Adding a New File Format to ImageVis3D

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

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 document describes the work involved in enabling ImageVis3D to read a new file format.

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 2.1 Example Regular Gridded Data (`.` represents a grid location)

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

Example 2.2 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 Fork in the Road

There are two ways by which you can enable ImageVis3D to read your file format. The first, simpler way is to write a converter class which can help ImageVis3D's IO routines convert your data into a special type of the UVF file format. The second, more complicated route is to inform ImageVis3D of a new dataset type, and write an implementation which will expose your data to ImageVis3D directly.

The advantage of the latter approach is that no conversion will be necessary before you can visualize your data in ImageVis3D. However, this method is only viable for multiresolution, bricked file formats.

This document currently only describes the conversion path.

4 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.

5 Writing Your Reader

The basic steps involved in writing your own reader 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.

5.1 Reader 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 reader which can be plugged in to ImageVis3D. Let's get it part of ImageVis3D before we start trying to read any data.

5.2 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 reader 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 reader 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. Do **not** run qmake on Windows, or you will not be able to compile ImageVis3D!

5.3 Register Your File Extensions

The constructor for your new reader should modify two internal class variables: `m_vConverterDesc` 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. Secondly, you should add any extensions your file format supports to the 'm_vSupportedExt' vector.

See `BOVConverter.cpp` for an example.

A good test at this point would be to add:

```
std::cerr << "constructor!" << std::endl;
```

to your constructor, and:

```
std::cerr << "convert!" << std::endl;
```

to your `ConvertToRAW` function. Run ImageVis3D from the command line and tell it to load your data file. The conversion will fail, but in the terminal you started ImageVis3D from you should see both of those messages.

5.4 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.
- `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`).
- `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 (1)
- `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.

5.5 OPTIONAL: Implement Native Conversion

Many readers in ImageVis3D 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`.

6 Examples

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