HDF5 thumbnailer

Author:

Marc Schädeli
Semafor AG
scm@semafor.ch

Basel, 2017-09-21

Contents

1 Introduction **3**
1.1 Purpose . . . 3
1.2 Definitions . . . 3
1.2.1 XMP . . . 3
1.3 System overview . . . 3
1.4 Source structure . . . 4

2 Building and installing **5**
2.1 Linux . . . 5
2.2 Windows . . . 5
2.3 OSX . . . 5

3 Components **7**
3.1 Python scripts . . . 7
3.1.1 thumbnailInserter.py . . . 7
3.1.2 metadataReader.py . . . 8

4 Userblock storage **9**
4.1 Overview . . . 9
4.1.1 How the userblock works . . . 9
4.2 Position in the userblock . . . 9
4.3 Signature . . . 9

1 Introduction

1.1 Purpose

The purpose of this document is to describe the functionality of the HDF5 thumbnailer and how to use it. It should also tell how to add your own data to the HDF5 userblock without conflicting the current XMP metadata.

1.2 Definitions

1.2.1 XMP

XMP is a format by Adobe, created to store metadata and thumbnails, and to be embedded into different file types. The XMP data can either be stored in a sidecar file next to the HDF5 file or directly in the file's userblock.

The Python scripts allow you to specify additional data to populate the XMP with when inserting or updating the data. The scripts can also read that data from the XMP file again in JSON format for further use.

1.3 System overview

The purpose of this thumbnailer is to generate thumbnails for HDF5 files by extracting them from XMP data. The XMP data is either stored in a sidecar file or embedded into the HDF5 userblock, prefixed by a signature.

The use of that signature should also be there to make storage of any data in the userblock more unified safer from overriding. A more detailed description about the signature can be found later in the document.

1.4 Source structure

```
/
├── common ..... Core functionality for extracting a thumbnail
├── doc ..... Documentation
├── examples Example HDF5 and XMP files with varying thumbnail store methods
├── hdf_python ..... Python scripts for inserting, reading and updating
│   metadata/thumbnails
│   └── tests ..... Unit tests for the Python scripts
├── linux ..... Linux implementations for the thumbnailer
│   ├── gnome ..... GNOME implementation for Nautilus
│   ├── kde ..... KDE implementation for Dolphin
│   └── tests ..... Unit-tests for the core thumbnailer functionality
└── windows . Windows implementation for the thumbnailer (Visual Studio project)
```

In the section Components 3 the important parts will be explained further.

2 Building and installing

2.1 Linux

The Linux versions use CMake for their build system.

To compile the thumbnailer, a build folder needs to be created. Open a terminal in the build folder and execute the CMake command.

```
cmake -DCMAKE_BUILD_TYPE=Release <path to linux folder in project>
```

With this command it will configure for the GNOME and KDE versions.

To only build the GNOME or KDE version, use these options to turn one of the platforms off:

```
-Dkde=OFF or -Dgnome=OFF
```

After it is done configuring, the *make* command can be executed to compile it. When it is compiled, the *make test* command can be used to run the unit-tests or the *make install* can to install it system-wide.

After the install command, it will work in GNOME immediately, in KDE the Dolphin plugin needs to be enabled first.

To enable it open Dolphin and go to *Control* > *Configure Dolphin* > *General* > *Previews* and enable previews for HDF5 Files.

If the thumbnailer should be uninstalled again, *make uninstall* needs to be executed in the build folder.

2.2 Windows

The Windows implementation is a VisualStudio solution.

It consists of 2 sub projects. One is the thumbnailer itself, the other one is a installer project to create an install wizard.

To install the thumbnailer, either the installer can be used (Recommended) or it can be installed manually, by executing the *regsvr32.exe* command as an admin on the dll.

2.3 OSX

The OSX implementation is a Xcode project.

When you build it, it automatically installs itself for the current user, into the directory */Users/<username>/Library/QuickLook/HDF5 Thumbnailer.qlgenerator*.

To make it available to all users of the system, copy the *HDF5 Thumbnailer.qlgenerator* folder into */Library/QuickLook/*

3 Components

3.1 Python scripts

The two Python scripts are called *thumbnailInserter.py* and *metadataReader.py* They require the *argparse* and *xmltodict* modules to be installed.

3.1.1 thumbnailInserter.py

This script takes a HDF5 file, and writes a version of it with the thumbnail and metadata. It has also the possibility of updating or delete existing data.

When working with an HDF5 file that doesn't contain any XMP block, a new one will be created. If the file already has XMP data inside the script will update the data you give it and leave the rest alone.

To work the script needs a few arguments. The first argument is the HDF5 file you want to use. If the HDF5 file doesn't already have a thumbnail, we need to provide the path to an image file via the ***–img*** argument. The format of the image can be any format understandable by the system, but generally PNG or JPG for biggest possible compatibility and good quality/fileSize.

If you want to write the data into a sidecar file, you need to provide the ***–sidecar*** argument.

When you don't provide the sidecar argument you need to specify an *outFile* where to write the file with the metadata and thumbnail. This is specified with the ***–outFile*** argument. (Or ***-o*** as a short version). If omitted when writing to a sidecar file the name of the outFile will be the name of the HDF5 file with the *hdf5* suffix replaced with *xmp*.

To provide additional data to insert into you can use the ***–data*** or ***-d*** argument. You can provide any amount of those arguments. The first word written after the argument represents the key name of the data and the second one the data to store in it.

There are few restrictions as to what can be used as key names and values. Key names can't contain any character not normally allowed in an XML tag and can't be called *Thumbnails*. Values can contain any printable character except &, < and >. If you want to use any of these they need to be escaped. *None* is also not allowed as a saved value.

To remove a key-Value pair when updating, just set the value to *None*

Here is an example of inserting an image and some data into a file.

```
./thumbnailInserter.py oldFiles/engine.hdf5 \
    -o newFiles/engine.hdf5 --img ~/Pictures/placeholder.png \
    -d Author "Bill from nextdoor" -d ApprovedBy Me
```

In this example we insert *placeholder.png* as the thumbnail, *Bill from nextdoor* as a value called *Author* and who the file got approved by.

If for example the ApprovedBy field isn't needed anymore and we want to remove it we can set its value to *None* to remove the field.

```
./thumbnailInserter.py newFiles/engine.hdf5 \
    -o newerFiles/engine.hdf5 -d ApprovedBy None
```

This removes the *ApprovedBy* field without touching the inserted thumbnail or Author.

3.1.2 metadataReader.py

This script reads the XMP data from the specified HDF5 or XMP file.

The script has the option to output the thumbnail to a file or print the metadata to stdout formatted as JSON.

```
./metadataReader.py newerFiles/engine.hdf5
```

This will output: **{"Author": "Bill from nextdoor"}**

4 Userblock storage

4.1 Overview

In this section will be explained how the XMP data is stored in the HDF5 file's userblock and how to store your own data without damaging other data stored there with this method.

4.1.1 How the userblock works

The userblock is a arbitrarily sized section that can be placed in any HDF5 file before it starts.

When reading a HDF5 file the HDF5 library checks for the HDF5 signature at the start of the file. If it doesn't find a signature then it assumes that it contains a userblock and then it checks at the *next power of 2* for the signature. It will do this until it either finds a signature or reaches the end of the file. When writing to the userblock it needs to pad the file until it reaches the next power of 2 for the HDF5 signature.

4.2 Position in the userblock

To ensure compatibility between multiple data blocks a signature to be used for the stored data blocks is specified. If the userblock has a correct signature then additional data can be inserted right before or after any other data block in the userblock.

4.3 Signature

The signature for the data blocks is an extension of the HDF5 signature with an added content length.

The HDF5 signature is made up of 8 bytes with the following content:

Decimal:	137	72	68	70	13	10	26	10
Hexadecimal:	89	48	44	46	0d	0a	1a	0a
ASCII:	\211	H	D	F	\r	\n	\032	\n

The signature for a custom data block is made up of two parts.

The first is the identifier:

Decimal:	137	72	**	**	13	10	26	10
Hexadecimal:	89	48	**	**	0d	0a	1a	0a
ASCII:	\211	H	*	*	\r	\n	\032	\n

Except for bytes 3 and 4 it is the same as the HDF5 identifier. Bytes 3 and 4 can be any value as long as it isn't used by another program, meaning they can't be the same

as the ones in the identifiers for the HDF5 and XMP metadata signature.

For example, the XMP block with the metadata and thumbnail have this signature:

Decimal:	137	72	77	80	13	10	26	10
Hexadecimal:	89	48	4D	50	0d	0a	1a	0a
ASCII:	\211	H	M	P	\r	\n	\032	\n

The **M**etadata XM**P** header is extracted by the thumbnailer, while others are skipped.

The next part of the signature is 8 bytes describing the size of the data, in **big-endian** byte order. Storing the size lets the thumbnail inserter know where it could potentially insert the XMP block and it lets the thumbnailer find the data faster

Here is an example size of 105585 bytes:

Hexadecimal:	00	00	00	00	00	01	9C	71

With the identifier, our XMP data will have this signature:

Hexadecimal:	89	48	4D	50	0D	0A	1A	0A	00	00	00	00	00	01	9C	71

When the thumbnailer reads the signatures it first reads the first 8 bytes. It then looks if it's a valid header by just ignoring byte 3 and 4. If it's valid then it checks if it's an XMP with metadata by comparing it to the HMP header.

List of Figures

Listings