of the etree at all.

The remainder of this section provides a reference guide for the etree API. See Section 5 for some examples of using the interface in application programs.

4.1 Constants and Structures

The `etree.h` header file provides the following constants:

- `ETREE_LEAF` and `ETREE_INTERIOR`: Denote leaf octants and interior octants, respectively, in the type field of the etree address structure.

- ETREE_MAXLEVEL: Defines the maximum octant level supported by the current version of the etree library. In the current version, the maximum level is 31.
- ETREE_BUFSIZE: Defines a safe size for any buffers passed to etree_straddr, which constructs a string representation of the fields in the octant address structure.

The header file also defines the *octant address structure* that applications use for inserting and searching operations. See Figure 5 for the definition of the address structure.

4.2 Error Reporting Functions

```
#include <etree.h>

etree_error_t etree_errno(etree_t *ep);
```

Returns: error code of the last failed operation.

The etree_errno function returns the error code of the last failed operation. The error code constants are defined in etree.h.

This function receives the following parameter:

- ep: handle to the etree for which to retrieve the error code.

```
#include <etree.h>

const char *etree_strerror(etree_error_t error);
```

Returns: text string describing an error

The etree_strerror function returns a text string describing the error corresponding to a given error code.

This function receives the following parameter:

- error: error code returned by the etree_errno function.

4.3 Opening and Closing Etrees

```
#include <etree.h>

etree_t *etree_open(const char *path, int32 flags, int bufsize, int pay-
loadsize, int dimensions);
```

Returns: a handle to an open etree file, NULL on error

The `etree_open` function opens an existing etree file or creates a new etree file depending on the options specified in the *flags* parameter. Internally this function allocates and initializes an etree handle.

This function receives the following parameters:

- `path`: The path name of the etree file in the file system.
- `flags`: This parameter specifies the mode in which the file is to be opened. Flags can have one of the following values: `O_RDONLY` or `O_RDWR`. The flags may also be bitwise-or'd with `O_CREAT` or `O_TRUNC`. The semantics are the same as in UNIX.
- `bufsize`: specifies the size of the internal buffer allocated to cache etree pages. The size is specified in megabytes. When set to 0, the default buffer size is set to be 20MB.
- `payloadsize`: The size of the payload of each octant. This parameter is only used to created a new etree database (i.e., O_CREAT, O_TRUNC was specified), otherwise the payload size is obtained from an existing etree.
- `dimensions`: The dimensionality for the new etree database. This parameter is only used to created a new etree database (i.e., O_CREAT, O_TRUNC was specified), otherwise the dimensions is the one stored in an existing etree.

On success this function returns an etree handle (`etree_t struct`) that can be used in subsequent calls to the etree library routines. This function returns NULL on error.

```
#include <etree.h>

int etree_close(etree_t *ep);
```

Returns: 0 on success, -1 on error

The `etree_close` function closes an etree file and releases the resources used to access that etree file.

This function receives the following parameter:

- `ep`: handle of the etree to be closed.

This function returns 0 on success, -1 on error. If an error occurs the application program should call `perror` to identify the nature of the problem.

4.4 Registering and Accessing Schema

Applications may optionally register a schema to make an etree portable across different platforms. Registration is a one time operation that occurs when an etree is newly created or truncated. No further actions are required to maintain the schema. To use a schema, an application simply specifies the field of interest when accessing octants stored in the etree. (See `etree_search` and `etree_getcursor` for details.)

```
#include <etree.h>

int etree_registerschema(etree_t *ep, const char *defstring);
```
Returns: 0 on success, -1 on error

The `etree_registerschema` function registers a schema with an empty etree (either newly created or truncated). Registering or changing a schema for a non-empty is not allowed no matter whether a schema has been registered with the etree or not. This function may only be called before insertion or appending to an empty octree. Otherwise, it is disabled and the etree is not allowed to have a schema.

This function receives the following parameters:

- `ep`: handle to the etree for which the schema is defined.
- `defstring`: definition text string of the schema. A definition is a list of field declarations separated by semicolons. Each variable declaration consists of a type name and *one* field name. Type names supported are listed in Figure 4.4. Field names can be arbitrary ASCII string. See Section 5 for examples of schema definition.

Type name	Size (bytes)
`char`	1
`int8_t`	1
`int16_t`	2
`int32_t`	4
`int64_t`	8
`uint16_t`	2
`uint32_t`	4
`uint64_t`	8
`float`	4
`float32_t`	4
`double`	8
`float64_t`	8

Figure 8: **Supported data types.** The numbers after the basic data types (`int`, `uint`, `float`) specify the exact number of bits for that data type. `char` is a synonym for `int8_t`, `float` for `float32_t`, and `double` for `float64_t`.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_DISALLOW_SCHEMA, ET_CREATE_SCHEMA.

Once a schema is successfully registered with an etree, it is associated with the etree for the lifetime of the etree. When the etree is closed, the schema (in its internal format) is stored in the etree on disk. The next time the etree is open (either on the same platform or on a different platform), the schema is automatically restored. There is no need to re-register the schema. With a schema in place, portability problems (byte ordering and data alignment) are resolved by the etree library. In addition, an application may retrieve individual field of the payload of an octant without having to know all the fields

```
#include <etree.h>

char *etree_getschema(etree_t *ep);
```
Returns: Pointer to a schema definition text string, NULL on error

The `etree_getschema` function reconstruct the schema definition text string associated with the etree. This function calls `malloc` to allocate the memory needed to store the application metadata text. The caller should release the allocated memory with `free`.

This function receives the following parameters:

- `ep`: handle to the etree for which the schema is defined.

This function returns a pointer to text string if there is a schema associated with the etree, NULL otherwise.

4.5 Inserting, Deleting, and Updating Octants

Inserting, deleting and updating are all stateless functions. They may be interleaved in any order in a program. However, they cannot be interleaved with stateful functions such as appending or cursor operations. Otherwise, the etree library reports an error of conflict in operation mode (ET_OP_CONFLICT).

```
#include <etree.h>

int etree_insert(etree_t *ep, etree_addr_t addr, const void *payload);
```
Returns: 0 on success, -1 on error

The `etree_insert` function inserts a new octant in an etree database. Inserting duplicate octants is not allowed. An octant is a duplicate of another if their octant addresses are exactly the same or only differ in the type field (ETREE_LEAF vs. ETREE_INTERIOR).

This function receives the following parameters:

- `ep`: handle to the etree to which the octant is to be inserted.
- `addr`: octant address structure containing the address of the octant to be inserted.
- `payload`: pointer to the payload of the octant to be inserted.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_LEVEL_OOB, ET_OP_CONFLICT, ET_DUPLICATE, ET_IO_ERROR, ET_NOT_WRITABLE.

```
#include <etree.h>

int etree_delete(etree_t *ep, etree_addr_t addr);
```
Returns: 0 on success, -1 on error

The `etree_delete` function deletes an octant from an etree.

This function receives the following parameters:

- `ep`: handle to the etree from which the octant is to be deleted.
- `addr`: octant address structure containing the address of the octant to be deleted.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_LEVEL_OOB, ET_OP_CONFLICT, ET_EMPTY_TREE, ET_NOT_FOUND, ET_IO_ERROR.

```
#include <etree.h>

int etree_update(etree_t *ep, etree_addr_t addr, const void *payload);
```

Returns: 0 on success, -1 on error

The `etree_update` function modifies the data associated with an octant in an etree database.

This function receives the following parameters:

- `ep`: handle to the etree where the octant is to be modified.
- `addr`: octant address structure containing the address of the octant to be modified. The data associated with the octant is modified only if the specified octant address matches exactly the address of the octant found in the database.
- `payload`: pointer to the new payload of the octant.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_LEVEL_OOB, ET_EMPTY_TREE, ET_NOT_FOUND, ET_IO_ERROR, ET_NOT_WRITABLE.

```
#include <etree.h>

int etree_sprout(etree_t *ep, etree_addr_t addr, const void *childpay-
load[8]);
```

Returns: 0 on success, -1 on error

The `etree_sprout` function looks up in the etree database for the octant with the specified address. Sprouting occurs if and only if such an octant exists in the etree and is a leaf octant. By sprouting, we mean replacing of a sprouting octant with its eight children. This operation is only supported in 3D etrees.

This function receives the following parameters:

- `ep`: handle to the etree from which the sprouting shall occur.

- `addr`: octant address structure containing the address of the octant to be sprouted. The octant is subdivided only if it exists in the database and is a leaf octant.
- `childpayload[8]`: array with the pointers to the payloads of the children octants. The pointers to the payloads are expected to be stored in a Z-order, that is, { (0,0,0); (0,0,1); (0,1,0); (0,1,1); (1,0,0); (1,0,1); (1,1,0); (1,1,1); }.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_NOT_3D, ET_NOT_LEAF_SPROUT, ET_LEVEL_OOB, ET_NO_MEMORY, ET_LEVEL_CHILD_OOB, ET_NO_ANCHOR, ET_EMPTY_TREE, ET_OP_CONFLICT, ET_IO_ERROR, ET_NOT_WRITABLE.

4.6 Appending Octants to Etrees

Internally, the etree stores an octree in its preorder traversal order. Therefore, applications that build etrees can achieve better performance by appending octants in the octree preorder traversal order rather than them in arbitrary order. In order to support appending operations, the etree runtime system needs to keep an internal state to record the position of the last append, which should not be tampered by other operations such as inserting and deleting.

The following three functions support appending octants to etrees efficiently. They are particularly useful to "bulk load" an etree.

```
#include <etree.h>

int etree_beginappend(etree_t *ep, double fillratio);
```

Returns: 0 on success, -1 on error

The `etree_beginappend` starts an *append* transaction. In an append transaction, an application can only append octants into the database. An attempt to restart an append transaction is ignored.

This function receives the following parameters:

- `ep`: etree handle of the database where the append operations are to be performed.
- `fillratio`: A positive value in the range between 0 to 1 inclusive to specify the fullness of the database storage pages for the append transaction. If the octree is not going to change structurally, set the fillratio to 1 to fully utilize the storage space and boost performance.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_ILLEGAL_FILL, ET_IO_ERROR.

```
#include <etree.h>

int etree_append(etree_t *ep, etree_addr_t addr, const void *payload);
```

Returns: 0 on success, -1 on error

The `etree_append` function appends a new octant to the etree. This function fails if the octant address specified does not preserve the preorder traversal order.

This function receives the following parameters:

- `ep`: handle to the etree to which the octant is to be appended.
- `addr`: octant address structure containing the address of the octant to be appended.
- `payload`: pointer to the payload of the octant to be appended.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_LEVEL_OOB, ET_APPEND_OOO, ET_NOT_WRITABLE, ET_IO_ERROR.

```
#include <etree.h>

int etree_endappend(etree_t *ep);
```
Returns: 0 on success, -1 on error

`etree_endappend` terminates an *append* transaction and restores the status of the runtime system to stateless.

This function receives the following parameters:

- `ep`: handle to the etree where an append transaction is being performed.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_NOT_APPENDING.

4.7 Searching for Octants

As an exception, search functions are special stateless functions that can be interleaved with stateful function calls. Since they will not destroy either the state of the etree or that of the runtime system. However, we do not encourage programmers to interleave stateless functions with stateful functions.

```
#include <etree.h>

int etree_search(etree_t *ep, etree_addr_t addr, etree_addr_t *hitaddr, const
char *fieldname, void* payload);
```
Returns: 0 on success, -1 on error

The `etree_search` function searches an octant in the etree database. On success the address of the octant found in the database is stored in `hitaddr` and its associated data is stored in `payload`. This function succeeds if it finds the exact octant being searched or one of its ancestors. An ancestor of an octant O_{child} is any octant O_{ancestor} that is closer to the root of the octree (i.e., O_{ancestor}.level $<$ O_{child}.level) and contains the octant for which the search is performed. Notice that the address of the found octant may differ from the octant being searched for.

This function receives the following parameters:

- `ep`: handle to the etree where the octant is to be searched.
- `addr`: octant address structure containing the address of the octant for which to be searched. This octant address does not need to specify the octant type (ETREE_LEAF vs. ETREE_INTERIOR).
- `hitaddr`: an output parameter that points to an octant address structure. If the search succeeds the address of the found octant is stored in the location pointed by `hitaddr`. The hit address also contains the octant's type information (ETREE_LEAF or ETREE_INTERIOR).
- `fieldname`: the name of a field if a schema is defined. To retrieve the entire payload, set `fieldname` to "*" if a schema is defined, or NULL if no schema is defined.
- `payload`: an output parameter that points to a structure instance if (1) a schema is defined and `fieldname` is "*"; or (2) no schema is defined and `fieldname` is NULL. Or it points to a basic data type corresponding to `fieldname` if a schema is defined and `fieldname` is part of the schema. On success, the data associated with the found octant is stored in the payload properly.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_LEVEL_OOB, ET_LEVEL_OOB2, ET_EMPTY_TREE, ET_NOT_FOUND, ET_IO_ERROR. ET_NO_SCHEMA, ET_NO_FIELD.

```
#include <etree.h>

int etree_findneighbor(etree_t *ep, etree_addr_t addr, etree_dir_t dir,
etree_addr_t *nbaddr, const char *fieldname, void *payload);
```

Returns: 0 on success, -1 on error

The `etree_findneighbor` function searches for an octant's neighbor in the etree database. This operation is supported only for 3D etree databases and the search is performed only for face and edge neighbors. Corner neighbor search is not supported in the current release.

This function receives the following parameters:

- `ep`: handle to the etree where the neighbor is to be searched.
- `addr`: octant address structure containing the address of the octant for which a neighbor is to be searched.
- `dir`: specifies the direction of the search.

- nbaddr: an output parameter that points to an octant address structure where the address of the found neighbor is stored if the search succeeds.
- fieldname: the name of a field if a schema is defined. To retrieve the entire payload, set fieldname to "*" if a schema is defined, or NULL if no schema is defined.
- payload: an output parameter that points to a structure instance if (1) a schema is defined and fieldname is "*"; or (2) no schema is defined and fieldname is NULL. Or it points to a basic data type corresponding to fieldname if a schema is defined and fieldname is part of the schema. On success, the data associated with the found octant is stored in the payload properly.

This function returns 0 on success and -1 on error. Application programs should call etree_errno and etree_strerror to identify the nature of the error. The following error codes can be set by this function: ET_NOT_3D, ET_INVALID_NEIGHBOR, ET_LEVEL_OOB , ET_LEVEL_OOB2, ET_EMPTY_TREE, ET_NOT_FOUND, ET_IO_ERROR, ET_NO_SCHEMA, ET_NO_FIELD.

4.8 Performing Preorder Traversals of Octrees

The functions in this section are stateful functions that provide an efficient mechanism for visiting the octants of an octree in preorder.

```
#include <etree.h>

int etree_initcursor(etree_t *ep, etree_addr_t addr);
```

Returns: 0 on success, -1 on error

The etree_initcursor function sets the etree in preorder traversal mode. This is also known as *initializing the etree cursor*. You must call this function before retrieving or advancing the etree cursor.

This function receives the following parameters:

- ep: handle to the etree where the traversal is to be performed.
- addr: octant address structure specifying the address of the octant where the traversal operation starts.

This function returns 0 on success and -1 on error. Application programs should call etree_errno and etree_strerror to identify the nature of the error. The following error codes can be set by this function: ET_LEVEL_OOB, ET_EMPTY_TREE, ET_IO_ERROR.

```
#include <etree.h>

int etree_getcursor(etree_t *ep, etree_addr_t *addr, const char *fieldname,
void *payload);
```

Returns: 0 on success, -1 on error

The `etree_getcursor` function obtains the address and content of the octant to which the cursor currently points.

This function receives the following parameters:

- `ep`: handle to the etree where the traversal is performed.
- `addr`: an output parameter that points to an octant address structure where the address of the current octant pointed to by the cursor is to be stored.
- `fieldname`: the name of a field if a schema is defined. To retrieve the entire payload, set `fieldname` to "*" if a schema is defined, or NULL if no schema is defined.
- `payload`: an output parameter that points to a structure instance if (1) a schema is defined and `fieldname` is "*"; or (2) no schema is defined and `fieldname` is NULL. Or it points to a basic data type corresponding to `fieldname` if a schema is defined and `fieldname` is part of the schema. On success, the data associated with the found octant is stored in the payload properly.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_NO_CURSOR, ET_LEVEL_OOB2, ET_NO_SCHEMA, ET_NO_FIELD.

```
#include <etree.h>

int etree_advcursor(etree_t *ep);
```

Returns: 0 on success, -1 on error

The `etree_advcursor` function moves the cursor to the next octant in preorder traversal.

This function receives the following parameter:

- `ep`: handle to the etree where the traversal is performed.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_NO_CURSOR, ET_END_OF_TREE, ET_IO_ERROR.

```
#include <etree.h>

int etree_stopcursor(etree_t *ep);
```

Returns: 0 on success, -1 on error

The `etree_stopcursor` function finishes the traversal operation.

This function receives the following parameter:

- `ep`: handle to the etree where the traversal is performed.

This function returns 0 on success and -1 on error. Application programs should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_NO_CURSOR.

4.9 Miscellaneous Helper Functions

```
#include <etree.h>

char *etree_straddr(etree_t *ep, char *buf, etree_addr_t addr);
```
Returns: a text representation of an octant address

The `etree_straddr` function creates a human-readable text representation of an octant address.

This function receives the following parameters:

- `ep`: handle to the etree that the octant belongs to.
- `buf`: character array where to store the text representation of the octant address.
- `addr`: octant address to represent as text.

This function returns a pointer to where the octant address text representation is stored (i.e., buf).

```
#include <etree.h>

int etree_getmaxleaflevel(etree_t *ep);
```
Returns: The maximum leaf level in the etree

The `etree_getmaxleaflevel` function returns the highest level in the etree where there is at least one leaf octant.

This function receives the following parameter:

- `ep`: handle to the etree where to obtain the information from.

If the return value is -1, then there are no leaf octants stored in the etree.

```
#include <etree.h>

int etree_getminleaflevel(etree_t *ep);
```
Returns: The minimum current leaf level in the etree

The `etree_getminleaflevel` functions returns the lowest level in the etree where there is at least one leaf octant.

This function receives the following parameter:

- `ep`: etree handle (i.e., etree_t struct) where to obtain the information from.

If the return value is -1, then there are no leaf octants stored in the etree.

4.10 Fetching and Retrieving Application Metadata Strings

An etree can contain an optional application-specific metadata text string. The functions in this section allow applications to manipulate these strings.

```
#include <etree.h>

char *etree_getappmeta(etree_t *ep);
```
Returns: Pointer to application metadata text string, NULL on error

The `etree_getappmeta` function retrieves the metadata text string associated with the etree. This function calls `malloc` to allocate the memory needed to store the application metadata text. The caller should release the allocated memory with `free`.

- `ep`: handle to the etree.

On success, this function returns a pointer to the metadata text string if application metadata is defined; NULL if no application metadata is defined.

```
#include <etree.h>

int etree_setappmeta(etree_t *ep, const char *appmetadata);
```
Returns: 0 on success, -1 on error

The `etree_setappmeta` function sets the application metadata text. When the etree file is successfully closed, the metadata text is stored in the etree file.

This function receives the following parameters:

- `ep`: handle to the etree
- `appmetadata`: the application metadata text to be stored in the etree file.

This function returns 0 on success, -1 on error. If an error occurs the application program should call `etree_errno` and `etree_strerror` to identify the nature of the error. The following error codes can be set by this function: ET_NO_MEMORY.

5 Example Programs

This section gives some examples of using the etree library to manipulate simple octrees. The example programs are available in the etree distribution. They were formatted directly from the original source files. File names are documented in the horizontal bars that surround the formatted code.

5.1 `tiny`: Writing a Simple Etree Program

Figure 9 illustrates the basic operations of creating etrees, inserting octants into etrees, and searching for octants. The program creates an empty etree and inserts a level-0 parent octant anchored at the origin that spans the entire etree domain. It then searches for a tiny non-existent level-31 octant that is a remote descendant of the existing parent octant. The search returns the address and payload of the nearest existing ancestor, which in this case is the original parent.

This simple example clearly shows how octrees aggregate numerous small volumes of space into a single larger representative. Since this is our first etree program, let us dissect it carefully. Line 1 includes the etree header file, which is required by every etree application. Lines 5–8 define the etree handle, the octant address structures for the insert and search operations, the payloads for the octants, and a buffer that will be passed to the `etree_straddr` function. Lines 11–15 creates an empty 3D etree where each octant holds a single integer value. Although this example manipulates octants that contain single integers, in general, etree octants may have arbitrary-sized payload.

Lines 17–21 register a schema with the etree. Note that schema registration is performed before inserting any octant to the empty etree. Lines 23–30 insert a parent octant spanning the entire domain and anchored at location $(0, 0, 0)$. Then lines 32–38 search for a non-existent level-31 descendant of the parent, also anchored at the origin. The location and payload of the resulting octant are returned in `res_addr` and `res_val` respectively.

Running the program gives the following output:

```
unix> ./tiny tree.e
Query : (0 0 0 31)L
Result: (0 0 0 0)L = 15213
```

Octant addresses are displayed as tuples of the form $(x, y, z, level)$ and appended with an “L” (leaf) or “I” (interior). The key idea here is that the search returns the closest existing ancestor, in this case the level-0 octant that spans the entire domain.

5.2 `txt2etree`: Building Octrees Iteratively

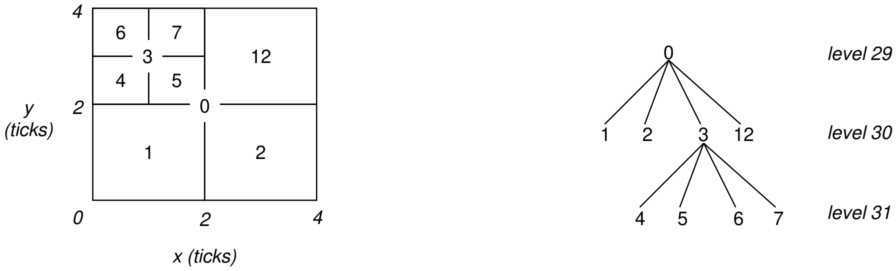
Next we develop a program that converts a text representation of an octree into an etree. In particular, we will build the 3D octree shown in Figure 10.

This example tree is the simplest nontrivial octree, and we will refer to it often throughout the remainder of this section. The tree consists of three levels. The level-29 root, which has an edge length of four ticks, is split into eight level-30 children octants with edge lengths of two ticks. One of these children is itself *examples/tiny.c*

```c
#include "etree.h"

int main(int argc, char **argv)
{
    etree_t *ep;
    etree_addr_t parent, child, res_addr;
    int parent_val, res_val;
    char buf[ETREE_MAXBUF];

    /* Create an empty 3d etree where each record is an int */
    ep = etree_open(argv[1], O_RDWR|O_CREAT|O_TRUNC, 0, sizeof(int), 3);
    if (ep == NULL) {
        fprintf(stderr, "Could not open %s\n", argv[1]);
        exit(1);
    }

    /* Register schema for the etree to make it portable */
    if (etree_registerschema(ep, "int32_t val") != 0) {
        fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
        exit(1);
    }

    /* Insert a parent octant that spans the entire domain */
    parent.x = parent.y = parent.z = parent.level = 0;
    parent.type = ETREE_LEAF;
    parent_val = 15213;
    if (etree_insert(ep, parent, &parent_val) != 0) {
        fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
        exit(1);
    }

    /* Search for non-existent child of the octant we just inserted */
    child.x = child.y = child.z = 0;
    child.level = ETREE_MAXLEVEL;
    if (etree_search(ep, child, &res_addr, "val", &res_val) != 0) {
        fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
        exit(1);
    }
    printf("Query : %s\n", etree_straddr(ep, buf, child));
    printf("Result: %s = %d\n", etree_straddr(ep, buf, res_addr), res_val);
    etree_close(ep);

    return 0;
}
```

examples/tiny.c

Figure 9: **The `tiny` program.** A simple program that illustrates some of the key operations on etrees. Builds an octree consisting of a single parent octant that spans the entire domain, then searches for a non-existent child octant. The search correctly returns the address and payload of the parent.



(a) Domain representation (b) Tree representation

Figure 10: **Example 3D octree at $z = 0$.** This tree will serve as a running example for the rest of the section. Octants are labeled with their position in Z-order. Units for coordinate axes are in etree ticks.

split into eight level-31 grandchildren with edge lengths of 1 tick. Notice that the grandchildren are at the maximum level in the tree and have the smallest possible edge lengths. Also, for convenience, each octant is labeled with its Z-order position. Recall that Z-order traversal of the domain is equivalent to a preorder traversal of the tree.

Figure 11 shows a text representation of the example octree from Figure 10. Each line of the file specifies the address and payload of an octant in the tree. The payload consists of two fields: `val` and `tag`. The `val` field records the position of the octant in preorder traversal and the `tag` field associates a letter with the level of the octant in the tree. In particular, 'A' corresponds to level 29, 'B' to 30, and 'C' to 31. Although the `tag` has no effect on the example program, it helps to illustrate the idea that the payload of octants may consist of arbitrary number of fields, each of which may have one of the basic data type listed in Figure 4.4.

The `txt2etree` program in Figure 12 builds an octree on disk by inserting octants into the etree one at a time. It reads a text file such as Figure 11 from stdin. Each line of the text file gives the address and payload of an octant, which the program inserts in the etree by calling the `etree_insert` function. After all the octants are inserted, a comment is added by `txt2etree` as a text string to the etree by `etree_setappmeta` (Lines 37–40). This is a piece of application specific metadata. It may be arbitrary long text string and can be modified any time when the etree is open.

Running the `txt2etree` program with the `tree.txt` file as input builds the octree shown in Figure 10:

```
linux> ./txt2etree tree.e < tree.txt
Loaded (0 0 0 29)I = A 0
Loaded (0 0 0 30)L = B 1
Loaded (2 0 0 30)L = B 2
Loaded (0 2 0 30)I = B 3
Loaded (2 2 0 30)L = B 12
Loaded (0 0 2 30)L = B 13
Loaded (2 0 2 30)L = B 14
Loaded (0 2 2 30)L = B 15
Loaded (2 2 2 30)L = B 16
Loaded (0 2 0 31)L = C 4
Loaded (1 2 0 31)L = C 5
Loaded (0 3 0 31)L = C 6
Loaded (1 3 0 31)L = C 7
```

examples/tree.txt

```
# Example octree. Each line has the following form:
# <x> <y> <z> <level> <leaf node?> <payload.val> <payload.tag>

# Parent node
0 0 0 29  0 0 A

# Children of node [0,0,0,29]
0 0 0 30  1 1 B
2 0 0 30  1 2 B
0 2 0 30  0 3 B
2 2 0 30  1 12 B
0 0 2 30  1 13 B
2 0 2 30  1 14 B
0 2 2 30  1 15 B
2 2 2 30  1 16 B

# Children of node [0,2,0,30]
0 2 0 31  1 4 C
1 2 0 31  1 5 C
0 3 0 31  1 6 C
1 3 0 31  1 7 C
0 2 1 31  1 8 C
1 2 1 31  1 9 C
0 3 1 31  1 10 C
1 3 1 31  1 11 C
```

examples/tree.txt

Figure 11: **The `tree.txt` file.** Text representation of the octree in Figure 10. The value of each octant is its Z-order position. Units for the x, y, and z coordinates are etree ticks.

examples/txt2etree.c

```
1 #include "etree.h"
2
3 typedef struct payload_t {
4     int32_t val;
5     char tag;
6 } payload_t;
7
8 int main(int argc, char **argv)
9 {
10     etree_t *ep;
11     etree_addr_t addr;
12     int type, count=0;
13     payload_t payload;
14     char buf[ETREE_MAXBUF];
15
16     /* Create an empty 3d etree */
17     ep = etree_open(argv[1], O_RDWR|O_CREAT|O_TRUNC, 1, sizeof(payload_t), 3);
18
19     /* Register the schema for the etree to make it portable */
20     if (etree_registerschema(ep, "int32_t val; char tag;") != 0) {
21         fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
22         exit(1);
23     }
24
25     /* Insert each octant into the tree */
26     while (fgets(buf, ETREE_MAXBUF, stdin)) {
27         /* read an octant from the input, code skipped */
28         count++;
29         if (etree_insert(ep, addr, &payload) != 0) {
30             fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
31             exit(1);
32         }
33         /* echo the address and payload of the octant, code skipped */
34     }
35     printf("Loaded %d octants\n", count);
36
37     /* Add a comment to the dataset */
38     if (etree_setappmeta(ep, "Dataset Generated by txt2etree") != 0) {
39         fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
40         exit(1);
41     }
42     etree_close(ep);
43     return 0;
44 }
```

examples/txt2etree.c

Figure 12: **The `txt2etree` program.** Running `txt2etree < tree.txt` builds the octree from Figure 11 by iteratively inserting each octant into the etree. Note that we have omitted the error checking code after opening an etree (line 17) in order to fit this program in one page. In practice, the result returned by an etree library function should always be checked.

```
Loaded (0 2 1 31)L = C 8
Loaded (1 2 1 31)L = C 9
Loaded (0 3 1 31)L = C 10
Loaded (1 3 1 31)L = C 11
Loaded 17 octants
```

The `txt2etree` program demonstrates two key ideas about building etrees:

- *Octants can be inserted in any order.* The order that we insert octants into the tree does not affect the correctness of the resulting tree. While the example text file builds the tree using a depth-first refinement, we might have also inserted the octants bottom up, starting with the leaf nodes. Or we might have inserted the octants in preorder. In fact any ordering would have resulted in the identical octree.
- *Inserting octants in the preorder is most efficient.* Etrees are stored on disk as a linear sequence of equal-sized octant records in preorder. Thus, the most efficient way to insert octants into the etree is in preorder.

5.3 `query`: Querying Existing Octrees

The `query` program in Figure 13 interactively searches for octants in an existing etree. This program is handy for improving our understanding of octree searches. For example, suppose that the example octree from Figure 10 is stored in the etree file `tree.e`. Then searching for an existing parent octant yields that octant, as expected:

```
unix> ./query tree.e
Input  (x y z l): 2 2 0 30
Output (x y z l): (2 2 0 30)L = 12
```

Searching for a non-existent descendant of the parent yields the closest ancestor:

```
unix> ./query tree.e
Input  (x y z l): 3 3 0 31
Output (x y z l): (2 2 0 30)L = 12
```

However, not all queries are valid. For example, exceeding the maximum octant level results in an error:

```
unix> ./query tree.e
Input  (x y z l): 3 3 0 32
Octant level out of bounds
```

Similarly, searching for a non-existent octant that does not have ancestors in the etree also results in an error:

```
unix> ./query tree.e
Input  (x y z l): 3 3 0 30
Can't find the octant
```

examples/query.c

```c
#include "etree.h"

int main(int argc, char **argv)
{
    etree_t *ep;
    etree_addr_t addr, res_addr;
    int res_val;
    char buf[ETREE_MAXBUF], *metadata, *schema;

    /* Open the etree for reading */
    ep = etree_open(argv[1], O_RDONLY, 0, 0, 3);
    if (ep == NULL) {
        fprintf(stderr, "Could not open etree file %s\n", argv[1]);
        exit(1);
    }

    /* Print application meta data if it exists. */
    metadata = etree_getappmeta(ep);
    if (metadata != NULL) {
        fprintf(stderr, "Etree meta data: %s\n", metadata);
        free(metadata);
    }

    /* Print schema if it exists. */
    schema = etree_getschema(ep);
    if (schema != NULL) {
        fprintf(stderr, "Etree schema: %s\n", schema);
        free(schema);
    }

    /* Query each octant specified on stdin */
    printf("Input  (x y z l): ");
    while (fgets(buf, ETREE_MAXBUF, stdin)) {
        sscanf(buf, "%u %u %u %d", &addr.x, &addr.y, &addr.z, &addr.level);
        if (etree_search(ep, addr, &res_addr, "val", &res_val) != 0) {
            fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
            printf("Input  (x y z l): ");
            continue;
        }
        /* echo the search result and prompt for input, code skipped */
    }

    etree_close(ep);
    return 0;
}
```

examples/query.c

Figure 13: **The `query` program.** Interactively queries the octants in an etree.

The query program shows how to make use of a schema conveniently. Since txt2etree (Figure 12) has defined a schema for tree.e, the schema is restored by the etree library runtime system when tree.e is reopened by query. Lines 25–29 print the original schema definition text string. The print operation has no effect on the normal operations of the schema manager and can be safely omitted. Line 35 searches the etree database and extracts the field of interest val from the payload to local variable res_val. The etree library guarantees that any conversion necessary is conducted automatically. So the query program can run correctly on a Sparc/SunOS platform even if tree.e is generated on an IA32/Linux platform.

5.4 etree2txt: Traversing Existing Octrees

The etree library provides an efficient iterator mechanism for traversing the octants in an etree in preorder. It maintains the position of the current octant in an opaque object called a *cursor*, and it provides functions for initializing the cursor, retrieving the cursor, and advancing the cursor to the next octant.

The traverse function in Figure 14 demonstrates how to use the cursor mechanism. Lines 9–14 initialize the cursor to start at the beginning of the etree. Line 19 retrieves the address and payload of the current octant. Note how we use "*" to specify the retrieval of the entire payload. Line 27 advances the cursor to the next octant and detects the termination of the traversal. Lines 30–34 check whether the end of the etree has been reached.

The etree2txt program (not shown) uses the traverse function to print the octants of an arbitrary etree in preorder. If the example octree from Figure 10 is contained in the etree called tree.e, then running etree2txt yields the following output:

```
unix> ./etree2txt tree.e
Etree metadata: Dataset Generated by txt2etree
Etree schema: int32_t val; char tag;
Visited (0 0 0 29)I = 0 A
Visited (0 0 0 30)L = 1 B
Visited (2 0 0 30)L = 2 B
...
Visited (0 2 2 30)L = 15 B
Visited (2 2 2 30)L = 16 B
Dumped 17 octants
```

Since each octant in this tree was created with a value equal to its position in preorder traversal, we can see immediately from the sorted values that the traverse function is indeed visiting the octants in preorder.

5.5 refine: Building Etrees Recursively

The etree library treats octrees as a collection of independent octants that can be inserted and deleted in any order. This provides applications with the freedom to build octrees top-down or bottom-up, recursively or iteratively, and to store any combination of leaf and interior octants. For example, etree mesh generators and solvers store only the leaf octants[22], while etree visualization programs might use interior nodes to support multiple resolutions.

examples/traverse.c

```
int traverse(etree_t *ep)
{
    etree_addr_t addr;
    payload_t payload;
    int count;
    char buf[ETREE_MAXBUF];
    etree_error_t err;

    /* Get the initial cursor */
    addr.x = addr.y = addr.z = addr.t = addr.level = 0;
    if (etree_initcursor(ep, addr) != 0) {
        fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
        exit(-1);
    }

    /* Interatively traverse the tree using the cursor mechanism */
    count = 0;
    do {
        if (etree_getcursor(ep, &addr, "*", &payload) != 0) {
            fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
            exit(-1);
        }
        count++;

        printf("Visited %s = %d %c\n", etree_straddr(ep, buf, addr),
               payload.val, payload.tag);
    } while (etree_advcursor(ep) == 0);


    if ((err = etree_errno(ep)) != ET_END_OF_TREE) {
        /* make sure that the cursor terminates correctly */
        fprintf(stderr, "%s\n", etree_strerror(err));
        exit(-1);
    }

    return count;
}
```

examples/traverse.c

Figure 14: **The `traverse` function.** Traverses the octants in an etree in preorder.

In Section 5.2 we developed a program called `txt2etree` that constructed an octree iteratively, one octant at a time. In this section, we explore a more traditional approach where the octree is constructed by a recursive depth-first refinement of the tree. The decision about whether to split an octant is made by an application-specific *refinement predicate*.

We develop a program, called `refine.c` that builds the octree in Figure 10 using recursive depth-first refinement. After creating an empty etree, we begin the operation by inserting a single level-29 octant that covers the domain of interest:

```
    root.x = root.y = root.z = 0;
    root.level = ETREE_MAXLEVEL - 2;
    root.type = ETREE_LEAF;
    payload.tag = 'A';
    payload.val = 0;

    if (etree_insert(ep, root, &payload) != 0) {
        fprintf(stderr, "%s", etree_strerror(etree_errno(ep)));
        exit(1);
    }
```

Next we call the refine function. After the refinement is complete, we traverse the tree:

```
    refine(ep, root);
    count = traverse(ep);
    fprintf(stderr, "The tree has %d octants\n", count);
```

Figure 15 shows the `refine` function. Line 9 checks the terminating conditions for the recursion. The input root octant will not be refined any further if it is already as small as possible, or if the application-specific refinement predicate returns `false`.

Before refining the children, we must change the root octant from a leaf to an interior. Notice that there is no way to do this in place. We must first delete the old octant and then insert a new one with the proper type.

After changing the root from a leaf to an interior octant, we initialize the level of the children (line 23), set their type to `ETREE_LEAF` (line 24), and compute the edge size of the children (line 25). The edge size of the children is computed using the rule that the edge length of an octant at level ($maxlevel - k$) is equal to 2^k ticks.

Lines 27–41 enumerate the children in Z-order, inserting each child into the tree, and then recursively refining it. Notice that we can iterate in Z-order by varying the x coordinate the fastest, followed by the y coordinate, followed by the z coordinate. Study this loop nest carefully to make sure you understand it.

When called with the following refinement predicate,

```
int refine_pred(etree_addr_t addr, int val) {
    if ((val == 0) || (val == 3))
        return 1;
    else
        return 0;
}
```

examples/refine.c

```
void refine(etree_t *ep, etree_addr_t root)
{
    int i, j, k, incr;
    etree_addr_t child;
    static int val = 0;
    payload_t payload;

    /* Check the terminating conditions for the recursion */
    if ((root.level >= ETREE_MAXLEVEL) || (!refine_pred(root, val)))
        return;

    /* Make the root an interior node */
    if (etree_delete(ep, root) != 0) {
        fprintf(stderr, "%s\n", etree_strerror(etree_errno(ep)));
        exit(1);
    }
    root.type = ETREE_INTERIOR;
    payload.val = val;
    payload.tag = 'A' + (root.level - (ETREE_MAXLEVEL - 2));
    etree_insert(ep, root, &payload);

    /* Edge of octant at level (ETREE_MAXLEVEL-k) is 2^k ticks */
    child.level = root.level + 1;
    child.type = ETREE_LEAF;
    incr = 1 << (ETREE_MAXLEVEL - child.level);

    /* Recursively expand the children in z-order */
    for (k = 0; k <= incr; k += incr) {
        child.z = root.z + k;
        for (j = 0; j <= incr; j += incr) {
            child.y = root.y + j;
            for (i = 0; i <= incr; i += incr) {
                child.x = root.x + i;
                payload.val = ++val;
                payload.tag = 'A' + (child.level - (ETREE_MAXLEVEL - 2));
                etree_insert(ep, child, &payload);

                refine(ep, child);
            }
        }
    }
    return;
}
```

examples/refine.c

Figure 15: **The `refine` function.** Constructs an octree using recursive depth-first refinement. The refinement decision is controlled by the application-specific refinement predicate `refine_pred`. Note that we have omitted the error checking code for `etree_insert` to fit the program in one page. In practice, the result returned by an etree library function should always be checked.

the `refine` function will build the examples tree from Figure 10. For example:

```
unix> ./refine tree.e
Visited (0 0 0 29)I = 0 A
Visited (0 0 0 30)L = 1 B
Visited (2 0 0 30)L = 2 B
...
Visited (0 2 2 30)L = 15 B
Visited (2 2 2 30)L = 16 B
The tree has 17 octants
```

6 Internal mechanisms

In this section, we focus on the internal mechanisms of the etree library that implement the API functions we just described.

6.1 Linear Octrees and Locational Codes

Although the etree address space (Section 3) and the etree octant address present a nice abstraction to identify an octant at the application level, the issues of how to index and access the octants remain. Our solution is to use the *linear octree* technique [8] to assign an internal key to each octant and then use the well-known B-tree structure to index these keys.

The basic idea of the linear octree is to encode each octant by a key called the *locational code* that uniquely identifies the octant. There are two different types of locational codes: variable-length and fixed-length. Conceptually, both schemes derive the locational code for an octant by concatenating the directional codes on the path from the root to some target octant. We refer to these concatenated bits as the *path information*. The difference between the two schemes lies in the number of bits used to encode the path information.

In the variable-length scheme, octants at different levels use a different number of bits to encode their path information. For example, in Figure 16, the path information for octant k is 1010, while path information for octant g is 100100. Since the path information can uniquely identify an octant, the variable-length scheme actually uses the path information directly as the locational code for an octant.

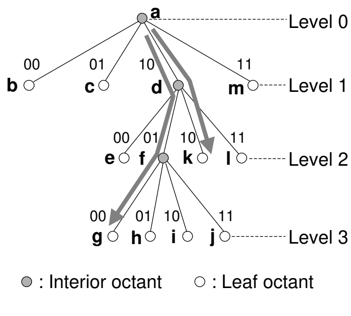


Figure 16: **Variable-length locational codes.** Concatenate the directional codes on the path from the root to the target octant.

In contrast, the fixed-length scheme uses the same number of bits to encode the path information for all octants. Obviously, the number of bits should be sufficient to encode the leaf octant at the maximum allowable level. For octants at higher levels, zeroes are padded to extend the path information to the specified length. For example, the maximum allowable level of the octree in Figure 3 is four. Thus the fixed-length path information for octant k is 10100000, with the trailing four zeroes padded to its variable-length counterpart. Similarly, octant g has a fixed-length path information of 10010000.

However, notice that octant f, the parent of octant g, produces the same bit string of 10010000 because zeroes are padded to the its variable-length path information. Therefore, in order to distinguish the trailing zeroes between the directional codes and the padded zeroes, additional information is needed to identify the level of an octant. Usually, the level of an octant is appended to its path information to form a complete fixed-length locational code. As a result, the total number of bits of a fixed-length locational key is $2 \cdot n + \lceil \log_2 n + 1 \rceil$, where n is the maximum allowable level. The first $2 \cdot n$ bits are used to store the path information. The last $\lceil \log_2 n + 1 \rceil$ bits are used to record the level of the octant.

In the etree library, we adopt the fixed-length scheme because the locational codes thus constructed can be treated as fixed length byte strings, which is a requirement for indexing by B-trees. For brevity, we will use the term "locational code" to refer to the fixed-length locational code.

6.2 Using Morton Codes to Construct Locational Codes

The concatenation method we described above is only a conceptual model. In practice, there is no need to actually build the entire octree and traverse the path from the root to a target octant. Instead, we can derive an octant's path information directly from its etree octant address. The idea is based on the well-known *Morton code* [14].

A Morton code maps a point in a d-dimensional space to a scalar integer. The mapping is computed by interleaving the bits of the binary representations of the d (fixed-length) coordinates of the point. By computing the Morton code of an octant's left lower corner pixel, we can obtain its path information (with possible zero paddings).

To better understand why this is the case, let us first look at an example that shows how to derive the path information for octant f in Figure 3(b). Even though octant f is not labeled in Figure 3(a), we know that its left lower corner pixel is the same as that of its child in the left lower corner, that is, octant g. So the pixel to be processed has a coordinate $(4, 8)$. The bit-interleaving is shown in Figure 17. It is easy to verify that the resulting Morton code is the same as the fixed-length path information for octant f.

In general, when we interleave the most significant bits of the x and y coordinates of a pixel, the resulting two-bit combination is either 00, 01, 10, or 11. Obviously, if the pixel's x coordinate is in the right half of the root octant, which occupies the entire domain, then the most significant bit of x must be 1. For example, in Figure 3(a), the x coordinate of any pixel in the right half of the domain must have a value in the range of 8 to 15, whose binary representations all have a most significant bit of 1. Conversely, if the most significant bit of x is 0, then the pixel must be in the left half of the root octant.

Similarly, if the pixel's y coordinate is the upper half of the root octant, the most significant bit of y must be 1. Otherwise, it is 0. Therefore, the four combinations of 00, 01, 10 and 11 correspond exactly to the definition of the directional code in Figure 3(b). In other words, the interleaving of the most significant bits

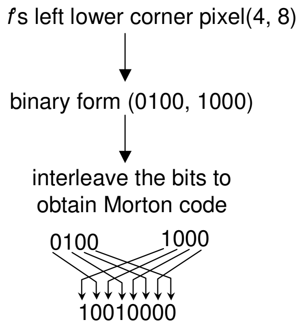


Figure 17: **Computing a Morton code.** Interleaving the bits of the left lower corner pixel of an octant computes its path information.

produces the correct directional code for the first edge on the path from the root to the pixel. By repeatedly applying this analysis to the second most significant bits, the third, the fourth, and so on, we conclude that the Morton code of a pixel is indeed the same as the path information for the pixel.

Next, we prove that the path information for an octant is the same as that of its left lower corner pixel. We observe that the left lower corner pixel of an octant is always the same as the left lower corner pixel of its child in the left lower corner, that is, the child reached through an edge with a directional code of 00.

Suppose that we start from an octant, say octant $\mathcal{O}$, and traverse down the octree (tree representation), following the edge labeled 00 at each step. We will finally reach either a pixel octant or a leaf octant larger than a pixel. In the latter case, we can conceptually expand the octree to the maximum possible level from the leaf octant and keep on following the 00 edge until a pixel is reached. In either case, we can traverse a path from octant $\mathcal{O}$ down to its left lower corner pixel. All the directional codes on this path are 00. Obviously, the path information for the left lower corner pixel consists of two parts: the concatenation of the directional codes from the root to octant $\mathcal{O}$, and the concatenation of the directional codes from octant $\mathcal{O}$ to the pixel. The second part, as we have shown, is all zeros. Recall that the path information of octant $\mathcal{O}$ is obtained from the concatenation of the directional codes from the root to octant $\mathcal{O}$ padded with zeros to the specified length. Therefore, the path information for octant $\mathcal{O}$ is exactly the same as that of its left lower corner pixel, which is equal to its Morton code.

To construct the complete locational code, we append the level of the octant to the Morton code. Although the maximum allowable etree level of 31 only requires 5 bits, we allocate one byte to record the level. This choice simplifies the implementation, since we can directly append an extra byte to the end of the byte sequence representing the Morton code. Furthermore, we are given three extra bits that can be used for other purposes. For example, in the current implementation, we use the most significant bit of the appended byte (the least significant byte of the locational code) to record the type (leaf or interior) of the octant at that location.

To summarize, the etree library constructs the locational code from the application-level etree octant address structure as follows:

- First, extract the coordinate of the octant's left lower corner pixel from the application-level etree octant address.

- Next, interleave the bits of the coordinate to produce the Morton code.
- Finally, extract and store the level information in one byte, and append this byte to the end of the Morton code.

To this point, we have shown the mechanics of deriving the locational code for an octant. Next we present on an interesting and powerful property of the locational code that we exploit in the etree library.

6.3 The Preorder Traversal Property of Locational Codes

As we have seen, the purpose of the constructing the locational codes is to assign internal keys to octants that can be used for indexing and accessing octants. Therefore, we must be able to define a comparison function for the locational codes.

In the context of key comparison, we ignore the original semantics of the locational code as a concatenation of the directional codes followed by the level. Instead, we view a locational code as a byte sequence. For the purpose of comparison, we treat the byte sequence as a large positive integer. In reality, the length of the locational code byte sequence may be longer than any integer data type defined in any programming language. For example, the etree library represents a locational code internally as a 13-byte sequence. To enable the comparison between two conceptually large integers, we perform a byte-wise comparison, starting from the most significant byte of the sequence down to the least significant byte. Each byte pair comparison is conducted as if both bytes contain unsigned short (8 bits) integers.

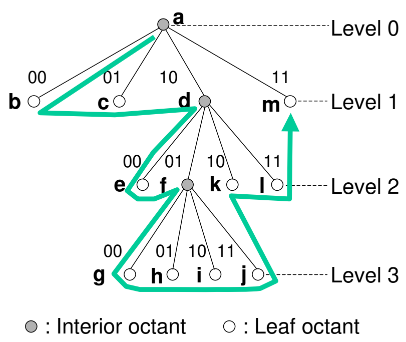
Using this comparison function to sort the octants of an octree in increasing locational code order, we find the ordering of octants is exactly the same as that processed by the *preorder traversal* of the octree. By preorder traversal, we mean visiting the root, and then recursively visiting its children in directional code order.

The restriction on the order of processing the subtrees guarantees a unique preorder traversal sequence. The preorder traversal of our example octree is shown in Figure 18(a). For convenience we have labeled the octants with an alphabetical order consistent with the preorder traversal. Figure 18(b) shows the locational code for each octant in Figure 18(a).

Figure 18 illustrates a neat fact about octrees: *The ordering induced by the locational codes is the same as the ordering induced by a preorder traversal of the octree.* To show this, we prove the following three points: (1) The root octant's locational code is the minimum among all the octants; (2) A subtree root octant's locational code is less than any of its children; and (3) If octant A's locational code is less than octant B's, then any descendant of A has a locational code less than that of any descendant of B. The first point guarantees that the root octant is the first to be produced in the ascending ordering of the locational codes. The second point guarantees that a subtree root is always produced before any of its children. And the third point guarantees that a subtree is always completed visited before moving to its sibling subtrees. These guarantees comprise the same definition of the preorder traversal.

Our proof is straightforward if we interpret the locational code using its original semantics. The first point is trivially true because the root octant's locational code consists of padding zeroes only.

For the second point, let us assume that a subtree root octant is at level k and that the maximum allowable level of the octree is n. Then the trailing $2 \cdot (n - k)$ bits of the path information are padded with zeros.



Octant	Locational code
a	00000000_000
b	00000000_001
c	01000000_001
d	10000000_001
e	10000000_010
f	10010000_010
g	10010000_011
h	10010100_011
i	10011000_011
j	10011100_011
k	10100000_010
l	10110000_010
m	11000000_001

(a) Preorder traversal (b) Locational codes

Figure 18: **Preorder traversal of the octree.** The ordering induced by a preorder traversal is the same as the ordering induced by the locational code. The underscores in the locational codes separate the path information from the octant level. They are included for presentation purposes only. Ignore them and view the bits as representing a positive integer value.

Therefore, all the children except the one reached through edge 00 have at least one non-zero bit in the trailing $2 \cdot (n - k)$ bits. Hence, the byte-wise comparison function will find the inequality and report that the children octants have greater locational code than the subtree root. For the child reached through edge 00, even though its path information is the same as the subtree root, it still has a greater locational code than the subtree root since it has level $k + 1$. This will be discovered when the last bytes (the least significant bytes) of the two locational codes are compared. *An important corollary of this fact is that the locational code of an subtree root is less than the locational codes of its descendants.*

For the third point, assume that the higher level between octant A and octant B is k. Then the leading $2 \cdot k$ bits of octant A's locational code is less than that of octant B. Because the locational code of any descendant of A has the same leading $2 \cdot k$ bits as A, the locational code of any of A's descendants is less than that of B, and thus less than those of B's descendants. The last step of deduction uses the corollary developed in the previous paragraph.

This completes our proof of our claim, which we refer to as the *preorder traversal property* of the locational code. Next we show two important applications of this property in the etree library.

6.4 Applications of the Preorder Traversal Property

The preorder traversal property plays a key role in two functions of the etree library:

1. Clustering nearby octants on disk pages.

2. Finding a leaf octant without knowing its exact locational code.

The first function helps improve the performance of the etree library. When the interior octants are ignored, then the preorder traversal of the leaf octants corresponds to a space-filling curve in the domain. This intuitively validates our choice of the domain representation of the octrees. In particular, the space-filling curve follows a Z pattern, as shown in Figure 19. Such an ordering is called the *Z-order* [16, 15]. It has been proven that the Z-order tends to cluster spatially close octants in the one-dimensional ordering [7]. Therefore, we can store the leaf octants sequentially on the disk pages (B-tree leaf pages) according to their locational codes, which naturally results in the clustering of nearby octants, thus improving disk I/O efficiency.

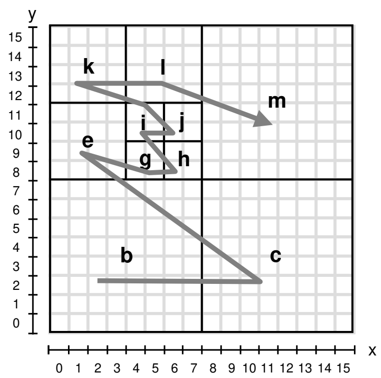
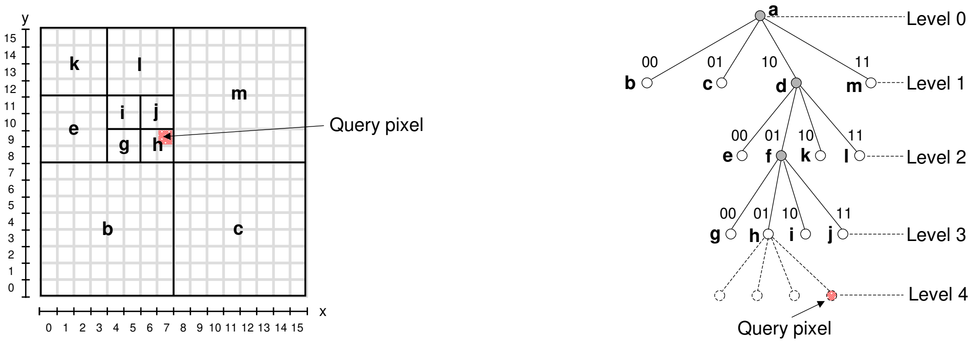


Figure 19: **Z-order curve through the domain.**

The second function enhances the functionality of the etree library, which is particularly useful when an octree is employed for domain decomposition. For example, when an octree database is generated to record the material properties of some region of the earth, only leaf octants are stored in the database, where each octant represents a contiguous chunk of soil with roughly the same density throughout. A common operation on this octree database is a point query. That is, given the coordinate of a *query pixel* (point) in the domain, what is the material property associated with that query pixel. The etree library should be able to retrieve the leaf octant that encloses the query pixel. But how do we find the leaf octant without knowing its exact locational code?

Suppose we have a query pixel surrounded by an octant h, as shown in Figure 20(a). We would like to know the material property associated with the query pixel, which in this is the same as the material property associated with h.

Now suppose that we virtually expand the enclosing leaf octant h to the maximum allowable level of the octree, as shown in Figure 20(b). The query pixel thus becomes a tiny octant at the deepest level in this virtual subtree. The preorder traversal guarantees that (1) The subtree root be visited before any descendants, and (2) The traversal of a subtree be completed before processing the next subtree (which may be either a leaf octant or an interior octant). Therefore, we know by (1) that the locational code of the query pixel is greater than the locational code of h, and by (2) that the locational code of the query pixel is less than any existing locational codes that are greater than that of h. Given these observations, we can use the locational code of the query pixel as a search key and ask the B-tree to return the octant whose locational code is the maximum among all the locational codes that are less than the search key. Thus we are guaranteed to locate the leaf octant as long as the query pixel is enclosed in the leaf octant.



(a) Query pixel enclosed by leaf octant h (b) Virtual expansion of leaf octant h

Figure 20: **Finding a leaf octant without knowing its exact locational code**

This has an important implication for applications that are querying the etree. That is, applications can query arbitrary points without any knowledge of the underlying structure of the octree.

7 Using Etrees for Physical Simulations

As we mentioned in the introduction, the etree library is part of a larger research effort, known as the Euclid Project, aimed at developing the capability to perform physical simulations by operating on databases. In particular, we are targeting simulations based on structured finite differences and unstructured finite elements. The goal is to allow scientists and engineers to run large simulations on their desktop machines, thereby deferring the point at which they must move their codes to a supercomputer center.

The physical simulation process involves iterations over the three steps shown in Figure 21: (1) *mesh generation*, which models a continuous problem domain with a discrete structure; (2) *solving*, which simulates the physical process by approximating the solution of a set of partial differential equations at the grid points of the mesh; and (3) *visualization*, which creates a visual representation of the simulation results. Each iteration of the procedure furthers the understanding of the physical system, and provides a basis for parameter tuning for the next iteration.

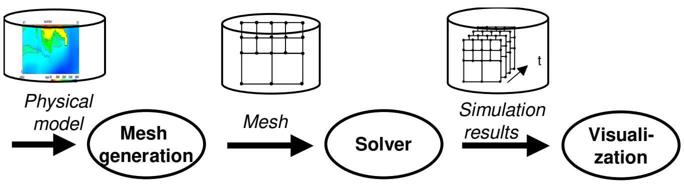


Figure 21: **Steps involved in one iteration of the simulation process.**

The basic idea is to store all of the data involved in the simulation process – the material model, the finite element meshes, and the 4D output data sets – as etree databases. The steps in the simulation process are performed by querying and updating these databases on disk. This section outlines these ideas in more detail.

7.1 Octree Mesh Generation

One important application for the etree library is to support large octree mesh generation. We have developed a prototype tool chain [22] that can be used to generate large finite element meshes on desktop machines with limited memory.

Figure 22 shows the process of generating an octree mesh using the etree library. In the *construct* step, the etree address space is decomposed into an octree according to the geometry or physics of the corresponding real-world problem domain. Since only leaf octants are transformed to finite elements, the interior octants are not stored in the result database. We refer to the result database as an *unbalanced octree* because the neighbors of an octant may be arbitrarily larger or smaller than the octant. To guarantee good element quality, the next step performs a *balance* operation, which repeatedly decomposes large leaf octants into smaller sizes until a so-called *2-to-1 constraint* is satisfied. The 2-to-1 constraint requires that two leaf octants sharing a face or an edge be no more than twice as large or small. After a *balanced octree* is derived, a *transform* procedure is invoked to collect mesh-specific information such as the element-node relationship and the node coordinates, which are stored in two different etree databases, one for the mesh elements, and the other for the mesh nodes.

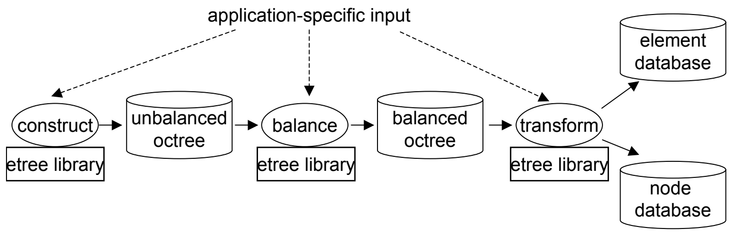


Figure 22: **The etree method of generating octree meshes on disk.**

Throughout the tool chain, we use the etree library as a *paging system* to swap octants between the memory and the disk. That is, instead of querying the etree for each data access, we input and output octree data to and from the etree in bulks. One research issue that we currently focus on is what kind of strategies of bulk loading/dumping are the best to exploit the locality of reference such that the overall end-to-end running time can be improved.

Our experiment results have shown that the etree-based tool chain can efficiently (2.6 hours) generate very large finite element octree meshes (4.3GB with over 15 million mesh grid points/nodes) on a machine with only 128MB main memory. In practice, this tool chain is being used by the Mechanics, Algorithms, and Computing Laboratory of Civil Engineering Department of CMU to generate large finite element meshes for earthquake ground motion simulations for the LA basin.

7.2 Finite Element Solver

Although this part of the project is still in its fledgling stage, it is of utmost importance to the final goal of enabling a complete simulation process on a desktop machine.

The basic idea is to iterate through the finite element octree mesh and process the elements one by one. For each element, calculate its contribution (in terms of force or other physical quantity) to the associated mesh grid points/nodes. The contribution is aggregated on the grid points stored in a intermediary etree. When all the elements are processed, the intermediary etree represents the complete nodal force vector. Then the physical properties (in terms of mass or other quantity) of each grid point/node are retrieved to assimilate the impact of the corresponding force. This approach treats the etree like an unbounded memory buffer. The only limits on the size of the problem being solved is the size of the hard disk space, which we believe is not a scarce resource to obtain.

We have developed a prototype octree finite element solver using the etree library. Non-trivial validation runs are carried out and the results match those generated by a conventional in-core solver developed by other engineers.

However, to create robust out-of-core solvers for production use, we need to understand many unsolved questions. For example, shall we follow the Z-order through the domain to process each element? If the Z-order is not the best solution, what are the alternatives? How does the order of traversing the domain affect the strategy of caching of the intermediary results? How will etree library perform when the problem size becomes extremely large? What procedure shall we take to avoid undesirable floating-point roundoff errors? All these are open research questions that are critical to the success of an out-of-core finite element solver. Without clear answers to these questions, worse case situations may happen where either every data access would cause a disk I/O; or the results are garbled by roundoff errors.

7.3 3D/4D Dataset Compression

An important extension to the functionality of the etree library is the compression of 3D and 4D datasets. This functionality is extremely useful for applications with large amount of redundant data. For example, Figure 23 shows the material property of a cross section of the Los Angeles basin, which is extracted from an 3D octree database built with the etree library. It can be seen that deep underneath the surface, the material properties of rocks are homogeneously hard. It is desirable to be able to compress these similar chunks of ground to save the storage space and speed up queries. Figure 24 shows the velocity response

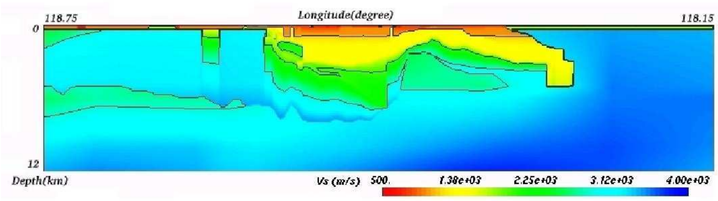


Figure 23: Density cross section of the Los Angeles basin [12].

of an observation point during an earthquake ground motion simulation for the Los Angeles basin[2]. The horizontal axis records the time in terms of second since the beginning of the simulation. The vertical axis record the velocity of at the observation point along the X, Y, and Z direction, respectively. Even though these plots are for one observation point, the data stored in the simulation output, a 4D dataset with time as the fourth dimension, actually contains the movement of every grid point/node in the domain. It is clear from the above plot that there exists great opportunity to compress the 4D simulation result dataset temporally. Since it takes time for the wave to propagate from the epicenter a particular area, the grid points in that area

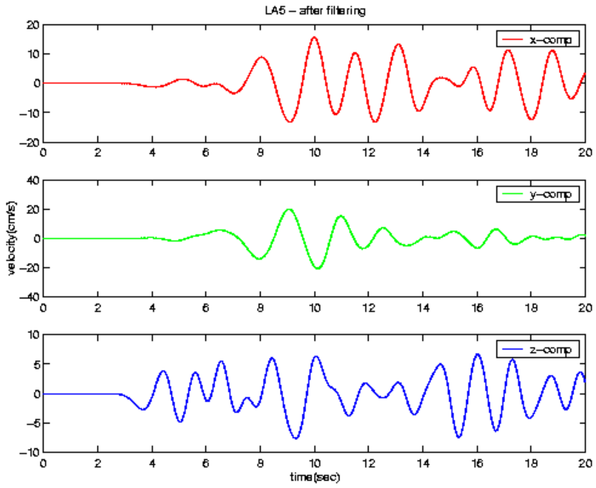
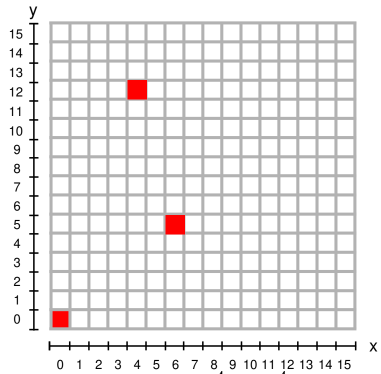


Figure 24: Velocity response of an observation point [11].

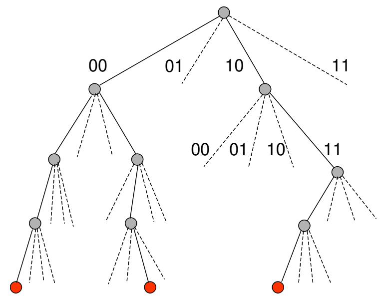
remain stationary during this time period. This is validated by the flat segment of the curve at the beginning few seconds. Meanwhile, a grid point may be shaking more or less similarly to its adjacent neighbors (other grid points). There also exists the opportunity to compress the results spatially.

Even though the conventional interpretation of octrees only applies to the 3D dataset, we can treat the 4D dataset as a special type of octree (hextree in 4D) that we call a *sparse octree*. Correspondingly, we refer to the conventional octree as *dense octree*. The difference between a sparse octree and a dense octree is that the former *may* have NULL children at an interior node while the latter always has full number of children (8 for octree, 16 for hextree) at each interior node. However, the maximum number of children at interior node of a sparse octree is bounded by the 8 (or 16 for hextree). All of the examples presented earlier in this paper were of dense octrees.

Figure 25 shows an example of a sparse octree that represents a point set consisting of 3 pixels at random locations. The dashed links indicate NULL children branches at the interior nodes.



(a) Point set on a $2^4 \times 2^4$ domain.



(b) Sparse octree representing the point set.

Figure 25: **Using a sparse octree to represent an arbitrary point set.**

We now describe how to construct an sparse octree conceptually. Initially, a sparse octree only has the root octant. For each new pixel, we compute the pixel's Morton code, which is the same as the path information for the pixel octant. Then we repeatedly extract two bits from the Morton code, starting from the most significant bit. These two-bits pairs correspond to the directional codes on the path from the root octant to the pixel octant. At each hop down the path, if the child octant does not exist, create a new octant and set a pointer from the parent to the child. Otherwise, we just follow the edge to the child node. When the bits in the Morton code are exhausted, we know we have reached the pixel octant.

In order to work in the abstract etree address space, we need to first map the points of an arbitrary 4D dataset to the etree address space. Usually, the mapping can be done without causing collision. That is, no two points map to the same pixel. For most applications, our current choice of a 31-bit etree address space is sufficiently refined to distinguish any two points in the domain. For example, if the domain has a maximum dimension of 200km, the finest resolution provided by the etree address space equals to 200km / 2^{31}, which is approximately 100 microns.

Once the mapping is done, we can interpret the point set as a sparse octree. Therefore, by overcoming the barrier of interpreting 4D dataset and extending the definition of octrees, we transform the obscure problem of compressing 4D datasets to a much more clearly defined question. That is, how to compress a sparse octree. Also note that once this problem is solved, the compression of dense octrees is only a special case where the interior nodes happen to have full number of children. We expect to find an elegant solution in the near future.

8 Related Work

Octree techniques have been applied in many different areas to solve important problems. [18] provides a survey of the applications of octree techniques in the area of computer graphics, image processing and geographical information system.

In the context of physical simulations, the octree provides a good compromise between the modeling power and the structure simplicity. It has been proven to be a successful strategy for generating three-dimensional adaptive meshes. There are two different approaches. The first one [26, 27, 20, 4, 13] uses an octree to discretalize a problem domain and then warp the vertices/edges of the leaf octants to generate a triangular or tetrahedral mesh. The second one [28, 24, 11] uses the octree structure as a hexahedral mesh directly, that is, the leaf octants are treated as finite elements without modification. To ensure element quality, both approaches require the octree be balanced.

In terms of the solvers, [17] presents a parallel, out-of-core octree-based method for N-body simulation. The key idea is to use *out-of-core pointer* to access octants stored on disk. A out-of-core points is represented in the form of $\langle$*disk page number, offset*$\rangle$. The operation of following a pointer in an in-core algorithm is transformed to seeking an offset on a disk page and then retrieving the data object. To increase spatial locality and cache hit rates [25], octants are clustered on disk pages in breath-first traversal order, and at each level, in the *Peano-Hlibert* order [7] ,

Besides the fact the etree library is a runtime system to support a wide variety of octree-based applications instead of a particular application solution, the etree library differs from the out-of-core N-body octree method in two major ways. First, octants are accessed by searching their internal keys in the etree library.

No pointer-chasing is necessary. Second, octants are clustered on disk pages in the (depth-first) preorder traversal order that is equivalent to the Z-order. We note that the optimal clustering of the octants are very much application-dependent. It is impossible to conclude which clustering strategy is the better. However, with the framework of the etree library implementation, it is convenient to extend the functionality to support the breath-first, Peano-Hilbert ordering.

In the area of visualizing large dataset, [23] proposes an out-of-core octree approach for streamline visualization on large unstructured meshes. As a prepossessing step, an octree is used to partition the dataset (cells) into (more or less) equal-sized data blocks, each stored in a separate file whose name is recorded in the corresponding leaf octant. The assumption is that the octree structure is small enough to totally fit in the main memory. At runtime, the octree structure is loaded into the main memory and serves as an application-level paging system. Leaf octants needed by the streamline construction algorithm may trigger disk I/O if the data blocks they pointed to are not in the memory buffer. Experiment results show that the out-of-core approach outperforms the in-core virtual memory based approach by two orders of magnitude.

In contrast, the etree library makes no explicit use of the octree structure. Thus there is no restriction on the size of the octree (structure) that can be processed. Moreover, the etree library decouples the low-level disk I/O from the application programs by a clean interface.

9 Summary

The etree library allows C programmers to build and manipulate large octrees on disk. The library exports 24 functions for creating, modifying, and searching octrees, including efficient mechanisms for appending octants and iterating over octants in Z-order.

Octrees are stored in database files called *etrees*, which are portable across different platform. An etree is a sequence of fixed-sized octant records that are sorted in a locational code order and indexed by a conventional B-tree. Octant records are accessed using a fixed-length locational code key. The main idea is that the octant address encodes its size and position in the domain. There is no need to store the edges in the octree.

The etree library is the foundation for a larger research effort aimed at enabling scientists and engineers to solve large physical simulations on their desktop systems by reworking the entire simulation process to operate directly on etrees stored on disk. We have used the library to build efficient out-of-core octree mesh generators and have developed prototype finite element solvers and visualization tools based on the same techniques.

Acknowledgements

We would like to thank the following people who have contributed to the etree library and its applications: Christos Faloutsos, Natassa Ailamaki, Jacobo Bielak, Omar Ghattas, Eui Joon Kim, Kim Olsen, and Hongsuda Tangmunarunkit. Special thanks to Tom Jordan and Phil Maechling at the Southern California Earthquake Center for their support and encouragement.

Availability

Source code and documentation for the etree library are available on the Web at `www.cs.cmu.edu/~euclid`.

References

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision,Graphics,and Image Processing*, 24:1–13, 1983.

[2] H. Bao, J. Bielak, O. Ghattas, L. Kallivokas, D.R. O'Hallaron, J. Shewchunk, and J. Xu. Large-scale simulation of elastic wave propagation in heterogeneous media on parallel computers. *Computer Methods in Applied Mechanics and Engineering*, 1998.

[3] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[4] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. In *Proceedings of 31st Symposium on Foundation of Computer Science*, pages 231–241, 1990.

[5] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice-Hall, 2003.

[6] Douglas Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, Jun 1979.

[7] Christos Faloutsos and Shari Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMID-SIGART Symposium on Principles of Database Systems (PODS)*, 1989.

[8] Irene Gargantini. An effecive way to represent quadtrees. *Communicatoins of the ACM*, 25(12):905–910, Dec 1982.

[9] Irene Garnagntini. Linear octree for fast processing of three-dimensional objects. *Computer Graphics,and Image Processing*, 20:365–374, 1982.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*, chapter 15. Morgan Kaufmann Publishers, Sep 1992.

[11] Eui Jiong Kim. *Terascale Ground Motion Simulation Using Octree-Based Adaptive Mesh*. PhD thesis, Dept of Civil and Environmental Engineering, Carnegie Mellon University, January 2003.

[12] Harold Magistrale, Steve Day, Robert W. Clayton, and Robert Graves. The SCEC Southern California reference three-dimensional seismic velocity model version 2. *Bulletin of the Seismological Soceity of America*, Dec 2000.

[13] Scott A. Mitchell and Stephen A. Vavasis. Quality mesh generation in three dimensions. In *Proceedings of the Eighth Symposium on Computational Geometry*, pages 212–221, Feb 1992.

[14] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[15] Jack. A. Orenstein. Spatial query processing in an object-oriented database system. In *Proceedings of ACM SIGMOD*, pages 326–336, Washington D.C, 1986.

[16] Jack. A. Orenstein and T. H. Merrett. A class of data structure for associative searching. In *Proceedings of ACM SIGACT-SIGMOD*, pages 181–190, Waterloo,Ontario,Canada, 1984.

[17] John Salmon and Michael S. Warren. Parallel, out-of-core methods for n-body simulation. In *Proceedings of the Eighth SIAM Conference on Parallel Processings for Scientific Computing*, 1997.

[18] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing and GIS*. Addison-Wesley Publishing Company, 1990.

[19] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, 1990.

[20] Mark S. Shephard and Marcel K. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engieering*, 32, 1991.

[21] A. Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*, chapter 11. McGrill Hill Companies, Inc., third edition, 1997.

[22] Tiankai Tu, David O'Hallaron, and Julio Lopez. Etree: A database-oriented method for generating large octree meshes. In *Proceedings of the Eleventh International Meshing Roundtable*, pages 127–138, Ithaca, NY, Sep 2002.

[23] Shyh-Kuang Ueng, Christopher Sikorski, and Kwan-Liu Ma. Out-of-core streamline visualization on large unstructured meshes. *IEEE Transcations on Visualization and Computer Graphics*, 3(4):370–380, 1997.

[24] Jianlin Wang. Octree-based finite element method for elastic wave propagation with application to earthquake ground motion. Master's thesis, Department of Civil and Environmental Engineering,Carnegie Mellon University, May 1999.

[25] Michael S. Warren and John Salmon. A portable, parallel versatile n-body tree code. In *Proceedings of the Seventh SIAM Conference on Parallel Processings for Scientific Computing*, 1995.

[26] Mark A. Yerry and Mark S. Shepard. A modified quadtree approach to finite element mesh generation. *IEEE Computer Graphics and Applications*, pages 39–46, 1983.

[27] Mark A. Yerry and Mark S. Shepard. Automatic three-dimensional mesh generation by the modified-octree technique. *International Journal for Numerical Methods in Engineering*, 20:1965–1990, 1984.

[28] D. P. Young, R. G. Melvin, M. B. Bieterman, F. T. Johnson, S. S. Samant, and J. E. Bussoletti. A locally refined rectangular grid finite element: Application to computational fluid dynamics and computational physics. *Journal of Computational Physics*, 92:1–66, 1991.

A The Portable Etree File Format

Users should be able to share and exchange etree databases even if they are using different types of machines, e.g., a SPARC running Solaris or an Intel Pentium running Linux. Additionally, it should be easy to identify the content and other properties of a database, e.g., its creation time, author, application that generated it, schema of the payload if one is defined, region of the ground that it describes and other application specific parameters.

Etree files are self contained and include information to make the file portable between machines with different architectures and operating systems. An etree file contains all the information relevant to an etree, including information that describes various features of the file like data type sizes and byte ordering, and application metadata.

An etree file is divided into three sections: *etree header*, *etree data* and *application metadata*.

Etree header: The etree header contains information needed by the etree library to manipulate the etree file. The header is located at the beginning of the file at offset 0. The etree header contains the fields shown in Figure A.

Octree data: This section of the file contains the actual database storing application octree data. We expect this section of the file to be very large as it contains the actual application data.

Application metadata: This section contains the application defined metadata. The application metadata is an arbitrary text string. The etree file format does not impose any restriction on the data stored in this section. The application metadata is appended to to octree data at the end of the etree file.

Offset	Name	Size	Type	Description
0	Tree format	1	`char`	The endianness of the etree. Either "L" or "B".
1	Version	4	`uint32_t`	The library version that generates this etree
5	Dimension	4	`uint32_t`	The dimensionality of this etree
9	Rootlevel	4	`uint32_t`	The root level of the underlying octree
13	Appmetasize	4	`uint32_t`	Application metadata size. 0 if no application metadata is set/defined.
17	Statistics	512	`uint32_t array`	Statistics on the number of leaf octants and interior octants at different levels of the octree.
529	Internal use	1	`byte`	Reserved for internal use.
530	Pagesize	4	`uint32_t`	Size of database pages. It is usually set to be the disk block size of the file system where the etree was first created.
534	Pagecount	8	`uint64_t`	Number of database pages used to store the underlying octree.
542	Root page number	8	`uint64_t`	Database index structure root page number.
550	Key size	4	`uint32_t`	The size of the locational key of this etree.
554	Payload size	4	`uint32_t`	The size of the payload of each octant.
558	Schema size	4	`uint32_t`	The size of the internally represented schema.
562	Schema	variable	`ASCII string`	Internal representation of a schema.

Figure 26: **Fields in the etree header.**