

# f3d — A File Format and Tools for Storage and Manipulation of Volumetric Data Sets

Miloš Šrámek  
milos.sramek@oeaw.ac.at

Leonid I. Dimitrov  
leonid.dimitrov@oeaw.ac.at

Commission for Scientific Visualization, Austrian Academy of Sciences  
Donau-City-Str. 1, A-1220 Vienna, Austria

## Abstract

*Different file formats for storage and manipulation of volumetric data exist today, but none of them has been accepted as a standard by the volume visualization and volume graphics communities until now. This paper tries to change this situation by proposing a new, freely available file format f3d, which, as we believe, fulfills the demands of research and educational environments. We hope that this new format will be accepted by the above mentioned communities and thus will provide a basis for improved communication between different groups and applications.*

## 1 Introduction

Today, the volume visualization and volume graphics communities are still missing a simple, powerful and widely accepted format for storage and exchange of volumetric data. Similarly, they are also missing a set of easy-to-use tools for basic data operations, similar in concept to the widely available 2D image viewers and tools. This paper is an attempt to change this situation by introducing the new file format f3d (Format 3 dimensional) which is, as we hope, both simple and versatile enough to address the demands of labs, groups or individual researchers on representation and manipulation of Cartesian, regular and rectilinear gridded data. Currently, the format supports properties, which we found to be important from the point of view of our experience. f3d is distributed as free software, which hopefully will encourage other contributors to add their own features, extensions and tools. The f3d implementation provides a set of simple and flexible C language routines for reading and writing files, which provides for a steep learning curve for novices and enables easy porting to existing applications. Additionally, a set of tools based on a high-level C++ class library is provided for format conver-

sions, geometric transforms, filtering and other operations with volumetric data.

The paper is organized as follows. In the second section we formulate our demands of a “good” volumetric data format. In Section 3 we describe the f3d format in detail, in Section 4 we introduce the C and C++ libraries for f3d file manipulation and in the last Section 5 we conclude and formulate our future goals.

## 2 3D format requirements

Here we postulate a list of demands, which we think a “good” file format should fulfill. In spite of the fact that it is hard to order them from the point of view of importance, we think that there are some preferences which weigh more than the others, and therefore we list them first.

**Flexibility:** A format, which is intended for use by the scientific community, must knit together two contradictory demands: on the one hand, being a *format*, it should define strict rules in order to achieve mutual compatibility of different applications; on the other hand, being *scientific*, it should provide the possibility for modification in order to be able to adapt itself to the unpredictably varying environments of its usage.

We implementat this feature by splitting the format in a strictly defined *data* part, which secures the inter-application compatibility and an almost completely free *application specific* part. The data part of the format defines the way how volumetric data are stored, together with the data specific parameters (f3d tags e.g. grid dimensions, type, voxel format, etc.). The data part must be readable by each application and therefore its syntax should be strictly defined. The aim of the application specific part is to store information which is relevant to the purpose of the data and usually describes higher level features. For example,

a threshold value which enables the identification of foreground voxels can be significant for a renderer, but it is not needed for a down sampling tool, which can ignore it.

It is advantageous that both parameters and comments be stored in a textual form, in order to provide for their inspection or even modification by standard tools (image viewers, editors).

**Simplicity:** A format in order to be universal and easy to use should not enforce any restrictions on the application itself or the style of programming. Scientists usually have their own ideas and styles and are not willing to sacrifice them in favor of something “less” important, like, for example, the data format. Therefore the basic API of the format implementation should be as simple as possible, preferably consisting of only a couple of functions. A simple API also simplifies porting to existing applications and thus would help the acceptance of the format by the community.

**Availability:** The format as well as its implementation should be available in source code and must be covered by a non-restrictive license. The open source movement has recently proved that such free access not only benefits the user, but also, by means of the feedback, profits the author by attracting users to the development (the bazaar style[1]). The source openness may invoke a feeling of an “ownership” within such a community, which further attracts users to testing and development.

**Preview:** A favorable feature would be a 2D preview image, giving the user a hint about the file contents. The preview, viewable by standard image viewers (or even a Web browser), should be an integral part of the format file. It enables rapid orientation in a larger set of volumes and simplifies their maintenance.

**Grids:** The data types in volume visualization tasks span a range from the simplest Cartesian grids, defined by a single number—the voxel size, to completely unstructured grids, where the location of each sample has to be registered individually. It is questionable, if a single file format should support all possible grid types.

The Cartesian, regular and rectilinear grids can be easily represented by a 3D matrix with interplane distances stored in three one-dimensional arrays. These arrays are further simplified to a single number (Cartesian) or a triple of numbers (regular grids). These grid types share similar properties (more or less direct access to voxels, grid traversal along a ray) and are the standard data type output by tomographic scanners. On the other hand, curvilinear, structured and unstructured grids are usually of interest in completely different areas, as, for example, computational fluid dynamics

and the manipulation of the data is different. Therefore we assume that a single format, in order to preserve the desired simplicity, should not mix these two categories of grids.

**Voxels:** Tomographic scanners usually produce data with 8 or 16 bit voxels. However, in research environments, we encounter also other voxel types: 32 bit integers, floats, doubles, complex numbers (Fourier images), vectors (gradient fields, 3D color texture maps).

Different computers store the bytes of multibyte variables in different order (big endians vs. little endians). If we assume that the data files are predominantly used on the same type of computer, in order to minimize the additional overhead, neither of both types should be preferred. A flag should be used instead, defining the order of bytes, which is then reversed only when reading the data set on a computer of different type.

**Data compression:** Volumetric data sets are usually huge. Often they compress well, especially noiseless synthetic data as, for example, segmented masks or voxelized objects. A lossless compression technique should be preferred for a general purpose data format.

### 3 The f3d file format

The hot spot of the new format design is the preview image and the demand for its viewability by standard image viewers. Therefore, it should be an extension of a well established 2D image format, which enables free-form textual comments. The most suitable candidate at hand is the PNM (portable anymap) format, in its PPM (portable pixmap) and PGM (portable graymap) variations. The first one is preferred for storage of 3D vector data (gradient arrays and color textures), while the second one is suitable for scalar volumes.

The structure of a PNM file is very simple. After a leading magic number (P5 for PGM and P6 for PPM) an arbitrary number of comment lines, starting with “#” can follow. These comments are succeeded by width, height and maximum color-component values, all stored as ASCII characters and separated by whitespaces. A single whitespace separates the header from  $width \times height$  bytes of image data.

The PNM format is extended to f3d by:

1. the definition of f3d tags and application specific comments in the form of PNM comments, and
2. the appendage of compressed slice data (slice records) behind the image data.

Both extensions are ignored by image viewers. Comments can be inspected by some viewers (e.g., xv), by the UNIX head command or even by some text editors (vi).

A slice record consists of two fields. The first one is a single 32 bit integer, written in the MSB order, which defines the length of the immediately following compressed slice. This provision enables the direct access to arbitrary slices without decompression of the previous ones. We selected the standard `compress` and `uncompress` routines from the freely available `zlib`<sup>1</sup> package.

### 3.1 f3d tags and task specific comments

Figure 1 shows an example of the `f3d` header. All lines starting with “#” are PNM comments, and those starting with “#!” define `f3d` parameters of the form

#!tag value

where `value` can be either numeric or textual.

The following tags are allowed:

#! **f3d version** identifies the file as a volume in `f3d` format. `version` defines the software version in order to enable backward compatibility.

#! **endian** {`f3dBigEndian`|`f3dLittleEndian`} defines the byte order of the computer the file was written on.

#! **vdim** `nx ny nz` defines the volume dimensions. `nx` is the dimension along lines (fastest growing coordinate), `ny` counts the lines and `nz` defines the dimension across slices.

#! **vtype** `value` defines the grid type. Possible values:

`f3dCubic` Cartesian grid. The voxel dimensions are defined by a single `px` tag (see later).

`f3dRegular` Regular grid. The voxel dimensions are defined by the tags `px`, `py` and `pz`.

`f3dRectZ` Regular grid in directions of the `x` and `y` axes, variable inter-slice distance along the `z` axis. The dimensions in the `x` and `y` directions are defined by the tags `px` and `py`. The slice positions in `z` the direction are defined by the `nz` times repeated `rz` tag.

`f3dRectXYZ` Rectilinear grid. The positions are defined by repeated `px`, `py` and `pz` tags.

#! **ctype** `value` Defines scalar and vector (color) data. Possible values: `f3dMono`—scalar data, one value per voxel, `f3dComplex`—complex data, two values per voxel, `f3dColor`—vector (color) data, three values per voxel.

#! **dtype** `value` Defines the numerical type of a voxel: `f3dUChar`, `f3dChar`, `f3dUInt16`, `f3dInt16`, `f3dUInt32`, `f3dInt32`, `f3dFloat`.

#! **units** `value` Physical units of the grid dimensions: `f3dUum` —micrometers, `f3dUmm` —millimeters, `f3dUm` —meters.

#! **px size** defines the voxel size in the `x` direction for `f3dCubic`, `f3dRegular` and `f3dRectZ` grids.

#! **py size** defines the voxel size in the `y` direction for `f3dRegular` and `f3dRectZ` grids.

#! **pz size** defines the voxel size in the `z` direction for `f3dRegular` grids.

#! **rx size** defines the voxel size in the `x` direction of a rectilinear grid (`f3dRectXYZ`). Must be repeated `nx` times in increasing order.

#! **ry size** defines the voxel size in the `y` direction of a rectilinear grid (`f3dRectXYZ`). Must be repeated `ny` times in increasing order.

#! **rz size** defines the voxel size in the `z` direction of a rectilinear grid (`f3dRectZ` and `f3dRectXYZ`). Must be repeated `nz` times in increasing order.

#! **absMax** `value`

#! **absMin** `value` define the actual density minimum and maximum. These values are computed just before storing the grid in a file.

#! **defaultMax** `value`

#! **defaultMin** `value` define the default density limits, which are kept during certain operations (filtering, cutting). These values enable consistent density scaling when processing the data by external tools.

#! **comment** `text` An `f3d` comment. It can be used for the definition of application specific comments, which are used to pass parameters between applications. In that case, the preferred form is

#!comment identifier text

where `identifier` is a keyword understood by an application or a group of applications.

## 4 Implementation

The implementation of the `f3d` format is separated in two parts: a low-level API, offering basic C functions for reading, writing and other simple tasks, and a high level C++ class library. While the first was intentionally kept very simple, in order to allow for easy porting of the `f3d` format to existing applications, the second one offers extended functionality, which is useful for writing different volume manipulation tools.

At the root of the `f3d` implementation is the structure `f3dHeader` storing all volume parameters, which were listed in the previous section. This structure is therefore an argument for most of the routines, the most important of which are `f3dWriteGrid`, `f3dWriteHeader` and `f3dWriteSlice` for data writing and similarly `f3dReadGrid`, `f3dReadHeader` and `f3dReadSlice` for data reading.

Classes, defined and implemented in the C++ library offer, except for a similar functionality as the C library, data

<sup>1</sup>`zlib-1.1.3`, <http://www.cdrom.com/pub/infozip/zlib/>

P5	PGM format (magic number)
#!f3d 1.2	f3d parameter
#!endian f3dBigEndian	f3d parameter
#!vdim 128 128 128	f3d parameter
#!vtype f3dCubic	f3d parameter
#!px 1.000000	f3d parameter
#!ctype f3dMono	f3d parameter
#!dtype f3dUChar	f3d parameter
#!units f3dUmm	f3d parameter
#!absMin 0	f3d parameter
#!absMax 240	f3d parameter
#!defaultMin 0	f3d parameter
#!defaultMax 255	f3d parameter
#!comment File created by 'vxtGrid3D'	f3d comment
#!comment vxt profile linear 1.8	f3d comment/application parameter
128 128	PNM format (image dimensions – in this case the preview image)
255	PNM format

**Figure 1. Header of an f3d file.**

structures for representation and manipulation of Cartesian, regular and rectilinear grids.

The `vxtGrid3D` class is the ancestor of all other classes. It is a template class which implements 3D data matrices of different voxels. Its functionality encompasses reading, writing, and basic data access methods.

The existence of different voxel types complicates processing of volumetric data, since it is necessary to know the voxel type before data loading. A typeless processing of volumetric data, i.e., such, when the type of a voxel is not known in the program, is provided by the abstract classes `vxtScalarVolume` and `vxtVectorVolume`.

#### 4.1 The f3d package

The `f3d` package is free software and is distributed together with the documentation in both source code and as precompiled binaries<sup>2</sup>. It was tested on GNU/Linux, SGI IRIX and Windows98–2000 systems.

Except for the libraries, the distribution package contains also an ever growing list of tools for format conversions (`raw2f3d`, `pnm2f3d`, `f3d2raw`, `f3d2pnm`), geometric transforms (`f3dScale`, `f3dtrans`, `f3diso`, `f3dmerge`), point operations (`f3dmask`, `f3darith`, `f3dthresh`), filtering (`f3dmin`, `f3dmax`, `f3dmedian`, `f3dgauss`, `f3dgrad`, `f3dgradmax`, `f3dgabor`) and for other operations. Some of these tools were written as student projects at the Technical University Vienna, Austria and Comenius University Bratislava, Slovakia. `f3d`

<sup>2</sup><http://www.viskom.oeaw.ac.at/~milos/page/Download.html>

has been ported to other packages, developed in our lab, for example `VORTEX`<sup>3</sup> for data visualization and the semi-interactive 3D segmentation tool `ISEG`<sup>4</sup>

## 5. Conclusion

We proposed and implemented a new `f3d` format for storage of 3D Cartesian, regular and rectilinear data, supporting different kinds of voxel types, together with C and C++ libraries for reading, writing and manipulation of such data. Although we find this format useful and have ported it to our older applications, we cannot say yet (though we believe in it), if it is really useful for the visualization and volume graphics communities. The implementation follows the open source model and is thus open to anybody, who wants to contribute his/her ideas. There is definitely a lot to be done, at least by writing additional tools for data manipulation and conversion from/to other formats or adding the important but still missing `f3dComplexType` voxel type. Except for the activity in this area, we started to build a database of freely accessible data sets<sup>5</sup>, which provide an environment for algorithm testing and comparison.

## References

- [1] Eric S. Raymond. *The Cathedral & the Bazaar*. O'Reilly & Associates, Inc., 1999.

<sup>3</sup><http://www.viskom.oeaw.ac.at/~leon/VORTEX>

<sup>4</sup><http://www.viskom.oeaw.ac.at/~milos/page/Iseg/Iseg.html>

<sup>5</sup><http://www.viskom.oeaw.ac.at/~milos/f3dData/>