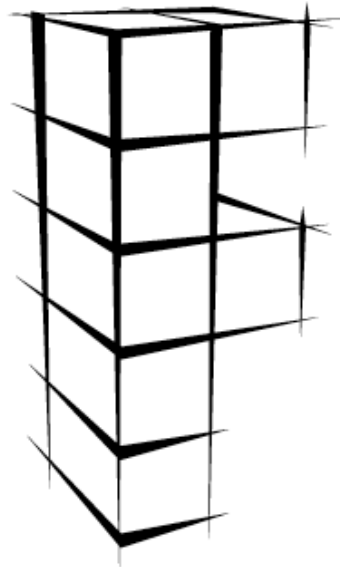


# FIELD3D

AN OPEN SOURCE FILE FORMAT FOR VOXEL DATA



Magnus Wrenninge  
Sony Pictures Imageworks

[magnus.wrenninge@gmail.com](mailto:magnus.wrenninge@gmail.com)  
<http://field3d.googlecode.com/>

# Table of contents

Revision history	3
Introduction	4
Concepts and conventions	6
Extents and data window	6
Mappings and coordinate spaces	6
Integer versus floating-point coordinates	8
System components	9
The Field class hierarchy	9
Metadata	11
Virtual and non-virtual access to voxels	11
Iterators	12
Interpolation	13
Concrete Field classes	15
DenseField	15
MACField	15
SparseField	16
Field3DFile	18
The use of HDF5	18
Structure of a Field3D file	19
Examples of usage	21
Manipulating voxels	21
Manipulating mappings	21
Mappings and interpolation	22
Creating a field, filling it with data and writing it to disk	22
Writing multiple fields to one file	23
Reading fields from disk	24
Programming guide	25
Coding standards	25
Typedefs	26
Use of namespaces	26
When does the namespace change?	27
Use of RTTI	27
Use of exceptions	28
Extending the system	29
Extending file I/O for new classes	29
Future work	29
Bridge classes	29
Reading of partial fields	29
Access to partial fields	29
Bounds-free fields	30
Frequently asked questions	31
Credits	32

## Revision history

**Nov 9, 2010**

Fixed the integer coordinates in the code example on page six.

**Sep 3, 2009**

Added descriptions of `DenseField`, `MACField` and `SparseField`.

**Aug 3, 2009**

First draft. Still missing documentation of concrete field classes and system extension.

# Introduction

FIELD3D is an open source library for storing voxel data. It provides C++ classes that handle storage in memory and a file format based on HDF5 that allows the C++ objects to be written to and read from disk.

The library was initially developed at Sony Pictures Imageworks as a replacement for the three different in-house file formats already used to store voxel data. It is the foundation for Imageworks' in-house simulation framework and volume rendering software and is actively being used in production.

FIELD3D comes with most of the nuts and bolts needed to write something like a fluid simulator or volume renderer, but it doesn't force the programmer to use those. While the supplied C++ classes map directly to the underlying file format, it is still possible to extend the class hierarchy with new classes if needed. In the future, it will also be possible to read and write arbitrary 3d data structures, unrelated to FIELD3D, to the same .F3D files as the built-in types.

The library has a number of features that make it a generally usable format for storing voxel-based data:

## Multiple types of data structures

The library ships with data structures for dense, sparse (allocate-on-write blocked arrays) and MAC fields, and each is stored natively in the .F3D file format.

## Arbitrary number of fields per file

A FIELD3D file can contain any number of fields. The file I/O interface makes it easy to extract all fields from a file, or individual ones by referencing their name or the attribute they represent. HDF5 ensures that only the bytes needed for a particular layer are read, allowing quick access to small fields even if they are part of larger files.

## Support for multiple bit depths

The included C++ data STRUCTURES are templated, allowing data to be stored using 16/32/64 bits as needed. Bit depths may be mixed arbitrarily when writing fields to disk.

## Arbitrary mappings

FIELD3D ships with a single mapping/transform type – a 4x4 matrix transformation. Other, arbitrary transformations can be supported by extending the Mapping class hierarchy.

## Storage of additional data

It's often necessary to store more information about a field than just the voxel data. To address this, each field can store an arbitrary number of key-value pairs as metadata.

**Heterogenous storage**

A FIELD3D file may contain a mix of all of the above features – there is no requirement on using a single data structure, bit depth, resolution or mapping for all fields in a file.

**Extendable via plugins**

The field types, mappings and their respective I/O routines can all be extended either by adding more data structures directly to the library, or by writing plugins.

**Proven back-end technology**

A FIELD3D file is really a convention for storing voxel data using the HDF5 file format. HDF5 handles the reading and writing of actual bytes to disk. It is used extensively by organisations such as NASA for storing simulation data and other gigantic data sets. More information can be found at <http://www.hdfgroup.org/HDF5/>.

**Data compression**

FIELD3D can use any compression algorithm used by HDF5. Currently, FIELD3D compresses all data using gzip.

**Open format**

Though FIELD3D (.F3D) files are easiest to read using the supplied I/O classes, it is still a standard HDF5 file, and can be read using those libraries directly if needed.

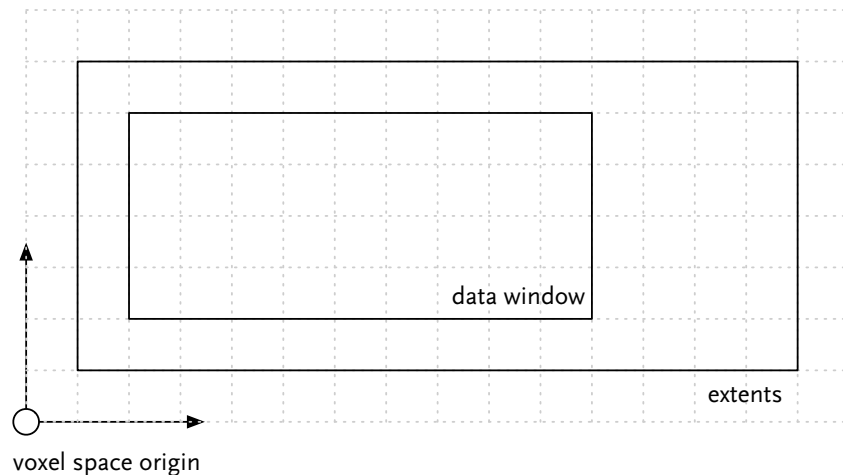
# Concepts and conventions

## Extents and data window

FIELD3D is similar to the OPENEXR format in that it allows a field to define the span of voxels it covers, as well as the span where its data is allocated. In OPENEXR, these are called the display window and data window, respectively. The display window in FIELD3D is called extents, but the data window concept and name is the same. Each is represented by an inclusive bounding box in integer voxel coordinates.

In the illustration below, the extents (which defines the  $[0,1]$  local coordinate space) is greater than the data window. It would be the result of the following code:

```
Box3i extents(V3i(1, 1, 0), V3i(14, 6, 10));  
Box3i dataWindow(V3i(2, 2, 0), V3i(10, 5, 10));  
field->setSize(extents, dataWindow);
```



Using separate extents and data windows can be helpful for image processing (a blur filter could run on a padded version of the field so that no boundary conditions need to be handled), interpolation (guarantees that a voxel has neighbors to interpolate to, even at the edge of the field) or for optimizing memory use (only allocate memory for the voxels needed).

## Mappings and coordinate spaces

Every `Field` object in FIELD3D has a `Mapping` object, which defines the world-to-local transformation. The mapping is owned by the `FieldRes` class which means it is accessible even if the template type of a field is unknown. (See System Components section below.)

Mappings only place the field in world space, and preserves that placement regardless of the resolution of the underlying field. This helps simplify the writing of resolution-independent code, such as defining fields of different resolution that occupy the same space.

There are three main coordinate systems used in `FIELD3D`:

- World space
- Local space
- Voxel space

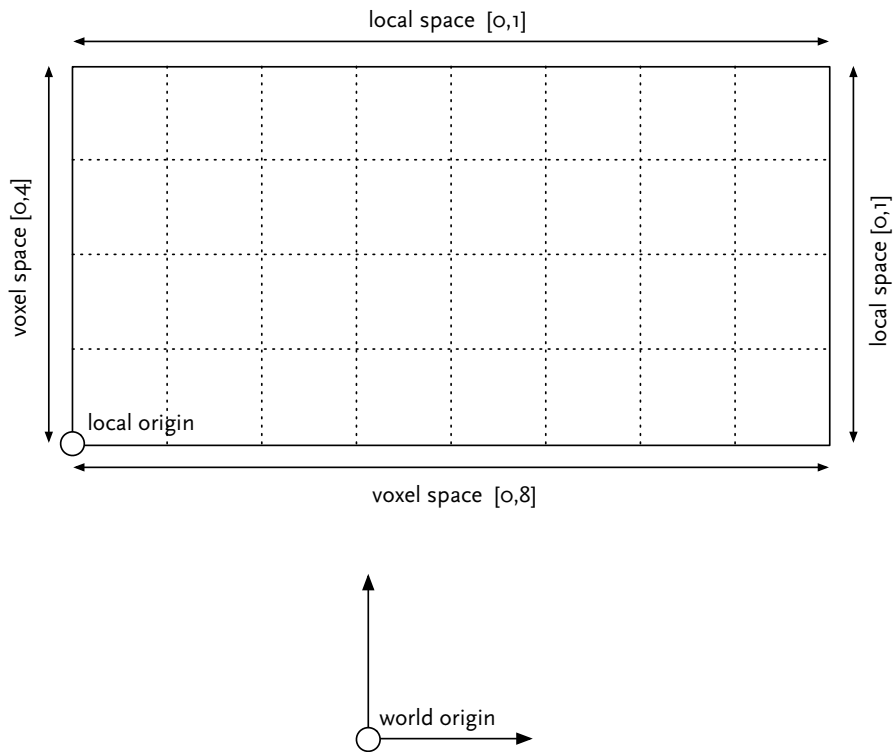
*World space* is the global coordinate system and exists outside of any `FIELD3D` field.

*Local space* is a resolution-independent coordinate system that maps the full extents of the field to a  $[0,1]$  space.

*Voxel space* is used for indexing into the underlying voxels of a field. A field with 100 voxels along the x axis maps  $[0.0,100.0]$  in voxel space to  $[0.0,1.0]$  in local space.

The reason for keeping two object-space coordinate systems (local and voxel) is that the local space is resolution independent, and makes it easier to deal with overlapping fields that cover the same space but are of different resolution. The mapping knows its field's resolution in order to provide local-to-voxel space transformations, but its transformation into world space (i.e. the field's position in space) stays the same after changing the resolution of a field. For this reason, the `Mapping` base class implements a `localToVoxel` transform directly.

The diagram below illustrates the coordinate systems used (in 2D for purposes of clarity).



## Integer versus floating-point coordinates

Voxel space is accessed in two ways – using integer or floating-point coordinates. Integer access is used for direct access to an individual voxel, and floating-point coordinates are used when interpolating values.

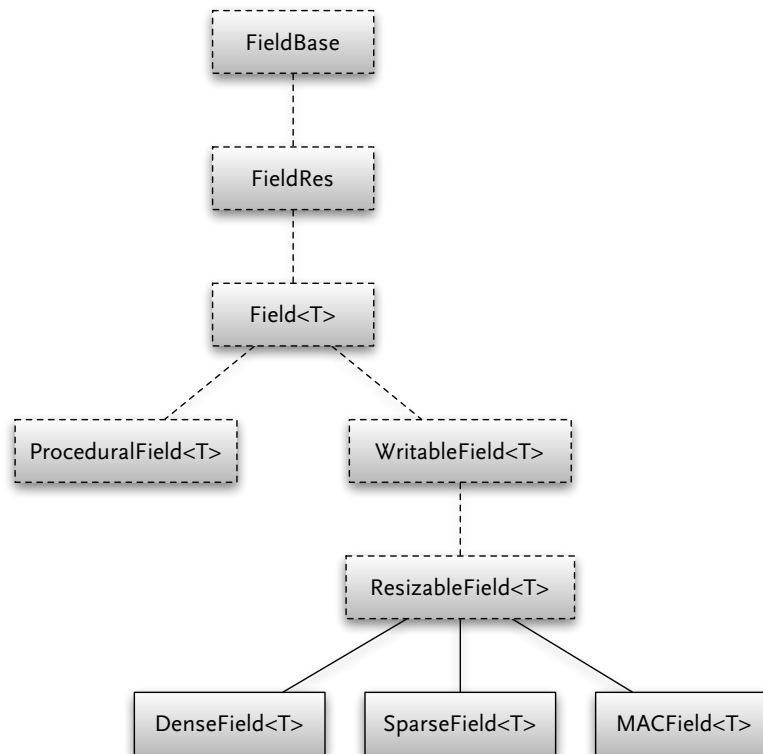
It is important to remember that care needs to be taken when converting between the two. The center of voxel (0,0,0) has floating-point coordinates (0.5,0.5,0.5). Thus, the edges of a field with resolution 100 are at 0.0 and 100.0 when using floating-point coordinates but when indexing using integers, only 0 through 99 are valid indices.

An excellent overview of this is found in *GRAPHICS GEMS I – What Are The Coordinates Of A Pixel?* by Paul S. Heckbert.

# System components

## The Field class hierarchy

Field objects belong to a class hierarchy which is broken down into the different major tasks performed. They help simplify generic programming, and given that the concrete classes are templated, the base classes separate as much of the generic information about the field away from the parts that depend on knowing the Field's template type.



### FieldBase

This class provides run-time class information to the library, metadata and other generic attributes of a field.

- Field name and attribute  
Each `FieldBase` object has public string data member for setting its name (indicating its association or perhaps purpose) and for its attribute name (indicating if it's used to store velocity, levelset, density, etc). These can be changed at will without touching the data, mapping or metadata of the field.
- ClassFactory registration  
`className()` is used to register new field types with the `ClassFactory`. Non-static

instantiation of fields is not used very often, but in the reading of `ProceduralField` objects from disk it comes into play.

- Metadata

Each field can store an arbitrary number of key-value pairs in the metadata section. Normally, this is passive information about a field, but subclasses of `ProceduralField` use it as a way of setting its parameters. `metadataChanged()` can be implemented by subclasses that need notification of changes.

- Reference counting

`FIELD3D` uses `boost::intrusive_ptr` for reference counting. Each `Field` class normally defines a `Ptr` typedef that refers to `boost::intrusive_ptr<FieldType>`. This helps keep syntax brief. `boost::mutex` is used for thread safety when incrementing/decrementing the reference count.

- RTTI

`FIELD3D` provides `field_dynamic_cast`, a version of `dynamic_cast` that is safe when objects cross a shared library boundary. See the “Use of RTTI” section in the programming guide below.

## **FieldRes**

Information about the field's dimensions and mapping in space is handled by `FieldRes`. Refer to the sections above on “Extents and data window” and “Mappings and coordinate spaces” for more detail. It should be noted that although this class holds the resolution of the field, it does not allow resizing. That is handled by `ResizableField`.

## **Field<Data\_T>**

Templating on the data type is introduced by `Field<Data_T>`. The class is responsible for all const access to the field. The reason for keeping non-const and const access in separate base classes is that certain types of fields (most notably `ProceduralFields`) don't have non-const access to data.

`Field<Data_T>` also provides the `const_iterator` class which is used to loop over the voxels in the field's data window. These STL-like objects are compatible with some (but not all) standard library algorithms. For example, `std::fill` works well but `std::sort` obviously doesn't. Also see the “Iterators” section below.

Creation of iterators is done with `cbegin()` and `cend()`, which also exist as `cbegin(subset)`. The bounding box version is used for iterating over only a subset of the field.

## **WritableField<Data\_T>**

This class adds non-const access to data in the same fashion as `Field<Data_T>` does for const access. It provides virtual `lvalue()` an iterator class, `begin()`, and `end()`.

## **ResizableField<Data\_T>**

The resizing of fields is handled by `ResizableField<Data_T>`. It provides a few different versions of `setSize()`, and each time the field is resized it also updates the `Mapping` object to reflect the new resolution.

## **ProceduralField<Data\_T>**

TO BE WRITTEN

## **Metadata**

Each layer in a `FIELD3D` file can contain its own set of metadata. Metadata is represented as a collection of key-value pairs and are stored in the `FieldBase` class. There is a fixed set of types supported:

- `std::string`
- `int`
- `float`
- `Vec3<int>`
- `Vec3<float>`

On disk, each metadata item becomes its own `Attribute` in the `HDF5` file.

## **Virtual and non-virtual access to voxels**

The virtual `Field<Data_T>::value()` call defines the generic calling convention for looking up the voxel value of a field. It does not provide optimal performance, but it does allow access to the data in any field, regardless of how that data is stored. Also, it should be noted that the return type of the virtual call isn't a reference – procedural fields that don't keep data in memory couldn't return one.

To allow non-virtual access to voxels, each concrete subclass by convention implements `ConcreteFieldClass::fastValue()`, which directly accesses voxels. It is up to each class to choose whether `fastValue` should return an object or a reference.

`Field<Data_T>::const_iterator` accesses voxels using the virtual `value()` call and thus isn't optimal in terms of speed. Just as with the `fastValue()` call, concrete classes may also implement their own version of `const_iterator` which either uses `fastValue()` or keeps a raw pointer to its data.

## Iterators

FIELD3D provides STL-like iterators for looping over the voxels in a field. These are more efficient than nested loops over `i, j, k` since they don't require calculating what memory location each new voxel index points to. For `DenseField`, a loop using `iterator` is very close to the speed of iterating over a `std::vector` or even using pointer arithmetic on a `float*`. Access to the current voxel index is still available through the iterator's `.x`, `.y`, `.z` member variables, if needed.

Some classes provide nonstandard iterator types. For example, `SparseField` has `block_iterator` which return a `Box3i` for each block in the field. This bounding box can then be used as a subset when creating iterators from the field itself.

`MACField` has a `mac_comp_iterator` which simplifies iteration over each MAC component for a given region of voxels.

A simple iterator loop example:

```
DenseFieldf::Ptr field(new DenseFieldf);
field->setSize(V3i(10, 20, 30));

for (DenseFieldf::iterator i = field.begin(); i != field.end(); ++i) {
    *i = static_cast<float>(i.x + i.y + i.z);
}
```

## Interpolation

FIELD3D comes with two types of interpolation – linear and monotonic cubic. Interpolation is always performed in voxel space, so in order to sample a field in world space, we need to use the field's *Mapping* to transform the point from world to voxel space.

A simple interpolation example:

```
DenseField<T>::Ptr field = someField;
DenseField<T>::LinearInterp interp;

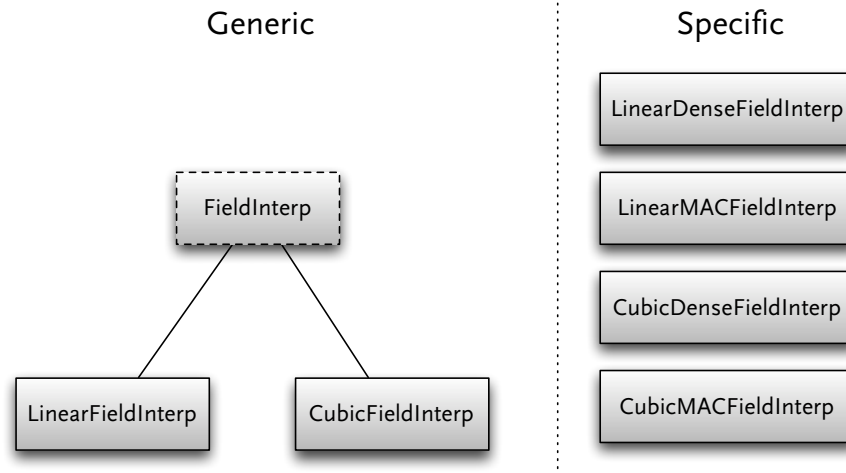
// Prefix indicates coordinate space
V3d wsP(0.0, 0.0, 0.0);
V3d vsP;

// Use mapping to transform between coordinate spaces
field->mapping()->worldToVoxel(wsP, vsP);

// Interpolation always uses voxel space
T value = interp.sample(field, vsP);
```

## Generic and specific interpolator classes

Just as with iterators, interpolators come in two flavors – generic ones, that access data using the virtual interface (`Field::value()`) and specific ones, that have knowledge of the data structure they traverse.



The generic interpolators all inherit from the `FieldInterp` base class and have a virtual interface for performing interpolation.

The specific classes share no base class – they are only usable when the concrete type is known.

To aid in the writing of generic code, each concrete class (for example `DenseField`) provides a set of convenience typedefs which bring in the appropriate interpolator class into the scope of the class. For example, `DenseField<float>::LinearInterp` is a typedef for `LinearDenseFieldInterp<float>`.

Some classes do not have specific interpolator classes – `SparseField` at the moment uses the generic interpolators. In these cases the typedef simply refers to those classes, i.e. `SparseField<float>::LinearInterp` refers to `LinearFieldInterp<float>`.

# Concrete Field classes

## DenseField

The dense field data structure is the simplest of the data structures included with `FIELD3D`. It assumes that voxel data is cell centered and allocates space for every voxel as soon as its size is set. Internally, it stores voxels in a linear array (a `std::vector<Data_T>`) that is continuous along the x dimension. `DenseField` has the least overhead when indexing directly to voxels and its iterators are the fastest of the included data structures. It is also the most direct implementation of the `Field` hierarchy interfaces, and doesn't add any significant member functions in addition to the common interface.

## MACField

A cell-centered field is the most straight-forward way to represent voxel data – each voxel stores a single value. This type of structure works well for volume rendering and representation of most scalar-valued voxel fields. When it comes to storage of vector fields for fluid simulation however, the cell-centered approach has serious shortcomings when it comes to differentiation. In 1965, Harlow and Welch introduced MAC grids, which represents fluid flow across the faces of a voxel instead of through its center. This allowed for much more accurate calculations of central differences, and is the most commonly used data structure for grid-based fluid simulation since. Robert Bridson's book *Fluid Simulation for Computer Graphics* (AK Peters 2008) is an excellent reference for more (and better) details on the computational aspects of MAC fields.

MAC fields usually reference the values on either face of a voxel as +/- half-indices, i.e. the component of the negative x-facing cell of the velocity field  $v$  at index  $i,j,k$  is  $v_{i-1/2,j,k}$ . `FIELD3D`'s `MACField` uses the notation  $s_{i,j,k}$  to refer to  $s_{i-1/2,j,k}$  and  $s_{i+1,j,k}$  to refer to  $s_{i+1/2,j,k}$ . This is consistent with most implementations of MAC fields.

In order to conform to the `Field` baseclass' definition of `value()` returning a cell-centered sample, `MACField`'s implementation of the virtual call automatically interpolates the voxel-face values to the cell center. To access the vector components directly, there is a set of separate member functions named  $u(i,j,k)$ ,  $v(i,j,k)$  and  $w(i,j,k)$  which all take indices as described above.

`MACField` also provides an iterator class (`mac_comp_iterator`) for quickly traversing all of the  $u$ ,  $v$  and  $w$  components sequentially for a given set of voxels. These iterators encapsulate the logic for traversing voxel faces, which differ in dimensions depending on the component selected.

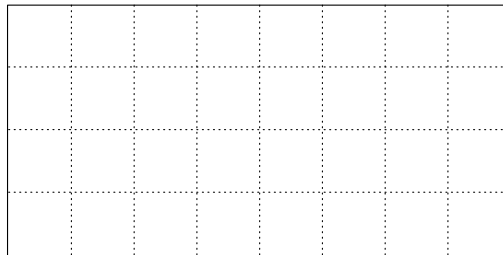
For more information on MAC fields, see the original paper: HARLOW, F. H., & WELCH, J. E. 1965. *Numerical calculation of time-dependent viscous incompressible flow of fluid with a free surface*.

## SparseField

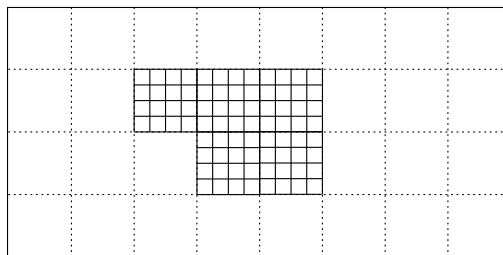
Sparse fields are different from dense fields primarily in that they don't allocate space for voxels until they are actually in use. This can be done in many different ways, and in the `SparseField` case it does so by ALLOCATING a coarse grid of "blocks" when the field is resized. Once a voxel is accessed for writing (using the `lvalue()` or `fastLValue()` calls) the block containing the voxel is allocated, making room in memory to store the voxel's value.

Before a block is allocated it stores an "empty value", which is constant across the entire block. This is used in cases where we don't want to imply that the value of unallocated areas is zero. For storage of levelsets this is important – information about whether a block is inside or outside the interface is always required. When calling `clear()`, the value used is set on each block as the "empty value", which means that the `SparseField` remains perfectly sparse even after initializing its values.

The following diagram illustrates how blocks get allocated to provide storage for individual voxels:



*SparseField blocks prior to individual voxels being allocated*



*SparseField after voxel access causes five blocks to be allocated*

`FIELD3D`'s `SparseField` class has certain voxel access overhead – it requires calculation both of block index and voxel index within a certain block. Also, the data is more fragmented than a `DenseField`'s, which further reduces speed. This overhead varies from architecture to architecture, and there is a test application called `access_speed` in the `apps` directory which tests the relative access speed between various field structures.

All blocks in the field have the same size, and at the moment the size is required to be a power-of-two. This requirement may change to be less conservative in a future version of `FIELD3D`, as it's easier to tune the block size for optimal caching if the size can vary arbitrarily.

`SparseField` provides various member functions for querying the state of individual blocks. For example, we may ask whether any given block is allocated or not. This is especially useful in conjunction with one of `SparseField`'s custom iterator types – `block_iterator`. This iterator traverses each block in order, and can return the integer bounding box (in discrete voxel space) for each block. This bounding box can then be passed to the `begin(subset)` iterator creation functions for efficiently traversing the voxels in a block sequentially.

In certain cases we want to release a block after it has been allocated. If, for example, we are converting a full levelset to a narrow band version, the blocks away from the interface can be deallocated. This is handled by the `releaseBlocks<Functor_T>()` call. It takes a functor which is given access to each block in order, and returns a boolean deciding whether the block should be deallocated. There are two examples included in `SparseField.h` – `CheckAllEqual` (which will compress blocks where all the values are the same) and `CheckMaxAbs` (which is used in the narrowband levelset case). These both serve as good examples when writing new functor types.

# FIELD3D FILE

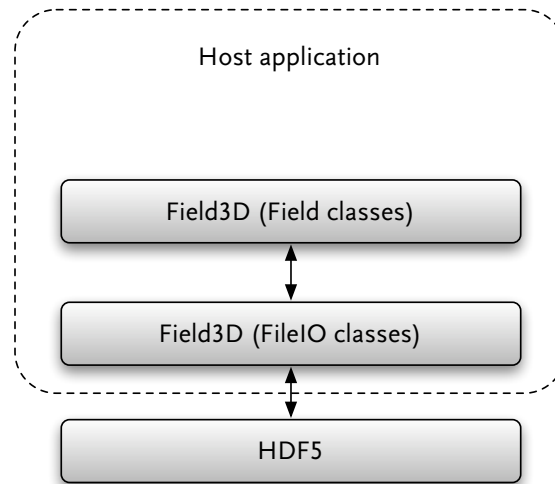
Subclasses of `Field` can be written to and read from disk using the `Field3DInputFile` and `Field3DOutputFile` classes. Any number of fields, along with their mappings and metadata may be stored in one file. There is no need for the data structure, template type, mapping or field resolution to be identical.

## The use of HDF5

The Hierarchical Data File 5 (<http://www.hdfgroup.org/HDF5/>) format is an open-source, generic file format for storing multiple structured, potentially huge, data sets in a single file.

On disk, `FIELD3D` is a fully compliant `HDF5` file, using the `HDF5` library for all data I/O. What `FIELD3D` adds is a well-defined structure for the data, metadata and coordinate systems (mappings) that are needed for the storage of volumetric and/or simulation data, with (initially, at least) a focus on the needs of visual effects production facilities.

By using `HDF5` for actual file I/O and merely describing the structure that volumetric data should have, `FIELD3D` leaves data open, but also makes it easy to extend the library as needed. If a new `Mapping` type is needed, this and its file I/O code can be written independently of the field data structures, and if a new field type is created, only the convention and code for writing the voxels themselves need to be implemented; routines for reading and writing mappings and metadata still stay the same.

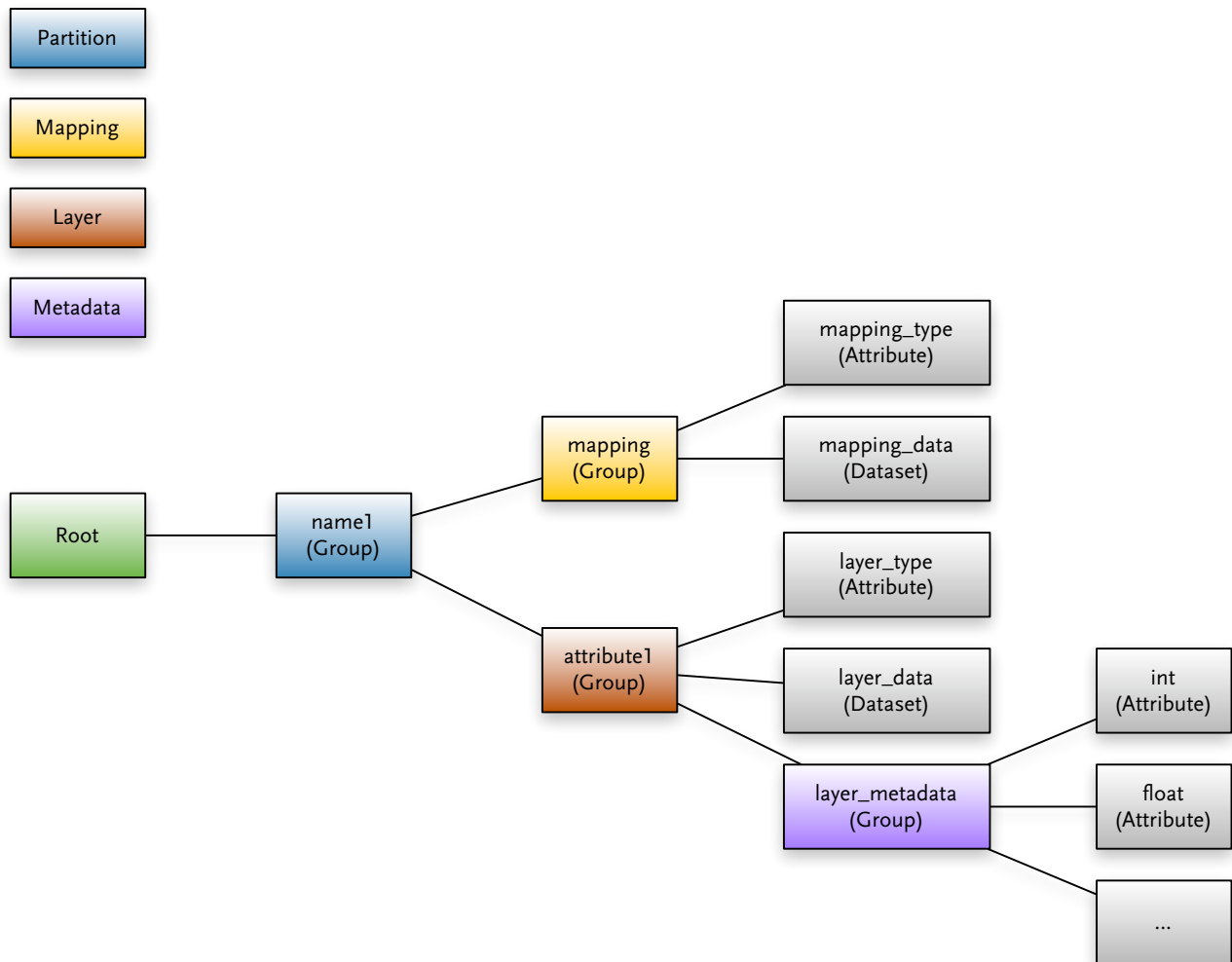


## Structure of a FIELD3D file

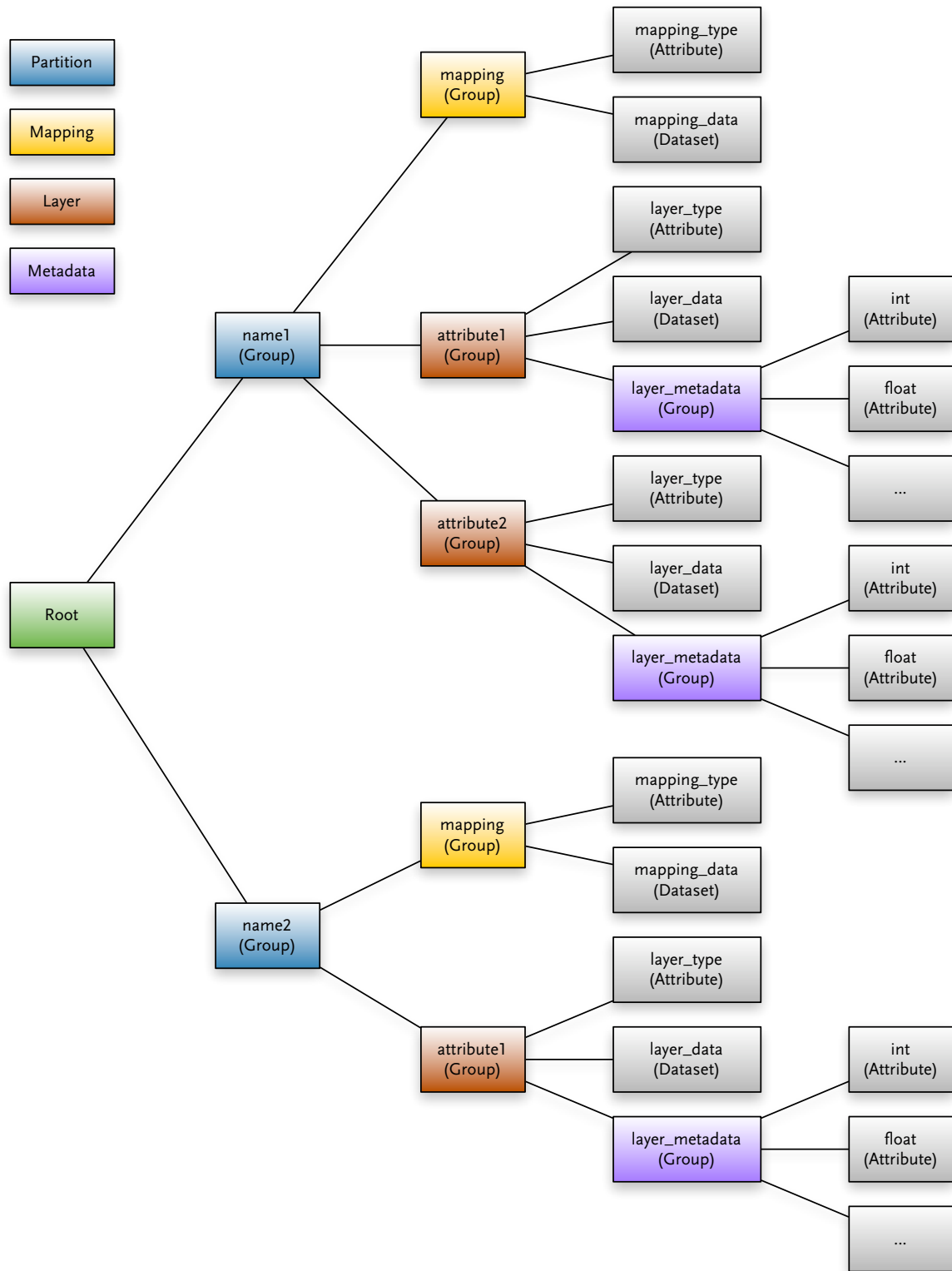
HDF5 defines a hierarchical structure of data in each file. Each unique field and mapping lives in a partition (which is named identical to the field's name), and each layer (which is named according to its attribute) lives under a partition. Layers share the same partition if they share the same field name and the same mapping.

Externally, the partition has the same name as the field(s) beneath it, but internally there is a counter appended, in order to keep all partition names unique. Thus, the fields `character:levelset` and `character:v` are internally referred to as `character.0:levelset`, `character.1:v` if they have different mappings, and `character.0:levelset`, `character.0:v` if they have the same mapping.

Somewhat simplified (excluding some HDF5 attributes) the structure is as follows, with the HDF5 type in parentheses.



When storing multiple fields, the graph grows as expected (see below). We would refer to the layers in this file as *name1:attribute1*, *name1:attribute2* and *name2:attribute1*.



## Examples of usage

For more complete, compiling examples, see the various samples in the *apps* directory in the project source root.

### Manipulating voxels

```
DenseField<float> field;
field.setSize(V3i(10, 5, 5));

// Write to voxel
field.lvalue(0, 0, 0) = 1.0f;

// Read from voxel
float value = field.value(0, 0, 0);
```

### Manipulating mappings

```
MACFieldf::Ptr sim;
sim->setSize(V3i(100, 200, 100));

MatrixFieldMapping::Ptr mapping(new MatrixFieldMapping);
M44d localToWorldMtx;

localToWorldMtx.setScale(V3d(10.0, 20.0, 10.0));
mapping->setLocalToWorld(localToWorldMtx);
```

## Mappings and interpolation

```
float myInterpolation(const DenseFieldf &field, const Vec3d &wsP)
{
    // Create interpolation object
    LinearDenseFieldInterp<float> interp;

    // Transform from world space to voxel space
    V3d vsP;
    field.mapping()->worldToVoxel(wsP, vsP);

    // Return interpolated result
    return interp.sample(field, vsP);
}
```

## Creating a field, filling it with data and writing it to disk

```
DenseField<float>::Ptr field(new DenseField<float>);
field->setSize(V3i(50, 50, 50));
field->clear(1.0f);

Field3DOutputFile out;
out.create("field3d_file.f3d");
out.writeScalarLayer<float>(field);
```

## Writing multiple fields to one file

```
// Since by default, the mapping is NullFieldMapping, both fields have
// the same mapping and giving them the same name will place both under
// the same "partition" in the f3d file.

// Create a scalar field to store a levelset
DenseField<float>::Ptr scalarField(new DenseField<float>);
scalarField->setSize(V3i(50, 50, 50));
scalarField->clear(1.0f);
scalarField->name = "character_head";
scalarField->attribute = "levelset";

// Create a vector field to store a velocity attribute
DenseField<V3f>::Ptr vectorField(new DenseField<V3f>);
vectorField->setSize(V3i(50, 50, 50));
vectorField->clear(V3f(0.0f, 1.0f, 0.0f));
vectorField->name = "character_head";
vectorField->attribute = "v";

// Note that the template type for writeVectorLayer<> is the template type
// of the vector, not the vector type itself.
Field3DOutputFile out;
out.create("field3d_file.f3d");
out.writeScalarLayer<float>(scalarField);
out.writeVectorLayer<float>(vectorField);
```

## Reading fields from disk

```
initIO();

Field3DInputFile in;
Field<float>::Vec scalarFields;

in.open("file.f3d");
scalarFields = in.readScalarLayers<float>();

Field<float>::Vec::const_iterator i = scalarFields.begin();
for (; i != scalarFields.end(); ++i) {
    cout << "Name:      " << (**i).name << endl
         << "Attribute: " << (**i).attribute << endl;
}
```

# Programming guide

## Coding standards

The following is by no means an entirely exhaustive breakdown of the coding standards used in FIELD3D, but should at least indicate the preferred format for the most common uses.

1. If nothing is mentioned in these standards regarding a particular style option, find existing cases in the code and match those.
2. Keep line lengths less than or equal to 80 characters.
3. The order of include files (both in .h and .cpp files) should be: System headers; External library headers; Project headers.
4. Always put spaces after commas
  - 4.1. `float value = someFunction(arg, arg2);`
  - 4.2. `not float value = someFunction(arg, arg2);`
5. Never put spaces directly inside parentheses
  - 5.1. `someFunction(arg1);`
  - 5.2. `not someFunction( arg1 );`
  - 5.3. `float value = (a + 1) * b + 1;`
  - 5.4. `not float value = ( a + 1 ) * b + 1;`
6. Always put spaces around binary operators, never around unary
  - 6.1. `float value = a * b + 1;`
  - 6.2. `not float value = a*b+1;`
  - 6.3. `float value = -a;`
  - 6.4. `not float value = - a;`
  - 6.5. `float value = a;`
  - 6.6. `not float value=a`
7. Never make exceptions to the above rules for spaces, even to shorten a line to 80 characters.
8. Always fully scope symbol names in header files. Never put `using` statements in header files, with the exception of locations where they have localized effect, e.g. inside function implementations.
9. Class data members should be prefixed `m_`, or in the case of static data members `ms_`.
10. Braces, `{}`, are placed on a separate line for classes and functions, and on the same line in conjunction with `for`, `while` and `if` statements.
11. The public section of a class always appears before the private section.
12. One-line functions may be implemented inline in a class header. All other functions should be implemented in the .cpp file, or at the bottom of the .h file (for templated methods and inline methods).
13. In the class definition, maintain the member functions in groups depending on their context, i.e. accessors, virtual functions to be implemented by subclasses, virtual functions previously defined in base classes, etc. The groups are delimited by a comment with dashes

extending to the full 80 character width. If desired, a doxygen group may also be created to improve the reading of generated documentation.

14. Separate new classes with a full-width comment line followed by the name of the class and a second full-width comment line.
15. Separate function implementations by a single full-width comment line.
16. Template arguments have a `_T` suffix.
17. Use `/// style (doxygen) comments to document anything in header files. Don't duplicate these comments in the source file.`
18. Feel free to use any other doxygen constructs in code documentation, such as `\note`, `\warning`, `\return`, `\todo`, `\param`.
19. If a vector, point, etc. is assumed to be in a given coordinate space, prefix the variable with a shorthand version of that coordinate space. For example, a point `P` in world space should be called `wsP`, and after transforming to voxel space it should be called `vsP`.

## Typedefs

Standard typedefs exist in the following cases:

- Concrete field template instances
  - `DenseField<float> == DenseFieldf`
  - `SparseField<V3f> == SparseField3f`
- Interpolators
  - `DenseFieldf::LinearInterp == LinearDenseFieldInterp<float>`
- `intrusive_ptr`
  - `boost::intrusive_ptr<DenseFieldf> == DenseFieldf::Ptr`

## Use of namespaces

The lack of namespaces often causes complications in applications that allow plugins to be written. For example, if the host application dynamically links to a library of version X, and a plugin is compiled against version Y, the symbols will be identical and the plugin will probably call version X, which may or may not be compatible with the headers used at compilation.

In order to prevent this, `FIELD3D` not only has its own namespace but also has a versioned namespace internally. Thus, in the first release, `FIELD3D` lives in `Field3D::v1_0`. To keep code that uses the library from having to explicitly scope symbols with `Field3D::v1_0`, a `using` statement is included everywhere the `FIELD3D` namespace is opened. While this breaks the rule (though only here do we break that rule) of never putting a `using` statement in a header file, it allows code to reference a class as `Field3D::DenseField<float>`, but to the compiler the mangled symbol will include its full scope, i.e. `Field3D::v1_0::DenseField<float>`, thus preventing any symbol clashes if multiple versions of the library are used in the same application.

While this seldom becomes an issue in simple, standalone applications, in a production environment with tens or hundreds of versioned plugins loaded in a host application it often causes problems.

In order to cleanly handle the versioning, and keep the version number in a single place, a `#define` called `FIELD3D_NAMESPACE_OPEN` is used instead of explicitly stating the full namespace in each file. This `#define` lives in `ns.h`. When closing the namespace, one of the `FIELD3D_NAMESPACE_HEADER_CLOSE` and `FIELD3D_NAMESPACE_SOURCE_CLOSE` are used as appropriate, depending on if the file is a header or source file.

## **When does the namespace change?**

The library version is in the format `major.minor.micro`. Whenever binary compatibility is broken, or whenever functionality changes in a significant way, the minor (or the major and minor) version will change. This in effect changes the namespace name of the library. Thus, small fixes can be released as updates to the micro number, which keeps the namespace name intact and allows an already compiled application to pick up changes to the shared library.

Updates to the micro number must never change the functionality of the library. If the data structures, file format or any other part of `FIELD3D` changes its behavior, the minor version must be incremented.

## **Use of RTTI**

Gcc has issues when it comes to maintaining RTTI information once an object crosses a shared library boundary. `dynamic_cast<>` works well if both the call and the allocated object reside in the same dynamic link unit, but if an object was allocated in a shared library and the `dynamic_cast<>` happens in the application itself, the result will be a zero pointer.

`FIELD3D` avoids this by providing its own run-time type info checks. `field_dynamic_cast<>` works only for the class hierarchy under `Field`, and performs a full string compare between type names to determine matching classes. It first checks the object itself, and if no match is found, it will check the base class recursively up the hierarchy.

If gcc's behavior changes in the future, the implementation of `field_dynamic_cast<>` could change internally to directly call `dynamic_cast<>`, but wouldn't break existing code.

## **Use of exceptions**

FIELD3D uses exceptions internally, but catches everything before it has a chance to cross the shared library boundary. When writing plugins and/or extending the library, it is ok to use exceptions. However, any functions called from outside the library should refrain from throwing exceptions.

# Extending the system

## Extending file I/O for new classes

Each of the provided `Field` subclasses can be written to disk. This is handled by the `FieldIO` subclasses, which means it's possible to extend the system to handle input and output of any new `Field` subclasses as well.

TO BE WRITTEN

## Future work

### Bridge classes

In the first release, the file I/O routines require the inputs and outputs to be classes from the supplied class hierarchy. Bridge classes would allow users of the library to write small translation functions that could pass voxel data, mapping definitions and metadata to arbitrary data structures.

### Reading of partial fields

Although the `SparseField` class allows its blocks to be read as-needed when a voxel is accessed, `FIELD3D` generally reads an entire field at once. In the future, support will be added for reading a subset of a field. `HDF5` has a concept called "hyperslices" which will allow this to be implemented efficiently.

### Access to partial fields

`Field3d` currently does not provide a way to read only a part of a field on disk. If an application knows it only needs data within a given subdomain, it should be possible to only read those voxels.

## **Bounds-free fields**

Regular sparse fields allow us to allocate voxel data only in certain regions. The fields do however still have a fixed domain outside which data cannot be stored. Also, we pay a small but not irrelevant price even for the space that is unused, since information about the block still needs to be stored. Bounds-free (or boundless) fields would take the sparseness one level further by using a spatial hash to store the block information. Using this technique one only pays the price of actual blocks used, regardless of the size of the domain.

Both cell-centered and MAC boundless fields will be required in order to support unbounded simulations.

## Frequently asked questions

This section answers some common questions about the FIELD3D library.

*FIELD3D mentions both generic (and presumably slow) voxel access as well as direct (and fast) access. Please explain.*

The `Field` base CLASS provides both virtual functions for access to voxel data (the `value()` function) as well as iterators for traversing all the voxels in a field. The iterators simply call the `value()` function, but makes it easy to loop over an entire field.

When coding directly against a known data structure, e.g. `DenseField<float>`, the `fastValue()` call is non-virtual (virtual functions can be unusually slow in threaded applications). Also, the `DenseField` template class provides `const` and `non-const` iterators specifically written for it, which are faster than the generic `Field::const_iterator` and `WritableField::iterator`.

Most likely an application would be written so that it deals with data in the direct, more efficient manner. However, it could also implement routines for handling unknown field types through the generic interface. This future-proofs the application, in that new data structures can be supported that were unavailable when the application first was written, simply by recompiling against the latest library version, or by extending the set of available subclasses through the use of plugins.

*Can each layer/field have its own resolution?*

Yes, there is no limitation on the resolution of a field in relationship to another field in the same file, even with the same field name and/or attribute name. Each field's mapping may also be different.

*Can the fields be offset in space or do they have to be coincident in space?*

Just as the resolution (extents and data window) may be different, each field may have its own unique mapping, and each mapping may be of a different type. For example, a `MatrixFieldMapping` may be used on one field while another uses some custom mapping type, for example a frustum-aligned mapping.

## Credits

The FIELD3D library was originally developed at Sony Pictures Imageworks during 2008 and 2009. The following people were involved in the design and implementation:

- Magnus Wrenninge
- Chris Allen
- Sosh Mirsepassi
- Stephen Marshall
- Chris Burdorf
- Henrik Fält
- Scot Shinderman
- Doug Bloom

Special thanks go to Barbara Ford for management of the project and to Rob Bredow for spearheading the Open Source effort at Imageworks.