Getting Data into ImageVis3D

Contents

1 Introduction

One of the most common inquiries we get is, "What is the best way to get my data into ImageVis3D?" The answer is invariably specific to your particular case. There are a number of ways you can get ImageVis3D to render your data, with a varying range of associated complexities. This document attempts to answer some of the most common questions.

Briefly, there a couple ways ImageVis3D can load data out of the box: by loading up a UVF, ImageVis' native image format, or by converting data into UVF. However, ImageVis3D is meant to visualize massive datasets; theoretically up to exabytes of data. In practice, even for data as small as a terabyte, the conversion process is going to be too time consuming for reasonable day-to-day use.

For data at this scale, there are two options: write out your data as UVF in the first place, or implement a reader which sits parallel to UVF within ImageVis3D's IO subsystem. Both impose restrictions on the structure of your data, and will require someone with C++ programming experience to make it happen. However, the payoff is large: ImageVis3D will be able to open your formats 'natively', without the costly conversion step, and you will no longer have to deal with the same data replicated across multiple files.

Use our code!
ImageVis3D's design is component-based. In this document, we will use "the IO subsystem" and "ImageVis3D" interchangeably, but we are usually referring to the IO library which we have written specifically for use with ImageVis3D. Despite the name, the library is independent of ImageVis3D, and we explicitly encourage those who are capable to use it separately, in their own applications! More information is available by asking on the mailing lists.

Broadly, both schemes accept data on regular (though potentially anisotropic) grids. The converted formats are what we will refer to as *single-block*, meaning that the entire dataset lives in one large array, regardless of how large the dataset is. By comparison, ImageVis3D's variant of UVF is inherently *multi-block*, meaning that large data are "chunked", such that multiple "chunks" are required to recreate the entire volume. Single-block data is inherently simpler and has more support from existing tools. Multi-block data is more complex, but — even ignoring ImageVis3D for a moment and speaking from a purely algorithmic standpoint — there is simply no way to scale single-block data access to provide efficient visualization and analysis for large data.

2 Grid Types

Data can be defined on a wide variety of grid types. ImageVis3D can only handle what is called "regular" gridded data. Think of an image file: it is defined as a series of pixels, with an implicit ordering and distance between each pixel. That is, if we call a pixel `x` and define a pixel `x+1` to be the pixel to the right of `x`, then we know that the distance between `x` and `x+1` is equal to the distance between `x+1` and `x+2`. Regular data is an extension of this property to 3D: all data are layed out on an even grid.

Example Regular Gridded Data (. represents a grid location)

```
.   .   .   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .   .   .
.   .   .   .   .   .   .   .   .   .   .   .
```

However, do note that ImageVis3D supports so-called **anisotropic** data. That is, the difference in space between `x` and `x+1` need not be equal to the difference between `y` and `y+1`. More succinctly, the spacing information can vary per-dimension, but not within a dimension.

Unstructured Data (.: grid location, \-/x: edge)

```
.----.--.---.---.
 \   |   \ /    .
  \  .    x     |\
   ./    / \    | \
    \   .---.---.--.
     \ /   /
      .---.
```

If your data do not exemplify regular grid data, ImageVis3D will not be able to handle it natively. For now, your best bet is to import such data into ImageVis3D is to resample it onto a regular grid.

3 Existing File Formats

ImageVis3D might already support converting the formats you care about into its native UVF container format. You might also consider rewriting data in one of these formats to better support ImageVis3D. If your data are inherently single-block, this will be your quickest path to loading and visualizing your data.

- **DICOM stacks** (`.dcm` extension, normally): A set of DICOM files which form a volume. Use the "Load Dataset from Directory" feature to load these datasets.
- **Image stacks** (`.png`, `.tif`, `.bmp`, `.jpeg`/`.jpg`, `.qif` extensions): sets of 2D images which form a volume when rendered together. As with DICOM stacks, use the "Load Dataset from Directory" feature to load these datasets.
- **QVis** (`.dat` extension): QVis header and accompanying raw data
- **NRRD** (`.nrrd` and `.nhdr` extensions): NRRD files, as output by the Teem library and associated tools.
- **Stack** (`.stk` extension): Metamorph "stack" files; a special case of TIFFs, output by the Metamorph software package.
- **TIFF** (`.tiff`, `.tif` extensions): A TIFF stack; these differ from an image stack in that the entire volume is stored in a single file.
- **VFF** (`.vff` extension): Visualization File Format
- **Brick of Values** (`.bov` extension): A raw 3D array and associated header; a common method to export data from the VisIt software package.
- **EZRT** (`.rek` extension): Fraunhofer EZRT format.

3.1 DICOM Stacks

DICOMs are notoriously difficult to parse, work with, and have poor access times. Do not use DICOM if you are not already. We will not document the format here, but if you are dead-set on this format, you can find the entire ~3000 page specification on the official DICOM web site.

3.2 Image Stacks

ImageVis3D supports loading sets of images in the common formats. Specifications for each image type are beyond the scope of this document. If you are designing a new visualization pipeline, we would discourage reliance on image stacks due to slow access times.

For many if not all image types, image data is downsampled to 8bits before it is handed to ImageVis3D's IO subsystem. This is fixable, but at present has yet to be a pressing concern.

3.3 QVis

QVis is a simple *header + data* file format. The file which the user selects via the ImageVis3D UI, which must have the extension `.dat`, is comprised of a series of ASCII key-value pairs. One of these pairs, with the `ObjectFileName` key, gives the name of a file which contains raw data.

QVis is currently the only mechanism by which one can load color data into ImageVis3D. Other file formats are capable of storing color data, but code must be added to the converter to inform higher levels of ImageVis3D to handle the data correctly. QVis would be a good format to standardize around if you were designing a pipeline to work with single-block, raw data.

An example `.dat` file is:

```
ObjectFileName: body.raw16
TaggedFileName: ---
Resolution:     512 512 1884
SliceThickness: 1 1 1
Format:         USHORT
NbrTags:        0
ObjectType:     TEXTURE_VOLUME_OBJECT
ObjectModel:    RGBA
GridType:       EQUIDISTANT
```

This describes a `512x512x1884` dataset, stored as 16-bit unsigned integers (`USHORT` s). As such, one would expect the data file, `body.raw16`, to be 987,758,592 bytes (`512 x 512 x 1884`, times 2 bytes per element because the data are 16-bit). Despite the `ObjectModel`, this is not color data, because the `Format` implies there is only a single element per grid location. The user might have specified color data by dictating the format was `USHORT4` data, indicating there are 4 unsigned short values per grid location. This volume is isotropic ("1 1 1" `SliceThickness`, `EQUIDISTANT` grid type).

If you would like to write your own QVis data exporters, we recommend:

- Always put the `ObjectFileName` as the first key available in the header.
- Include the `TaggedFileName`, `NbrTags`, `ObjectType`, `ObjectModel`, and `GridType` keys, even though ImageVis3D will ignore them (just copy the values used above, if nothing else). You never know when some other piece of software will require those fields.
- Stick to `CHAR/BYTE`, `UCHAR`, `SHORT`, `USHORT`, `UCHAR4` (color data), and `FLOAT` for your `Format` s. Other types are not currently implemented! Patches are welcome.

3.4 Nearly Raw Raster Data (NRRD)

NRRD is a very simple format comprised of a header with accompanying data. The NRRD file format is described extensively on its web site.

ImageVis3D's implementation suffers from some quirks because it eschews the use of the more general Teem library. We would welcome contributions which would allow ImageVis3D to understand more of NRRD's fields. At the very least, please provide bug reports if you find fields which are not understood by ImageVis3D.

NRRDs can be "attached" or "detached", the former meaning that metadata and data appear concatenated in a single file, and the latter meaning that the metadata information contains a specific tag which details a second file comprising the data. In the "detached" form, the file extension is normally ".nhdr" instead of ".nrrd".

3.5 Metamorph "Stack" Files

Metamorph "Stack" files are a special case of TIFF. ImageVis3D can read them by forcing the underlying `libtiff` library to read TIFF headers which it does not natively understand.

This is a compatibility format; it exists solely to ease migration from past and present Metamorph users. If you have a choice, do not use this format.

3.6 TIFF Volumes

While the majority of TIFFs contain just a single image inside them, a little known feature of the format is that it stores each image in what it calls a ‘Directory’. Most TIFFs you would find on the web or in similar places consist of a single Directory, but some scientific software can generate these multi-directory TIFFs.

TIFF volumes suffer from poor access time, because each image slice is stored separately instead of as a continuous volume. Further, the bundled converter for TIFF volumes is among the worst in ImageVis3D: ImageVis3D assumes that an input volume is a contiguous chunk of data. As such, the converter must read each slice and append it into a single volume, requiring significantly increased temporary disk storage, as well as a costly additional conversion step.

It is not recommended you utilize TIFF volumes if given the choice, but the format may be convenient for interoperation with other software. If you do go this route, we recommend utilizing the `libtiff` library, which has excellent documentation.

3.7 VFF

This format was written for a specific collaborator. It is a ‘header + data’ format, very similar to an attached NRRD. Little other information is available at this time.

3.8 Brick of Values

This is a compatibility format, meant to allow importing data from the VisIt software package. It is a header + data format, very similar in concept to a detached NRRD.

In some cases, a BOV may be split into multiple files, using a special ‘%‘ notation for the filenames within the header. This variation is not supported by ImageVis3D. To ensure you will only have a single data file, make sure you have a "single block" dataset in VisIt. You can obtain one by applying a non-distributed ‘Resample’ operation on the data.

Here is a VisIt-python script which converts a dataset in the `/path/to/dataset.silo` file into a 512^3 volume which ImageVis3D could import:

```python
import os
db = "/path/to/dataset.silo"
OpenDatabase(db)

AddPlot("Pseudocolor", "helium")
AddOperator("Resample")
ra = ResampleAttributes()
ra.samplesX = 512
ra.samplesY = 512
ra.samplesZ = 512
ra.distributedResample = False
SetOperatorOptions(ra)

DrawPlots()
dbAtts = ExportDBAttributes()
dbAtts.db_type = "BOV"
dbAtts.filename = "singlehelium"
dbAtts.variables = ("helium")
ExportDatabase(dbAtts)

RemoveAllOperators()
DeleteAllPlots()

CloseDatabase(db)

import sys
sys.exit(0)
```

After modifying the `db = ...` line, invoke the script with: `/path/to/visit -cli -nowin -s script.py`.

3.9 Fraunhofer EZRT

This is a simple binary format which directly supports a collaborator's work. Not much else is known at this time, but you could ask for more information on the tuvok-developers mailing list.

4 Adding a Converter

4.1 Introduction

ImageVis3D supports some file formats "out of the box". In many cases, the easiest route to getting your data into ImageVis3D is to write it out in a format the program can read natively. However, this might not always be a viable option. This section describes the work involved in enabling ImageVis3D to convert a new file format into UVF.

Writing a converter is a good choice if you have pre-existing, single-block data which is not readable by any of the converters which ship with ImageVis3D.

4.2 ImageVis3D Conversion Steps

ImageVis3D's IO scheme is based around a multi-stage paradigm. In the first stage, converters identify relevant metadata and modify the data into a raw input stream. Subsequent stages perform tasks such as endian conversion, compute derived metadata, and generate the hierarchy of bricks needed for interactive rendering.

4.3 Writing Your Converter

The basic steps involved in writing your own converter are:

1. Create a new class derived from `RAWConverter`
2. Register your format's file extension with the IO subsystem.
3. Modify a conversion routine to create metadata and (potentially) rewrite it as a raw data stream.

4.3.1 Converter Skeleton

You'll need to create both a header file (`.h`) and an implementation file (`.cpp`) for your new format. They should define a new class, derived from `RAWConverter`, which implements the methods `ConvertToRAW`, `ConvertToNative`, and `CanExportData`. Here is an example header file:

```
class YourConverter : public RAWConverter {
public:
  YourConverter();
  virtual ~YourConverter() {}

  virtual bool ConvertToRAW(const std::string& strSourceFilename,
                            const std::string& strTempDir,
                            bool bNoUserInteraction,
                            UINT64& iHeaderSkip, UINT64& iComponentSize,
                            UINT64& iComponentCount, bool& bConvertEndianess,
                            bool& bSigned, bool& bIsFloat,
                            UINT64VECTOR3& vVolumeSize,
                            FLOATVECTOR3& vVolumeAspect, std::string& strTitle,
                            UVFTables::ElementSemanticTable& eType,
                            std::string& strIntermediateFile,
                            bool& bDeleteIntermediateFile);

  virtual bool ConvertToNative(const std::string& strRawFilename,
                               const std::string& strTargetFilename,
                               UINT64 iHeaderSkip, UINT64 iComponentSize,
                               UINT64 iComponentCount, bool bSigned,
                               bool bFloatingPoint,  UINT64VECTOR3 vVolumeSize,
                               FLOATVECTOR3 vVolumeAspect,
                               bool bNoUserInteraction,
                               const bool bQuantizeTo8Bit);
  virtual bool CanExportData() const { return false; }
};
```

Now define a skeleton of an implementation in a corresponding `.cpp` file:

```
YourConverter::YourConverter() {}

bool YourConverter::ConvertToRAW(...)
{
  return false;
}

bool YourConverter::ConvertToNative(...)
{
}
```

(I've omitted the arguments here; they should be identical to those in the aforementioned header file.)

Congratulations! You've got a minimal converter which can be plugged in to ImageVis3D. Let's get it part of ImageVis3D before we start trying to read any data.

4.3.2 Building Your Reader

ImageVis3D stores the list of files that are part of the program in a few places. The files which are part of the IO subsystem are currently listed in 'Tuvok/Tuvok.pro'. In that file, you'll find two variables, 'HEADERS' and 'SOURCES', which identify which files to build. You'll need to add your converter to the list specified in both variables.

```
HEADERS += \
         ...
         IO/AbstrConverter.h \
         IO/BOVConverter.h \
         IO/YourFileHere.h \
         IO/BrickedDataset.h \
         ...
SOURCES += \
         ...
         IO/AbstrConverter.cpp \
         IO/BOVConverter.cpp \
         IO/YourFileHere.cpp \
         IO/BrickedDataset.cpp \
         ...
```

Once you've done that, rerun 'qmake' from the root directory, and 'make' to rebuild — your converter should now be part of ImageVis3D!

Important
On Windows, the solution/project files do **not** respect the settings given in the `Tuvok.pro` file. You must manually add your new files to the Visual Studio project in the normal way.

Caution
On Windows, the solution/project files have diverged from the settings given in the qmake project file. You will need to add your files to the Visual Studio project file in the normal way. Do **not** run qmake on Windows, or you will not be able to compile ImageVis3D!

4.3.3 Register Your File Extensions

The constructor for your new converter should modify two internal class variables: `m_vConverterDesc` and `m_vSupportedExt`. The former should be set to a short human-readable string that describes the file format. This must be a single line and should generally be a short phrase of a few words or less. The second, `m_vSupportedExt`, should be populated with any extensions which are common for your file format.

Note
You may leave `m_vSupportedExt` empty if you reimplement the `CanRead` method, described later in this document.

See `BOVConverter.cpp` for an example.

A good test at this point would be to add:

```
MESSAGE("constructor!");
```

to your constructor, and:

```
MESSAGE("convert!");
```

to your `ConvertToRAW` function. Run ImageVis3D and enable the "Message" channel in the Debug Window (under "Help") tell it to load your data file. The conversion will fail, but in the debug log you should see both of those messages (among many others).

Tip
You can use the `WARNING` and `T_ERROR` macros to report warnings and errors, respectively, in your converter.

4.3.4 Modify Raw Conversion Routine

This is where all of the work happens. The purpose of this routine is to take an input data file, fill in the appropriate metadata as given by the arguments, and create a `strIntermediateFile` raw file with implicit structure. Let's start with the arguments to the method:

- `strSourceFilename` - The filename where your data lives. This is the file that the user selected via the ImageVis3D UI.
- `strTempDir` - if you need to create any temporary files, you should prepend this directory string to each of the filenames.
- `bNoUserInteraction` - if `true`, any ambiguities should be treated as fatal errors. Otherwise, you may query the user for more information (say, via a `QMessageBox`).

Note
Qt UI elements may not be used in the Tuvok IO subsystem. If you want to perform a graphical query when `bNoUserInteraction` is `false`, you must put the code into the "imagevis3d" repository. Since converters are registered dynamically, this will work fine; see the `DialogConverter` code.

- `iHeaderSkip` - Many formats are "sectioned", in that an initial header is given which describes the data, and a raw chunk of data follows the header. Write the byte offset of the start of such data into this header; write `0` if there is no header or this field makes no sense for your data format.
- `iComponentSize` - write the number of bits per component into this argument. Note this is **bits**: so-called "short" data should write `16` into this field.
- `iComponentCount` - write the number of components in the dataset into this variable. This will almost always be `1`, because volume rendering really only makes sense for scalar fields. ImageVis3D also currently supports "color data", or RGBA data, in which case you would write `4` into this variable. Any other setting is likely to fail later on in processing.
- `bConvertEndianess` - set this to true if the endianness of the data differs from the endianness of the current platform. You can use the static `EndianConvert::IsBigEndian()` method to determine the endianness of the currently-running ImageVis3D.
- `bSigned` - set to true if the data are signed.
- `bIsFloat` - set to true if the data are floating point. This only makes sense in combination with certain values for `iComponentSize`
- `vVolumeSize` - the dimensions of the dataset, in X (index 0), Y (1), and Z (2)
- `vVolumeAspect` - default aspect ratio of these data, indexed just like `vVolumeSize`. Normally, set this to `(1,1,1)`.
- `strTitle` - any special string which identifies or describes the dataset. Perhaps the name of the variable stored in this field.
- `eType` - See UVF's source for more detail, but generally just set this to `UVFTables::ES_UNDEFINED`.
- `strIntermediateFile` - if you need to create a new file, set this to the new file name. Otherwise, copy `strSourceFilename` into here.
- `bDeleteIntermediateFile` - if you need to create a new file, you should set this to `true` to make sure ImageVis3D deletes the file when it no longer needs it. Otherwise, make sure it is `false`, or ImageVis3D will try to delete the input file!

The format of `strIntermediateFile` should simply be raw data which varies slowly in X and quickly in Z. These data should be written in "raw" format: do not use C++'s formatted IO routines if you need to generate these data.

If all goes well, you should return `true` from this method.

4.3.5 Optional: Reimplement the `CanRead` Predicate

Since there are many converters available, at various times the IO subsystem needs to know **which** format within the candidate set is the appropriate one to use. It does this via the `virtual CanRead` method.

```
virtual bool CanRead(const std::string& filename,
                     const std::vector<int8_t> bytes) const;
```

The default implementation of this method is based purely on file extensions. The extension[s] used for your format are the ones you added to the `m_vSupportedExt` vector in your constructor. For most formats, this implementation will be perfectly fine.

However, some converters need to know a bit more. You might, for example, be working with a file format that relies on **prefixes** for file names instead of **postfixes** (i.e. "extensions"). You can override the `CanRead` method to implement a predicate more specific to your file format. This method should return `true` if you are reasonably sure that your `ConvertToRAW` method will succeed for the given file, and `false` otherwise.

The method takes two arguments. The first is the name of the file that the IO subsystem is trying to find a converter for; for ImageVis3D, this will be the file selected by the user in the GUI. If the user has selected multiple files (for example, while attempting to convert a time-dependent dataset), this will be the first file in the sequence. The second argument is an array which contains a few bytes from the beginning of the file (again, the first file if multiple files have been selected).

Important
Although the method is given the full file name and could easily open and scan the file to see if it is valid, please do **not** do this in your converter. If every converter operated in this fashion, identifying the appropriate converter would be extremely slow. The `bytes` array argument should be sufficient to identify the file; if you need more data to do so definitively, please contact the lead developers via the mailing lists and ask them to increase the number of bytes given to the method.

You can use the `filename` parameter to key into any sort of custom file naming procedure that your file format has. Many formats also implement some concept of `magic` bytes: the first few bytes of the files given in this format might always be statically set to a specific value. As examples, the first 4 bytes of every NRRD file spell out "NRRD"; the QVis file format is based on a series of key-value pairs, and it is common for the first key to be "ObjectFileName". `CanRead` implementations for these formats could key into such conventions to verify that the file is what it says it is.

Note
You do not need to go all-out to detect errors at this stage. For example, you should not attempt to identify if the file is corrupted in the `CanRead` method. This method is meant to quickly whittle down the list of available converters, and as such should do relatively little work, and certainly no dataset-sized work. The correct place to detect file corruption would be in the `ConvertToRAW` method.

4.3.6 Optional: Implement Native Conversion

Many converters in the IO subsystem implement the `ConvertToNative` method. This allows one to use ImageVis3D to convert data from one file format to another. To do this, implement the method and modify it to return `true`. Make sure to also modify the `CanExportData` method to return `true`.

4.4 Examples

You can read ImageVis3D's existing code for converting data to get hints about how your converter should work.

- `REKConverter.cpp` - This is the smallest of ImageVis3D's converters. The EZRT file format that it reads is an example of a "header plus raw data" format; as such, the converter reads in some metadata, and then sets up the `iHeaderSkip` variable to the location where the data starts. No new output file is necessary.
- `QVISConverter.cpp` - This is purely a "header" file format: the user is expected to select a file which has a simple ASCII header. One of the fields in this header gives the name of a raw filename which stores the data. The converter finds that field and sets `strIntermediateFile` to be the raw filename. Since the raw file is actually **part** of the input dataset, the converter deliberately sets `bDeleteIntermediateFile` to `false`.
- `TiffVolumeConverter.cpp` - A little-known feature of TIFF is that it supports so-called "directories", which provide a mechanism to store multiple images in a single file. If these images align, then a single TIFF file forms a volume instead of just an image. This converter provides an example of using an external library to read the data, and then rewriting that data as a raw binary file that the rest of ImageVis3D's IO routines can handle.

5 Implementing a Reader Parallel to UVF

If your data format meets certain criteria, it is possible for you to write a reader which sits parallel to UVF. This will allow ImageVis3D to read your data natively, without any conversion process. For extremely large data, a conversion process is infeasible.

The criteria your format must meet are:

- You must meet the *standard* requirements for ImageVis3D data: namely, your data must be defined on a regular, though potentially anisotropic, grid.

- Your data must be *bricked*; a large volume is composed of a set of small volumes.
- You have at least two levels of resolution for your data; in general, more levels will be better, within reason.
- The coarsest resolution of your data consists of a single *brick*.
- You can load the coarsest resolution of your data in 3- or 4-hundred milliseconds.

Further, this is the most complicated of methods to get your data into ImageVis3D. Someone in your lab must be capable of writing C++ code.

5.1 Writing Your Reader

The basic steps involved in writing your own reader are:

1. Create a new class derived from `FileBackedDataset`.
2. Register your format with the IO subsystem.
3. Implement `CanRead` and `Create` methods, to allow the IO system to identify your dataset type.
4. Implement methods to query data and metadata from your dataset.

5.1.1 Reader Skeleton

You'll need to create a header file (`.h`) and an implementation file (`.cpp`) for your new format. They should define a new class which includes a large set of methods.

1. Copy `Dataset.h` to `<YourFormat>.h`
2. Modify `YourFormat.h`:
   - Add the header `FileBackedDataset.h`, remove other headers.
   - Derive the class from `FileBackedDataset`.
   - Make all of the pure virtual methods simply virtual.
   - Add `CanRead` and `Create` methods with the same signatures as from `FileBackedDataset`.

Next, stub out an implementation file. At this stage, we highly recommend you do not attempt to implement your reader, but instead just add the minimal return statements required for compilation. For instance, implement the `GetLODLevelCount` method like so:

```
virtual UINT64 GetLODLevelCount() const { return 0; }
```

etc.

5.1.2 Building Your Reader

You will need to add your new reader to the build. The files which are part of the IO subsystem are currently listed in the *Tuvok/Tuvok.pro*. You'll find `HEADERS` and `SOURCES` variables in that file, which list all of the files involved in the build. You must add you reader to both variables.

```
HEADERS += \
         ...
         IO/AbstrConverter.h \
         IO/BOVConverter.h \
         IO/*YourReaderHere.h* \
         IO/BrickedDataset.h \
         ...
SOURCES += \
         ...
         IO/AbstrConverter.cpp \
         IO/BOVConverter.cpp \
         IO/*YourReaderHere.cpp* \
         IO/BrickedDataset.cpp \
         ...
```

Important
On Windows, `qmake` is unfortunately no longer used. You must add your files to the Visual Studio project file in the normal way on this platform. Make sure to add your files to the `.pro` file anyway, though — it will be required for the Linux and Mac builds.

You can now compile ImageVis3D and test that your new class includes all the right methods.

5.1.3 Register Your Data Format

For ImageVis3D to attempt using your format, it must know how to identify your files. Two steps are required here: you must implement the `CanRead` and `Create` methods, and you must register your format with the `IOManager`.

`CanRead` and `Create`

`CanRead` is straightforward; it must return `true` when the file is in your format. Normally you would do this by verifying the magic bytes at the beginning of the file.

`Create` is a simple "virtual constructor"; you just need to instantiate an object of your class. The constructor of your class, or the `Create` method itself, is allowed to throw any exception derived from `DSOpenFailed` (see `TuvokIOError.h`) to indicate any errors encountered while actually opening the file.

Important
Do **not** report errors in the `CanRead` method! Save error detection for `Create`.

`uvfDataset.cpp` provides a good example of how these methods might be implemented. Note that `UVFDataset` does its error detection in `Open`, which is automatically called from the constructor.

Registration with the IO Subsystem

This just requires a small addition to the `IOManager`. Open up `IOManager.cpp` and find the constructor. There is already a line to register UVF with the IO manager:

```
m_dsFactory->AddReader(std::tr1::shared_ptr<UVFDataset>(new UVFDataset()));
```

Add a similar line for your reader. You will also need to add a `#include` line slightly above the constructor.

Test!

At this point, ImageVis3D should be able to both identify and at least attempt to read your data. It will return garbage values for just about every inquiry, but now would be a good time to test the registration. Selecting one of your files from the ImageVis3D UI should attempt to open it, and promptly report some sort of error — probably complaining about the lack of data in the file.

The exact error is unimportant. The important thing is that ImageVis3D does not prompt you for a UVF file name after selecting your file. This indicates that ImageVis3D knows about your format and recognizes that it does not require conversion.

5.1.4 Implement `Dataset` Methods

There are too many dataset methods to exhaustively outline implementations for each one of them. We outline them by categories here. For specific details, see the doxygen comments, and/or follow the implementation in `uvfDataset.cpp`.

Note
Never cache any **data** in your reader implementation. Tuvok, ImageVis3D's renderer, implements caching of bricks at a level slightly higher than your dataset. You will defeat its memory management routines by wasting memory in your dataset, which may lead to excessive swapping. Caching metadata, or other small information about data, is fine. If you feel you must store a large amount of information, you should tie into the memory management system to "register" your memory.

Histograms

ImageVis3D requires 1D and 2D histograms to use as the backdrop of the transfer function editors.

UVF stores these histograms in the file. The `GetHistograms` method initializes the `m_pHistXD` variables. This occurs when the UVF is first opened. The alternative would be to generate these on the fly and cache the results. We highly discourage such an ad hoc histogram calculation: for one, users will have to pay for it every time they open your file. Secondly, the calculations can take significant time, and that time is taken between when the user selects the file and when they see a window opening on their screen. Your format will look very slow if it does a long calculation at this point.

Do note that the calculation of a 2D histogram is an **extremely** expensive operation. ImageVis3D only calculates the 2D histogram for the coarsest level of detail, *even though we have a preprocess to generate these data*! You are welcome to implement an exact solution, but be careful about how much work you do at open time.

Brick Inquiries

`GetBrick...` functions query data and metadata about a brick. Of course, the most important of these is the `GetBrick` method itself, which loads the data for a brick. This method takes a `BrickKey` and is expected to fill the `std::vector` argument with the data for that brick. The `BrickKey` is under your control: when you open your file, you should make a series of `AddBrick` calls, each with a unique `BrickKey`. ImageVis3D will then use those `BrickKey` s to request data from your Dataset implementation. You can "encode" whatever information you need into that key; as an example, `UVFDataset` encodes the spatial position of a brick in the "brick index" portion of the key. The `IndexToVector` method then decodes the "brick index" back into the 3D index.

Data Type and Size

There are a variety of methods, such as `GetBitWidth` and `GetIsSigned`, which are used to identify features of the dataset. These provide important information on how the data received out of `GetBrick` is interpreted.

Optional: Rescale Factors

These are normally handled by your parent classes. However, there is one method, `SaveRescaleFactors` that you may reimplement. If the user changes the rescale factors in the UI and then selects "Save To File", ImageVis3D will call this method. You should re-open your file in read-write, and add some metadata about the spacings in the X, Y, and Z axes.

Optional: First/Last Bricks

`Dataset` contains two methods, `BrickIsFirstInDimension` and `BrickIsLastInDimension`, which inform the renderer whether or not they are the "leftmost" or "rightmost" brick. `BrickedDataset` does provide a default implementation for these methods, so ImageVis3D will work even if you do not override them. However, `BrickedDataset` has very limited knowledge of your dataset; it requires `O(n)` time in the number of bricks to answer this. Some formats can answer this in constant time. If you can do so, it is highly recommended you implement these methods; it will significantly speed up ImageVis3D's "frame planning" algorithms.

Optional: Acceleration Queries

These consist of the `ContainsData` methods. These are meant to key into any acceleration structures you might store in a file, and allow ImageVis3D to reject bricks without rendering them. This can significantly increase performance when available.

They are given a brick key and some kind of information on what ImageVis3D is looking for. There are three methods, one for each of the primary render modes. As an example, the two-argument version of `ContainsData` is used when rendering isosurfaces. If the user selects an isosurface of 42, and you know that a given brick only has values from 96 to 428, then there is no reason to render that brick, and so `ContainsData` would return false.

Do not scan your data to identify the results of these methods. The GPU will do that much quicker than you could ever dream of doing here. However, if you have precomputed this information, implement these methods to significantly increase performance.

Optional: Export

ImageVis3D can also be used as a converter from one file format to another. The `Export` method helps in doing this. If you implement this method, you should serialize your data to the file name given in the argument, in a completely "raw" (unbricked) form.

If you do not care about this functionality, just ignore this method. The default implementation already indicates that the export operation has failed.

6 Converting Your Output Format to ImageVis3D's UVF

UVF is more of a container format than a full file format. The primary author of this document considers it more of a filesystem than a file format. This is because UVF is very general, relying on the applications which implement it for some of the semantic value of the data. As an example, with existing code it would be easy to write a 9-dimensional, 7-channel tensor dataset into a UVF file. ImageVis3D will have absolutely no idea how to render that data, but it is perfectly valid as a UVF file.

UVF does, however, provide a means to query the kind of data stored in the file. If ImageVis3D encountered such a dataset, it would simply ignore that data. This is the recommended way to handle UVF files: pull out the portions your application understands, and ignore the rest.

This naturally only works if the format itself supports multiple *portions*. UVF supports this natively, and in fact ImageVis3D itself makes use of a few different *portions*, or, in UVF parlance, *blocks*. Each block is almost like a file in-and-of itself; it contains a small header detailing specifics of that block, followed by whatever payload is relevant. A global header presides over the entire UVF file, giving very general information and helping to index the various blocks which exist in the file.

An academic paper about UVF was published while the format was still in its infancy. The format has changed slightly since then, but not in concept, and the changes have actually been surprisingly minor. As such, it still provides a relevant overview of UVF's format, features, and design goals.

6.1 Unified Volume Format Reference

In brief, UVF files start with a common header. After the header follows a series of blocks, each with specialized data contained within them. All blocks contain a block header, and share a common subset of header information, which appears before block-specific header information.

6.1.1 Master Header

The first bytes within a UVF comprise the *Master Header*: a global header for the file which describes globally relevant parameters. In order, these components are:

- The raw, constant bytes: `0x55 0x56 0x46 0x2d 0x44 0x41 0x54 0x41`. These are UVF's "magic" bytes; many files may begin with these bytes, but a UVF file will never begin with anything different. Note that these bytes, if interpreted as ASCII characters, spell out the string "UVF-DATA".
- A single byte which details whether the file stores data in little or big endian. Zero values imply little endianness; non-zero values imply big endian data. This effects all subsequent data in the file, from the next values in this header, to data stored within UVF's blocks.
- An unsigned, 64-bit integer which gives the version of the UVF file. ImageVis3D has been writing out version 2 UVF files since 1.0 was released in May 2009. ImageVis3D does not support version 0 or 1 UVF files, and any differences between those and version 2 will not be documented here.
- An unsigned, 64-bit integer which details how the file checksum was calculated, according to the following table.

Table 1: Integer Representation of Checksum Calculation Method

Value	Meaning of the 64-bit value
0	No checksum exists
1	The checksum was calculated via a simple crc32 algorithm
2	The popular MD5 algorithm was used to calculate the checksum
3	An unknown method was used to calculate the checksum
*	Undefined. Reserved.

- An unsigned, 64-bit integer containing `n_checksum_bytes`, the number of bytes in the checksum.

Important
The number of bytes in the checksum **always** appears, even if the type of checksum is *none* (i.e. `0`, as per above)! Of course, this field will be set to 0 if that is the case.

- `n_checksum_bytes` of data which comprise the file's checksum.
- An unsigned, 64-bit integer giving the number of bytes to seek *from the position after reading this element* (i.e., use `SEEK_CUR`). For example, if you are using the C streams interface, code to read this element and seek to the first block might look like this:

```c
uint64_t seek;
fread(&seek, sizeof(uint64_t), 1, fp);
if(needed) { /* as per the endianness byte, above */
  seek = bswap_64(seek);
}
fseek(fp, seek, SEEK_CUR);
/* fread now will read the first byte of the first real block */
```

- Any number (including 0) of undefined bytes. Future UVF versions may add meaning to the bytes starting here and up to the first data block, so *portable UVF readers must seek over these bytes*!

6.1.2 Data Blocks

There are relatively few data blocks currently defined. They are: 1D histograms, 2D histograms, Key-value pairs, maximum and minimum data values, and raster data.

All data blocks start with a common header. The data in this header, in order, is:

- An unsigned, 64-bit integer detailing the size of . . .
- . . . a series of bytes forming an ASCII string which describes the block's type. This element is intended for display in, for example, a UI that lets one view and manipulate UVF blocks, **not** for doing string comparisons to figure out what kind of data are in the block.
- The semantic type of a block, which details the specific data block present.

Table 2: Semantic Block Types

Value	Meaning of the 64-bit value
0	Empty. This should not occur in production.
1	Regular, N-dimensional gridded data ("Raster data").
2	An N-dimensional representation for a transfer function
3	Preview image, meant to give an idea what is in the file
4	Key value pairs: strings used to provide extraneous metadata
5	1D Histogram
6	2D Histogram
7	Maximum and minimum data values (acceleration structure)
*	Undefined. Reserved.

- The type of compression used for the block.

Table 3: Block Compression Type

Value	Meaning of the 64-bit value
0	None; the block is not compressed
*	Undefined. Reserved.

- An unsigned, 64-bit integer offset to the next block. This is relative to the position of the file pointer *after* reading the offset.

1D Histogram Data Block

This data block essentially contains just an array of values which form a histogram for a scalar dataset.

- First 8 bytes: unsigned integer which gives the `histogram_size`
- `histogram_size` 8-byte elements following that: the histogram. Indices into the array detail the data value; values in the array give the number of times that data value occurs. Thus, if `arr[8] == 19`, then the value `8` occurs in the dataset `19` times.

The association between histograms and datasets is purely implicit. If a UVF contains two dataset blocks and one 1D histogram block, it is technically ambiguous as to which dataset the histogram was computed from. In ImageVis3D's UVF, mappings are 1:1; the situation listed here would be an error.

2D Histogram Data Block

Stores a 2D histogram for a given dataset. Essentially, this stores a contiguous array of 1D histograms. The histogram is assumed to be "square": the second dimension does not vary from its initial value. It is implied that the second dimension is gradient magnitude.

- First 32-bits: IEEE-754 representation of the max gradient magnitude.
- Following 8 bytes: unsigned integer count, `elems_X`.
- Following 8 bytes: unsigned integer count, `elems_Y`.
- Final `elems_X` x `elems_Y` x 8 bytes: the histogram, with X as the slowest moving dimension.

Key Value Pairs Block

Stores a set of simple string-based key-value pairs. This is typically used to store metadata about the dataset which are not relevant to visualization.

- First 64-bits: unsigned integer `n_elements`.
- `n_elements` of:
  - 64-bit unsigned integer giving a `key_length`
  - `key_length` bytes storing an ASCII "key" string.
  - 64-bit unsigned integer giving a `value_length`
  - `value_length` bytes storing an ASCII "value" string.

Raster Data Block

This is where ImageVis3D stores the data it renders. It is intended to store N-dimensions, regularly gridded data.

- First 64-bits: unsigned integer `n_dimensions`
- Only if `n_dimensions` is non-zero:
  - `n_dimensions` 64-bit semantic block identifiers
  - `n_dimensions+1` x `n_dimensions+1` IEEE-854 64-bit FP values storing a default transformation.
  - `n_dimensions` 64-bit unsigned integers giving the size of the domain, slowest dimension first.
  - `n_dimensions` 64-bit unsigned integers giving the size used for bricks, slowest dimension first.
  - `n_dimensions` 64-bit unsigned integers giving the overlap among bricks
  - `n_dimensions` 64-bit unsigned integers giving the factor by which successive domains are downsampled.
  - `n_dimensions` 64-bit unsigned integers ???
- `n_dimensions` 64-bit unsigned integers giving the number of LODs for each dimension of the data.
- a 64-bit unsigned integer, `n_elements`
- `n_elements` 64-bit unsigned integers giving the element size (number of bytes element) per dimension. . . .

(Sadly, this part of the documentation is not complete. In the meantime, see the source for more information.)

7 Contribute Back

While not strictly related to the main purpose of this document, we would like to take this opportunity to encourage you to make any changes you make to ImageVis3D's source code available.

ImageVis3D is an open source project. You can download the source, modify it to your heart's content, and redistribute it however you would like. The licensing is incredibly liberal; you could even recompile it as-is and sell it as a commercial product. It is not **required** that you make your source available. It is not **required** that you participate in future development. However, it is in everyone's best interest that you do so.

7.1 Benefits of Contribution

The benefits to the ImageVis3D project are somewhat obvious. If you write some code to allow ImageVis3D to read a new type of data, anyone else can then pick up the program and use it to visualize that kind of data. This enables ImageVis3D to be used in more domains, increasing the project's user base, which significantly helps us pursue funding for development of ImageVis3D features and maintenance.

However, contributing your code actually helps **you** as well:

- The lead developers will verify that your format is still readable with each new release of ImageVis3D
- Internal APIs within ImageVis3D change from time to time; if your source is part of ImageVis3D, we will update your code to adhere to the new interface.
  - It is safe to assume that, if your code is not part of the ImageVis3D tree, it will eventually stop compiling and require manual edits.
  - New features added to ImageVis3D will automatically apply to your format/code.
- Your code will receive more testing than you could perform on your own.

7.2 What We Ask From Contributors

We ask that you don't just throw a bunch of files at us and expect we do the work of merging in changes or fixing blatant issues. We ask that you submit:

- Well-formatted, complete patches
  - `svn diff` (or even `git diff`) formats are preferred, as opposed to files
  - Patches should be against the current source in version control, not against the previous release.
  - If plausible, split multiple changes into multiple patches
  - Each change should compile and work if applied in order (i.e. do not have one patch break the build and the next one fix it — just use one patch).
  - Should compile using MSVC and recent gcc's.
    - We understand that you might not have both these compilers available. Please just do the best you can.
  - Use MESSAGE, WARNING, and T_ERROR to give information on what the code is doing.
  - Try to break lines at 80 characters
  - Remove whitespace at the end of each line
- Rights to distribute your code under the MIT license
  - Do note that permission to do this might have to come not just from yourself, but from your organization and/or funding organization as well. Do **not** *assume* you have the right to distribute code you have written!
  - More general rights are fine too; you might release the code into the public domain, for example.

- For support of new file formats:
  - At least one dataset which can be read by the new code
  - An image of the dataset, preferably rendered in some other software, though an ImageVis3D image is acceptable. We will use this to verify that the code actually works.

The dataset request is a **requirement**, not an option. No exceptions! If we cannot test your code, it **will** break at some point. Even if your file format is common, do not assume we will go scouring the web for sample data — that is your responsibility.

7.3 Logistics

You should send your patches to the tuvok-developers mailing list. If you have very small sample data (i.e. measured in kilobytes), you can send that along through the list as well. If your sample data are large, you may:

- Upload them via FTP to `ftp.sci.utah.edu`; anonymous uploads are allowed in the `upload` directory.
- Use the `Report an Issue` dialog and attach the dataset and sample image to the bug report. Be sure to include your email and at least a brief message mentioning the thread you've started on tuvok-developers.

In both cases, please send a ping to tuvok-developers and let us know you've uploaded data. You'll need to send a mail there anyway, to include the patches.

7.3.1 Privacy

Please note any restrictions on the data at submission time. We are used to receiving data which we are not allowed to use in our publications, or that we are not allowed to demo, etc. If this restriction is time-limited (i.e. "do not demo this until I've published a paper about <X>!"), please note that as well.

Do note that any data which is uploaded to SCI via FTP, or uploaded via ImageVis3D's "Report an Issue" feature, is readable by any member of the SCI Institute. The ImageVis3D developers will gladly copy it to our workstations and then delete it from the public FTP as quickly as possible, but there will always be a window where data are available to the internal network.

Furthermore, the SCI Institute is an Institute within a University. Security policies are somewhat liberal; there is not a notion of a "secure network" versus an "open network", as is common at some national labs. We will make reasonable efforts to ensure your data remain as private as you would like, but do recognize that no computer network is ever completely secure, and SCI is no different. As such, please do not send us extremely sensitive data.