TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

LASSE HEIKKILÄ

# HIGH EFFICIENCY IMAGE FILE FORMAT IMPLEMENTATION

Master of Science thesis

Examiner: Prof. Petri Ihantola
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 4th of May 2016

# ABSTRACT

**LASSE HEIKKILÄ**: High Efficiency Image File Format implementation
Tampere University of Technology
Master of Science thesis, 49 pages, 1 Appendix page
June 2016
Master's Degree Programme in Electrical Engineering Technology
Major: Embedded systems
Examiner: Prof. Petri Ihantola
Keywords: High Efficiency Image File Format, HEIF, HEVC

During recent years, methods used to encode video have been developing quickly. However, image file formats commonly used for saving still images, such as PNG and JPEG, are originating from the 1990s. Therefore it is often possible to get better image quality and smaller file sizes, when photographs are compressed with modern video compression techniques.

The High Efficiency Video Coding (HEVC) standard was finalized in 2013, and in the same year work for utilizing it for still image storage started. The resulting High Efficiency Image File Format (HEIF) standard offers a competitive image data compression ratio, and several other features such as support for image sequences and non-destructive editing.

During this thesis work, writer and reader programs for handling HEIF files were developed. Together with an HEVC encoder the writer can create HEIF compliant files. By utilizing the reader and an HEVC decoder, an HEIF player program can then present images from HEIF files without the most detailed knowledge about their low-level structure.

To make development work easier, and improve the extensibility and maintainability of the programs, code correctness and simplicity were given special attention. In addition to automatic testing also static code analysis and dynamic analysis were employed.

The developed software proved to be reliably functioning and of sufficient quality. It has been successfully used to e.g. demonstrate HEIF features in several international technical organization meetings such as MPEG.

# TIIVISTELMÄ

**LASSE HEIKKILÄ**: Korkean hyötysuhteen kuvatiedostoformaatin toteutus
Tampereen teknillinen yliopisto
Diplomityö, 49 sivua, 1 liitesivu
Kesäkuu 2016
Sähkötekniikan koulutusohjelma
Pääaine: Sulautetut järjestelmät
Tarkastajat: Prof. Petri Ihantola
Avainsanat: High efficiency image file format, HEIF, HEVC, kuvatiedostoformaatti

Videokuvan pakkaamiseen käytettävät menetelmät ovat kehittyneet viime vuosina nopeasti, mutta yksittäisten kuvien tallentamiseen yleisesti käytetyt kuvatiedostomuodot, kuten PNG ja JPEG, ovat peräisin 1990-luvulta. Valokuvien pakkaamisessa on usein mahdollista saavuttaa varsinaisia kuvatiedostomuotoja pienempi tiedoston koko ja parempi kuvanlaatu, kun pakkausmenetelmänä käytetään nykyaikaisia videokuvaa varten kehitettyjä koodausmenetelmiä.

Standardi HEVC-videonkoodausmenetelmästä julkaistiin 2013, ja samana vuonna alkoi työ sen hyödyntämiseksi myös kuvatiedostojen kanssa tarkoituksena julkaista korkean hyötysuhteen kuvatiedostomuodosta HEIF-standardi. Tehokkaan pakkauksen lisäksi se tarjoaa monipuolisia ominaisuuksia kuten useiden kuvien tallentamisen samaan tiedostoon ja mahdollisuuden muokata kuvia ennen niiden esittämistä.

Tässä diplomityössä kehitettiin standardin mukaisia tiedostoja kirjoittavaa ohjelmaa, sekä tehtiin toteutus rajapinnalle minkä avulla on mahdollista lukea HEIF-tiedostoja. Kehitystyön sujuvuuden ja ohjelmien ylläpidettävyyden takaamiseksi ohjelmakoodin selkeyteen ja virheettömyyteen kiinnitettiin erityistä huomiota. Lisäksi laadun varmistamiseksi hyödynnettiin automaattisen testauksen lisäksi laajalti sekä käännettyjen ohjelmien dynaamista analyysia että ohjelmakoodin staattista analyysia.

Kehitetty ohjelmisto osoittautui toimivaksi ja laadultaan riittäväksi. Sitä on käytetty onnistuneesti HEIF-kuvatiedostojen ominaisuuksien esittelemiseen eri yhteyksissä.

# PREFACE

This thesis was made about the first public implementation of a new image file format standard, the High Efficiency Image File Format. It has been an interesting and very educational experience. I hope this thesis has been able to provide some useful information for the project, and helps to develop it further.

I would like to thank my instructor and examiner Assistant Professor (tenure track) Petri Ihantola for his patient supportive guidance, and numerous helpful remarks and ideas.

I am truly grateful to people at Nokia Technologies for letting me to participate in the project and for allowing me to write my thesis about the topic. I want to thank especially Emre Aksu and Miska Hannuksela for their support and highly valued comments.

Furthermore, I would like to thank Juha Simola for the advice he gave and especially for pushing to get the thesis eventually done. I want to thank Masha particularly for the essential encouragement and help she gave me during this project.

Pirkkala, 23.5.2016

Lasse Heikkilä

# TABLE OF CONTENTS

# LIST OF ABBREVIATIONS AND TERMS

| | |
|---|---|
| API | Application Programming Interface |
| ASCII | American Standard Code for Information Interchange |
| AVC | Advanced Video Coding, H.264 |
| BPG | Better Portable Graphics |
| CD | Committee Draft |
| Codec | Coder-decoder |
| Decoder | An algorithm to convert compressed image to an uncompressed form |
| DIS | Draft International Standard |
| Encoder | An algorithm to convert uncompressed image to a compressed form |
| FDIS | Final Draft International Standard |
| FITS | Flexible Image Transport System |
| Framebuffer | Portion of memory holding the full bitmapped image that is sent to the display |
| GIF | Graphics Interchange Format |
| H.265 | Recommendation ITU-T H.265, HEVC |
| HEIC | HEVC Image File Format |
| HEIF | High Efficiency Image File Format |
| HEVC | High Efficiency Video Coding, H.265 |
| HTML | HyperText Markup Language |
| IEC | International Electrotechnical Commission |
| IFF | Interchange File Format |
| ISO | International Organization for Standardization |
| ISOBMFF | ISO Base Media File Format |
| ITU | International Telecommunication Union |
| ITU-T | ITU Telecommunication Standardization Sector |
| JPEG | Joint Photographic Experts Group |
| JSON | JavaScript Object Notation |
| LZW | Lempel–Ziv–Welch, a data compression algorithm |
| MP4 | MPEG-4 Part 14, a multimedia format |
| MPEG | Moving Picture Experts Group |
| PICT | A graphics file format by Apple |
| PNG | Portable Network Graphics |
| RIFF | Resource Interchange File Format |

| | |
|---|---|
| RLE | Run-length encoding |
| TGA | Truevision TGA, a raster graphics format |
| TIFF | Tagged Image File Format |
| VIDTEX | A telecommunication client software |
| VP8 | A video compression format developed by Google |
| VP8L | Lossless version of VP8 |
| WebP | An image format, currently developed by Google |
| WWW | World Wide Web |
| XML | Extensible Markup Language |
| XMP | Extensible Metadata Platform |

.

# 1. INTRODUCTION

The first known digital computer image was created in 1957 when researchers in the United States at National Bureau of Standards scanned a photograph to memory of an electronic computer [19]. The operation turned a 44 mm by 44 mm photograph into a 176 by 176 grid of black or white squares [17]. A single image of very modest quality by modern standards consumed more than half of the storage capacity of the computer [16]. Ways to compress image data to help this were not investigated until much later, because storage capacity was so expensive that storing large quantities of images was considered unrealistic. Even observing the image in the computer memory needed special arrangements. To be able to see the image without first having to print it time-consumingly, a staticizer device was connected to the computer memory and to an oscilloscope which then functioned as a display.

The processing power and storage capacity of computers was increasing quickly. In 1964, NASA (National Aeronautics and Space Administration) used computer processing to enhance the quality of images sent by a spacecraft from the Moon. In the late 1960s and early 1970s digital images were already being used in the fields of medical technology, remote sensing, and astronomy. [7]

Nowadays digital images are everywhere. The globally connected Internet has made transfer and consumption of images effortless. On the other hand, the availability of affordable electronics such as digital cameras, and mobile phones equipped with digital cameras, has made capturing and storing digital images available for masses of consumers.

Because of differences in computer system designs and implementations an image stored in a native format of some computer would most likely be beyond recognition if it was retrieved and presented by another just slightly different computer. Detailed standardized descriptions of image data storage are needed, as computer systems do not have inherent understanding about the interpretation of images or image data. Standardized image file formats are needed to make it possible to store, archive, and

interchange digital images in a convenient and reliable way.

## 1.1 Need for the research

Nowadays several different image file format standards are widely supported. However, most of these file formats date back to the 1990s or even 1980s. Some simple-seeming features, such as saving multiple images to a single file or versatile features for saving auxiliary data to the same file, are missing from several popular image file formats. [9]

The High Efficiency Image File Format standard (HEIF), developed by the Moving Picture Experts Group (MPEG) since 2013, supports a full set of features which are needed for modern digital image applications. Perhaps equally importantly, image storage space requirements can be greatly reduced by employing modern highly efficient techniques to compress image data. For instance, this can result in better perceived image quality, reduced storage costs, and faster loading times when transferring images in networks to the end users.

## 1.2 Objectives

An implementation capable of writing and reading HEIF files was needed to support standard development efforts. For example, creating complicated image files for compliance testing would be unreasonably slow and error-prone to do manually. Working reader and writer implementations may also be used for demonstrating and promoting the new file format standard.

Work carried out in this thesis consists of implementing an HEIF image file reader, and further development of an HEIF writer application. Today these created programs already form a basis for several applications and for the promotional website[1] owned by Nokia Technologies, to demonstrate HEIF features and benefits.

Additionally, an objective is to examine if it was possible to improve and maintain good code quality in an environment which required fast progress and standard drafts used as the basis for the development were still changing. Most of the time there was no possibility to organize code reviews to get continuous peer feedback,

[1] `http://nokiatech.github.io/heif/`

so continuous integration and automatic testing and analysis tools were extensively used in order to mitigate code quality deterioration. The result is examined by using several software metrics extracted from the source code version control history and by assessing changes which were done to the public source code after the first release.

## 1.3 Scope of the thesis

The thesis is about HEIF image file format and implementation of programs which write and read HEIF files. One central part of HEIF is HEVC (High Efficiency Video Coding) standard, which is employed by HEIF to compress image data to smaller storage space. However, as HEIF and aforementioned programs mostly operate on higher level, details about HEVC are mostly omitted.

The run-time performance of written software is not analyzed, as implemented components present only one part in complete image writing and reading process. The most computationally expensive operations are related to compressing and decompressing image data, which is not directly related to the file format handling itself.

## 1.4 Structure of this thesis

This thesis is structured as follows. Chapter 2 discusses digital images and related concepts. Chapter 3 describes the High Efficiency Image File Format standard history, standards it is based on, and its structure. Chapter 4 summarizes the work and how it was done. These are then evaluated and discussed in Chapter 5. Conclusions are made in Chapter 6.

# 2. DIGITAL IMAGES

The word *image* commonly refers to a two-dimensional artifact that can reproduce the visual appearance of physical objects or visualize concepts or artificial data. Digital images use numerical binary data, formed from ones and zeroes, to describe image content. The earliest digital images were created in the 1920s when telegraph printers were used to transmit images for long distances, but for computer use they arrived later in the 1950s. [19]

## 2.1 Raster and vector images

Digital images can be roughly categorized into pixel based raster images and geometry based vector images. The way image data is saved is fundamentally different, and in most use cases it is relatively straightforward to decide, which type of digital image is the better choice.

Since the 1960s vector graphics has used mathematical models of geometrical primitives, for instance polygons, circles and lines, to represent images. This makes it possible to scale an image independently from the resolution of the display medium, meaning vector images do not have a fixed native resolution for representing them. This is very desirable for example in the computer-aided design or manufacturing applications, or with graphics which needs to be scalable. An example of latter is an icon in a graphical user interface which has to adapt to different sized monitors. Figure 2.1 represents a vector image of a line. The line is described by its parameters: width, and start and end points. One way to store such information could be a text file with content "line width=1; line from 0,0 to 20,20;".

On the contrary, a raster image consists of a grid of usually square, round or rectangular dots, also called pixels, each of which is designated a color. Figure 2.2 represents a simple black-and-white raster image. Storage could be a 2-dimensional 5x5 array of bits, where 0 signals a gray square and 1 is a black square. From the

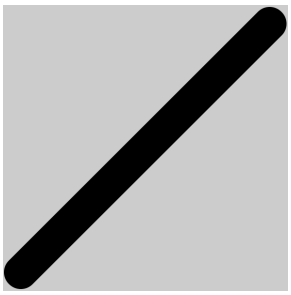

***Figure 2.1*** *An example of a vector image. A possible way to store such an image could be "line width=1; line from 0,0 to 20,20;"*

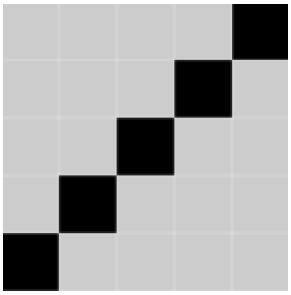

***Figure 2.2*** *An example of a raster image. Storage could be a 2-dimensional 5x5 array of bits, where 0 signals a gray square and 1 is a black square: [0 0 0 0 1; 0 0 0 1 0; 0 0 1 0 0; 0 1 0 0 0; 1 0 0 0 0].*

top-left corner to right and down this could be expressed as a matrix: [0 0 0 0 1; 0 0 0 1 0; 0 0 1 0 0; 0 1 0 0 0; 1 0 0 0 0]. A sufficient amount of dots with many enough color options close to each other can create an impression of a continuous tone image. This suits very well for representing content such as photographs. Raster images are often referred to also as bitmap images. HEIF operates with raster images only.

A raster image is characterized by its dimensions, width and height, which are expressed in pixels. Information amount needed to describe the properties of each pixel is called bit depth. Usually, each pixel has information about its intensity (gray-scale images) or color, sometimes also transparency. Bit depth normally varies from 1 bit of two-colored, often black-and-white, images to 64 bits, while a typical example is a 24-bit true color image where each of red, green and blue component has 8 bits of data. This means 16,777,216 or $2^{24}$ possible color variations.

Represented graphical objects are difficult or impossible to map back to individual pixels, making editing possibilities limited. Having said that, the described scene

complexity, the image content, does not affect image handling.

## 2.2 Compression

The way data is typically stored in a raster image is simple: a two-dimensional array holds descriptions of regular sized dots. Therefore the size of raw image data is directly dependent on the resolution (in this context width and height) the image has. High resolutions, especially combined with high bit depths, can easily result in data sizes handling of which imposes a challenge even for modern computing devices, not to mention transferring such images via mobile networks.

Size of a raw uncompressed image data can be calculated in the following way:

$size = width \times height \times bitdepth/8$

Where *size* is image data size in bytes, *height* image height in pixels, *width* image width in pixels, and *bitdepth* bits per pixel.

To mitigate issues related to excessive file sizes many raster image formats use some compression method to reduce file size. This results in savings with storage costs, faster data transmission times, and in less bandwidth needed to transfer images. For modern computer systems, compression rarely presents a significant computational overhead. Low-power embedded systems such as digital cameras can rely on hardware implementation of encoding or decoding algorithms as needed.

When a compression algorithm is lossless, it is possible to reconstruct original data perfectly. With a lossy compression algorithm some of the original data is lost irrecoverably. Lossy compression often offers a better compression ratio by relying on the inability of human senses to notice the discarded information.

Even though image formats are often classified as lossy or lossless, nothing prevents a usually lossless format encoder from manipulating the input image in a lossy manner in order to improve the compression ratio. A utility named pngquant[1], for instance, claims to reduce normally losslessly compressed PNG (Portable Network Graphics) file sizes often as much as 70%. As the extra processing happens on the encoder side this approach also retains compatibility with existing decoders.

[1] `https://pngquant.org/`

Lossless encodings used by image file formats include run-length encoding (RLE), Huffman coding, and Lempel–Ziv–Welch (LZW) coding. RLE is a simple compression method employed by several older image file formats such as PCX (PiCture eXchange), Truevision TGA (Truevision Graphics Adapter) and VIDTEX. Huffman encoding is used as a part of the widely used JPEG (Joint Photographic Experts Group) encoding in addition to lossy compression scheme. LZW is used by TIFF (Tagged Image File Format) and GIF (Graphics Interchange Format) image file formats.

JPEG compression is a lossy or lossless compression method although most implementations support only the lossy mode. JPEG works well with continuous-tone grayscale and color images, but does not suit so well for black and white images. In the lossy mode compression factor is adjustable, so the balance between image degradation and amount of data can be adjusted. [21]

## 2.3 Metadata

Metadata, data about data, is auxiliary data stored with an image which describes e.g. properties, content or characteristics of the image. Metadata can be useful for an application or database systems even if they are not able to understand the format of the actual content. Examples about metadata are work title, description, copyright status, photograph shooting location, and creation and modification dates. File systems are able to provide some further metadata such as aforementioned time stamps, but here only metadata embedded in the file itself is considered. Several standards have been created to make metadata handling easier.

Exif (Exchangeable image file format) is a widely adopted standard used by digital still cameras for saving images, sound, and metadata tags. It originates from Japan Electronics and Information Technology Industries Association. Exif uses JPEG ISO/IEC 10918-1 for saving compressed images, but uncompressed images are stored using TIFF Rev. 6.0. Also Exif metadata structure originates from TIFF. However, Exif specifies more tags, such as camera system specific private tags like focal length and aperture value. [14]

Extensible Metadata Platform (XMP) is an ISO standard 16684-1 "for the definition, creation, and processing of metadata that can be applied to a broad range of resource types." [12] The standard describes XMP data model, serialization and

core properties. The data model describes the structure of statements that XMP can make about resources. The serialization defines how the data model can be saved as XML. Core properties are items that can be used within several file formats and domains of usage. One example of XMP use cases is carrying licensing information. Embedding XMP is standardized for numerous file formats including audio, image and video formats, and HTML (HyperText Markup Language).

## 2.4 Multi-picture features

Saving several images to a single file can be desirable e.g. for storing a smaller thumbnail image along the primary image, keeping a pair of stereo images together, or for storing animations and time-lapse photography. Usually, it is desirable to still mark one image as a primary image, which represents the main image resource in the file and can act as the cover image of the file if needed.

Images might have timing data connected to them. For example, short repeating animations are a typical use case for timed image files. Recently emerged cinemagraphs[2] combine still images with some subtle movement, further dissolving border between video and still image.

## 2.5 Image file formats

The need to store digital images has caused a wide array of image file formats to be developed. For a long time hardware or application specific file formats were the only option, as hardware resources were limited and systems were not significantly interconnected. In early digital paint systems storing images for later use could have been done simply by storing frame buffer memory to disk [23]. Figure 2.3 represents this situation.

Diversity of image file formats made interchanging image files more difficult although this was not much of a problem when computer systems capable of representing graphics were rare, and used mostly for research purposes. Coupling a file format tightly to hardware or software might make implementation and execution faster, but can severely impact the interchangeability of the resulting files. Apple PICT

[2] `https://en.wikipedia.org/wiki/Cinemagraph`

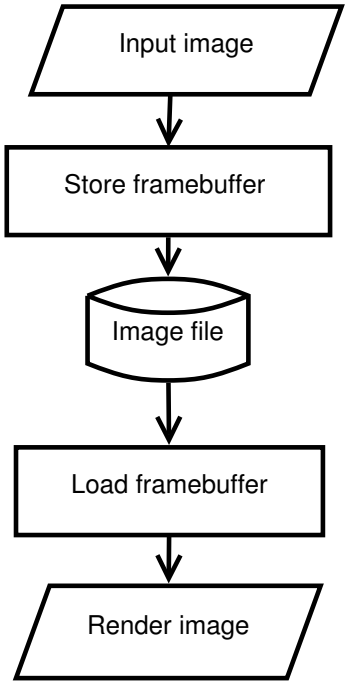


***Figure 2.3*** *A simplified framebuffer saving diagram.*

is an example of a badly interchangeable format, as it was mostly a wrapper for operating system specific QuickDraw API drawing instructions.

One early example of an image file format which was designed to be interchangeable is Flexible Image Transport System (FITS), designed for astronomy use. Development started in the late 1970s and the format was published in 1981. FITS supports multi-dimensional data arrays, which can carry basically any kind of data. The header of a FITS file is in human-readable ASCII (American Standard Code for Information Interchange) format, which enables the convenient access of metadata. [26] Use of FITS is not limited only to the field of astronomy. It is an interesting alternative to be used in the long-term preservation of images, as format versions are backward and forward compatible, FITS allows no references outside the file itself, it supports multiple images in the same file, and has no support for features such as encryption or access control. As there is a proposal for making it also an ISO standard, it could become widely used format in digitization projects by libraries and archives. [2]

As microcomputers became more common, CompuServe recognized a need to distribute images in their online service to different microcomputer platforms and in-

troduced VIDTEX in 1981. The RLE-encoded black-and-white image file format supported resolutions 128×96 and 256×192, and soon the format received support from third party applications too. [28]

One of the earliest standardized image file formats was Computer Graphics Metafile (CGM) in ISO/IEC 8632 which was published in 1987. CGM essentially wraps streamed Graphical Kernel System (GKS) operations, offering storage for vector graphics, but also for raster graphics. Many features of the format made it difficult to implement. [27]

Other early attempts to help interchangeability were made for instance by game developer Electronic Arts by documenting their generic container file format Interchange Format Files (IFF) type IFF Interleaved Bitmap (ILBM) in 1985. All IFF files consist of chunks, starting with a 4-byte ASCII type field, followed by a 4-byte length field, and then a type-dependent data. This makes extending the format possible, as readers can skip unknown chunks. Basically the same approach is used by HEIF.

## 2.6 Current raster image formats

Common raster image formats currently include GIF, PNG, TIFF and JPEG. Newer formats include WebP and BPG (Better Portable Graphics), which can be considered as rivals of HEIF because they, too, rely modern video encoding techniques to achieve good compression levels.

### Graphics Interchange Format - GIF

GIF was introduced by CompuServe in 1987 as the successor of the RLE-based image format used in VIDTEX. In 1989, GIF was updated to support animations and transparency. GIF features lossless compression and image blocks with 256 colors from the 24-bit palette. These features made GIF a good choice for the lossless storage of graphics with limited amount of colors. Lossless LZW compression enables preserving sharp edges in images. Usually, GIF is not a good option for storing photographs because of the limited number of colors available for image blocks it consists of.

However, the LZW compression method was patented, which slowed down 3rd party

support development. Last relevant patents expired in 2004, but by this the technically superior newer PNG format had already gained popularity.

### Portable Network Graphics - PNG

In 1995 CompuServe proposed the PNG as a replacement for the GIF, with intent to create a patented-free alternative for it [19]. The first PNG specification was released in 1996. Compared with GIF, PNG provided better compression, as well as offers better true color support and an optional alpha channel transparency [30]. On the 3rd of March 2004, the ISO/IEC 15948:2004 standard for PNG was published. PNG superseded GIF as the most popular lossless image format on the WWW in early 2013 [6].

### Tagged Image File Format - TIFF

TIFF files can be used for storing both photographs and graphics. It was originally created in mid-1980s to become a common image format for storing scanned images. Lossless compression support makes it possible to use TIFF files for image archiving and preservation purposes. Even though TIFF is currently public domain, its varied implementations can cause compatibility problems so that applications are able to access only files of a certain kind. [19]

### Joint Photographic Experts Group - JPEG

Abbreviation JPEG, Joint Photographic Experts Group, is often used to refer to several image formats which use a compression defined by the group. The JPEG issued the first JPEG standard in 1992 when it was also approved as ITU-T Recommendation T.81, and in 1994 as ISO/IEC 10918-1 standard [29]. Common image formats using this compression are JPEG/JFIF (JPEG File Interchange Format) and JPEG/Exif. Both formats are widely supported.

JPEG 2000 became international standard ISO/IEC 15444-1 in December 2000 [15]. It defined state-of-the-art compression techniques based on wavelet technology and a basic file format called JP2. Standard part 12 describes an ISO base media file format (ISOBMFF) based storage, with a text identical to ISO/IEC 14496-12 (MPEG-4

Part 12). Motion JPEG and Motion JPEG 2000, file formats for motion sequences, are also extended from ISOBMFF.

JPEG 2000 has notably higher computational resource requirements than JPEG [5]. The intention was to replace the original 1992 JPEG standard, but JPEG 2000 is not backwards compatible, and in early 2016 support is still missing from most WWW browsers[3] and several popular graphics applications.

JPEG XR (JPEG eXtended Range) was originally developed by Microsoft. Image coding specification standard ISO/IEC IS 29199-2 was published in 2009. The target was to keep high image quality while requiring low computational and storage resources [5]. JPEG XR is not compatible with JPEG/JFIF. In 2013, Microsoft released an open source JPEG XR library under the BSD license, but still in early 2016 only Microsoft web browsers support the format[4], making JPEG XR an incompatible option to be used in WWW pages.

### WebP

WebP image format was introduced by Google in 2010. In the beginning it used lossy intra-frame coding of the VP8 video format. Later releases added lossless VP8L compression, transparency, color profile, animation support and metadata storage. Google claims WebP images using lossless compressed are 26% smaller in file size compared to PNGs, and files with lossy compression are 25-34% smaller in size compared to JPEG images at equivalent perceived quality. As container format WebP uses Resource Interchange File Format (RIFF) which originates from IFF.

Google has released WebP format as open-source with a BSD-style license. In early 2016, some web browsers have a native WebP support (Google Chrome, Opera). Some graphics software have native support as well.

### Better Portable Graphics - BPG

BPG has been developed by programmer Fabrice Bellard since 2014. The developer states BPG is intended to replace the JPEG image format when quality or file size is

[3] `http://caniuse.com/#feat=jpeg2000`
[4] `http://caniuse.com/#feat=jpegxr`

an issue. BPG uses HEVC encoding, but it only supports a subset of the Main 4:4:4 16 Still Picture Profile, Level 8.5. Because of trade-offs for simplicity and required storage space, used bitstreams are not HEVC-compliant as such. BPG enables both lossy and lossless compression. Also features like support for animations and various metadata are present. [4]

# 3. HIGH EFFICIENCY IMAGE FILE FORMAT STANDARD

The High Efficiency Image File Format (HEIF) is a new image file format standard for storing raster images, image sequences and related metadata. The standardization effort started soon after the High Efficiency Video Coding (HEVC) standard was finalized, when it was realized that HEVC had good compression performance also with still pictures, and could be used e.g. with digital cameras if it was possible to save photographic metadata to the same file [10]. However, HEIF is not restricted to storing HEVC encoded images only, but it can contain other encoded bitstream formats as well. As HEIF is built on existing ISO Base Media File Format (ISOBMFF) and HEVC standards, these are presented briefly before proceeding to HEIF structure and features in more detail.

## 3.1 Standard development

The first version of HEVC video compression standard was finalized in January 2013 [10]. MPEG requirements documents for storing HEVC compressed still images and image sequences were ready in August 2013. Working draft ISO/IEC CD (Committee Draft) 23008-12 "High efficiency coding and media delivery in heterogeneous environments Part 12: Image File Format" was published early in 2014 San José MPEG meeting. It presented an ISOBMFF based way to store HEVC compressed single images, image collections, image sequences, and related metadata.

In July 2014 MPEG Sapporo meeting ISO/IEC DIS (Draft International Standard) 23008-12 "Carriage of Still Image and Image Sequences" was published. FDIS (Final Draft International Standard) was published in the February 2015 MPEG meeting in Geneva. This draft added to the standard derived image items, e.g. image grids and rotated images. The standard was then refined and a public review period was held from April to June 2015.

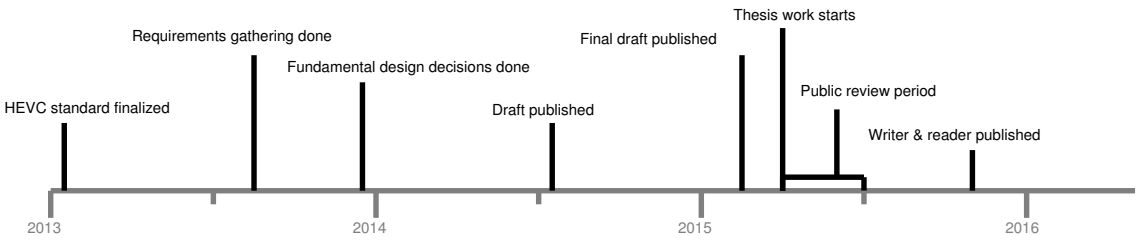


***Figure 3.1*** *Timeline of the High Efficiency Image File Format development and thesis work [10].*

The timeline of events related to thesis work and HEIF development is presented in Figure 3.1. On the time of writing this in May 2016, the editing of the final specification for publication is ongoing.

## 3.2 ISO Base Media File Format

The ISO Base Media File Format (ISOBMFF), defined in standard ISO/IEC 14496-12 - MPEG-4 Part 12, is a structural, media-independent definition for the storing time-based media such as audio and video. It also has support for untimed data. [22]

ISOBMFF is based on Apple's QuickTime file format although on a low level a rather similar structure was already present in IFF files by Electronic Arts. Among others, ISOBMFF is used by the JPEG 2000 image file format and several video formats such as MP4 (MPEG-4 Part 14), and 3GP (3GPP file format). Figure 3.2 presents ISOBMFF relationship to several file formats including HEIF.

The type of an ISOBMFF-based file is identified at the beginning of it. A file may conform to one or several so-called brand definitions. A brand defines what specifications the file follows. A single major brand indicates the best usage of the file. A file reader can then check if it supports the file, before processing it further. [22]

ISOBMFF defines box-structured files. This means that data is arranged to subsequent boxes. Boxes are allowed to contain another boxes, which may be mandatory or optional. The header of each box contains information about the type of the box, and the size of the box. This makes it possible for reader programs to skip unnecessary or unsupported boxes if needed.

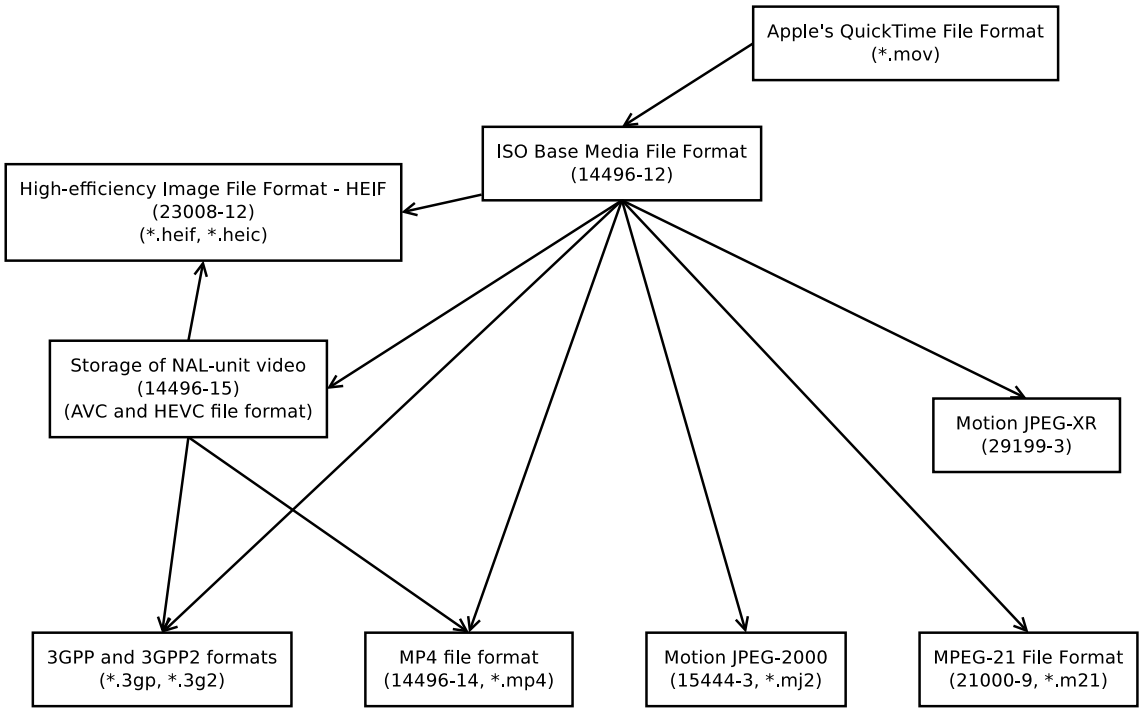


***Figure 3.2*** *Relationship between ISO Base Media File Format and various other file formats. Arrows point to the specification which is based on the other one. Adapted from [22], HEIF included by the author.*

Four-character ASCII codes are used to to identify various entities inside a file such as box types. These four-byte codes are commonly referred to as FourCC or 4CC and are marked with **bold** font in this text.

## 3.3 High Efficiency Video Coding

High Efficiency Video Coding (HEVC) video compression standard was developed by ITU-T Video Coding Experts Group (VCEG) and ISO/IEC Moving Pictures Expert Group (MPEG) in their Joint Collaborative Team on Video Coding (JCT-VC). It is also known as ITU-T Recommendation H.265. The goal was to reduce required bitrate of high-resolution videos to about half, when comparing with equivalent visual quality video encoded with Advanced Video Coding (AVC, ITU-T Recommendation H.264) standard. HEVC was the third video coding standard developed by ITU-T and ISO/IEC as a joint project. Tests indicated that HEVC met its goal, and that it performs especially well with high-resolution video. [20]

HEVC supports both intra and inter-coding of images. Decoding an intra frame

(i.e. an image) in video is possible without information about any other frames. On the other hand, decoding an inter-coded frame requires first decoding one or several other frames, as inter coding relies on information from them.

Several comparisons have been done about using HEVC intra coding for compressing still images. In subjective evaluation HEVC compression was on average 16% more efficient than JPEG 2000 4:4:4, and 43% more efficient than JPEG, while perceived quality remained similar [8]. In an objective quality test measuring peak signal-to-noise ratio, HEVC performance was superior as well [10]. According to the results, on average HEVC was 58% more efficient than JPEG, 40% more efficient than JPEG XR and 31% more efficient than JPEG 2000.

As a video coding standard, HEVC has also advanced inter-coding support. The idea of inter-coding is to use temporal redundancy between neighboring frames in order to enable a higher compression rate. Advanced motion compensation helps with determining reoccurring areas, even if their location changes. This can make the storage of image bursts and animations very efficient, as subsequent images might share a big part of their content. Tests suggest, that when comparing inter-coding with intra coding, the efficiency of inter-coding can be even tens of times better, and ratios like two to three times better can occur for natural content [10].

## 3.4 HEIF overview

HEIF is based on ISOBMFF standard and it enables the storage of image data encoded according to HEVC standard. Being based on ISOBMFF makes HEIF very flexible in comparison with several other image file formats. HEIF has been designed to store one or several images, and image sequences. An elementary difference between images and image sequences is that an image sequence can contain also timing information and that they are allowed to use HEVC inter coding. On the contrary, images and image collections have no time structure, and only intra coding is allowed.

Saving information about images and image sequences in an HEIF file is done in different ways. Still images, called image items, are stored as ISOBMFF items. Timed, and untimed but interrelated, image sequences closely follow the composition of MP4 video tracks. Difference between images and image sequence storage is a fundamental part of HEIF, so it is described briefly after presenting the high-level

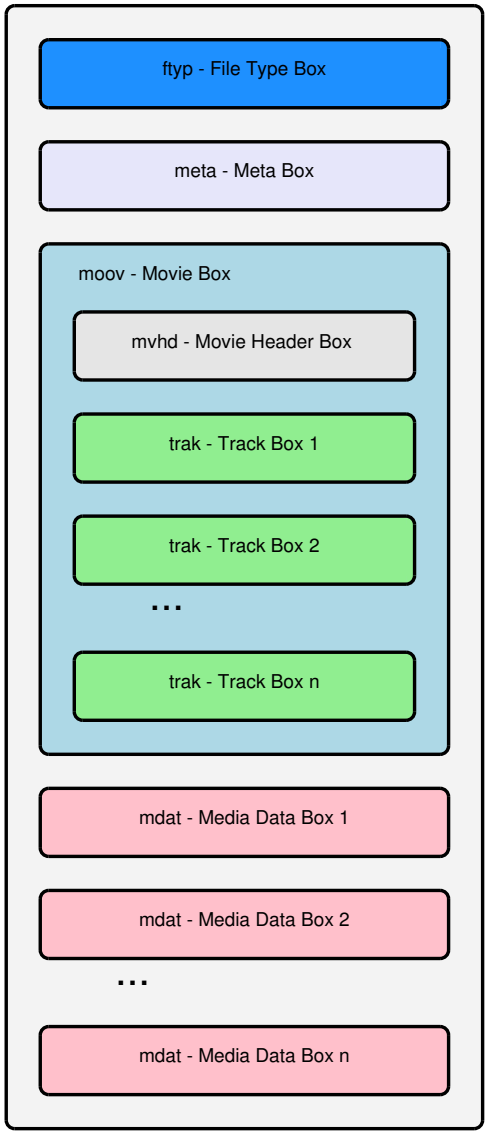


***Figure 3.3*** *A stylized presentation of an HEIF file structure when images and image sequences are present.*

structure of the file format. Finally, there are some example use cases and a feature comparison with other image file formats.

### 3.4.1 High-level structure

An HEIF file always starts with a mandatory File Type box **ftyp**. It describes the type of the file and compatibility information as brands. Information about metadata, single images and derived images are stored to a Meta box **meta**. If the

file includes image sequences, a Movie box **moov** is present. It then contains Track boxes, one for each image sequence. Figure 3.3 shows a stylized presentation of an HEIF file box structure on root-level. Also the location of image sequences, stored as Track boxes inside a Movie box, is displayed.

Both single images and image sequences can use separate Media Data **mdat** boxes for the storage of the payload image data. As many as needed Media data boxes can be included in a file. This makes it possible to refer to the same image data from both image and image sequence. However, this is not the only option allowed by ISOBMFF. For example, storing image data inside a Meta box is possible as well.

### 3.4.2 Image items

If one or several images are stored to an HEIF file, a single file root-level **meta** box will be present. It holds information about images, called image items, and possibly also the actual coded image data. In case several images are saved in an HEIF this way the result is called an image collection.

Every image item, and other items such as a derived image item and a metadata item, is assigned an individual identifier number. Information about numbered items is saved to several contained boxes inside the **meta** box. Figure 3.4 shows this simplified Meta box structure. Most central boxes are briefly described here.

Item Information box **iinf** contains entries for each item present in the **meta** box. Entries contain, for example, the human-readable symbolic names, and type identifiers of items.

Item Location box **iloc** makes finding item data possible. Primarily, it describes the locations, offsets and sizes of item data. Item data can be also fragmented into several extents.

Item Reference box **iref** enables creating directional links from an item to one or several other items. Item references are extensively used by HEIF. For instance, thumbnail images are recognized from a thumbnail type reference which links from the thumbnail image to the master image.

Handler box **hdlr** tells the metadata type. For HEIF, it is always 'pict'. Primary Item Reference **pitm** allows setting one image as the cover or primary item. Data

Information box **dinf** and **dref** contain information about where the data is located, as ISOBMFF allows it be located even outside the file.

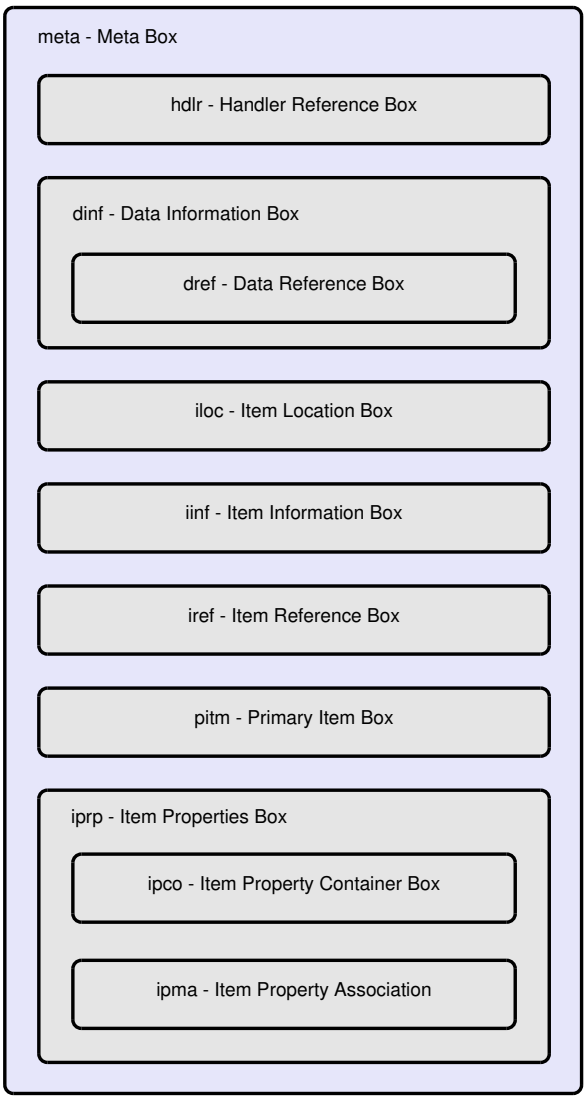


***Figure 3.4*** *Simplified structure of a Meta box in an HEIF file.*

Item Properties box **iprp** makes it possible to assign properties to image items. It contains two boxes. Item Property Container box **ipco** contains property data, which are then associated to items by **ipma** Item Property Association. Properties are classified to be either descriptive or transformative.

Examples of descriptive properties are **ispe** which describes spatial extents of an image, and **auxC** which describes the type of an auxiliary image (an alpha mask or a depth map). These provide information about the image.

An example of a transformative property is image rotation, **irot**. Transformative properties tell to a player program how the image should be modified before presenting it. This allows some non-destructive editing, like complex crop-and-rotate operations.

A versatile feature of HEIF is support for derived images. Derived images can be created either by assigning transformative properties to an image, by identity derivation, or by creating a derived image item like an image grid or image overlay from other image items. Identity derivation means that for example both the original and a rotated version of an image can be presented for the user. An image grid would then allow the creation of a third image which presents these two images next to each other.

Pre-derived coded images are images which have been derived from one or several other images. Unlike derived images, pre-derived images are simply a way to link an image to another image or images it has been derived from. Information about the type of a pre-derivation is not necessarily present in the file. An example of a pre-derivated image is an high-dynamic-range image which has been derived from several images which were photographed using different exposure times.

### 3.4.3 Image sequences

Image sequences are saved as ISOBMFF tracks, **trak** boxes. HEIF image sequences are differentiated from video tracks by a different handler type 'pict' in the Handler Box which is contained in every Track Box. Figure 3.5 shows how one **trak** box is contained in a **moov**, and the most central boxes contained by the **trak**.

In the **moov** a mandatory Movie Header box **mvhd** defines certain media-independent information which is relevant to the entire presentation. This includes data such as modification and creation time stamps and length of the presentation. Track Header box **tkhd** contains information about the individual track, such as its identifier. Track Reference box **tref** allows tracks to reference another tracks in the presentation. In HEIF image sequences, thumbnail tracks are linked to the track for full-sized images by using **tref**. Media box **mdia** contains several boxes that declare information about the media data. This includes Media Header box **mdhd** which includes information characteristic to the media in a track, and Media Information box **minf**. Further, the **minf** contains Data Information box **dinf** that declares

the location of the media information in a track, and a Sample Table box **stbl** that contains information of every single sample, here image, on the track.

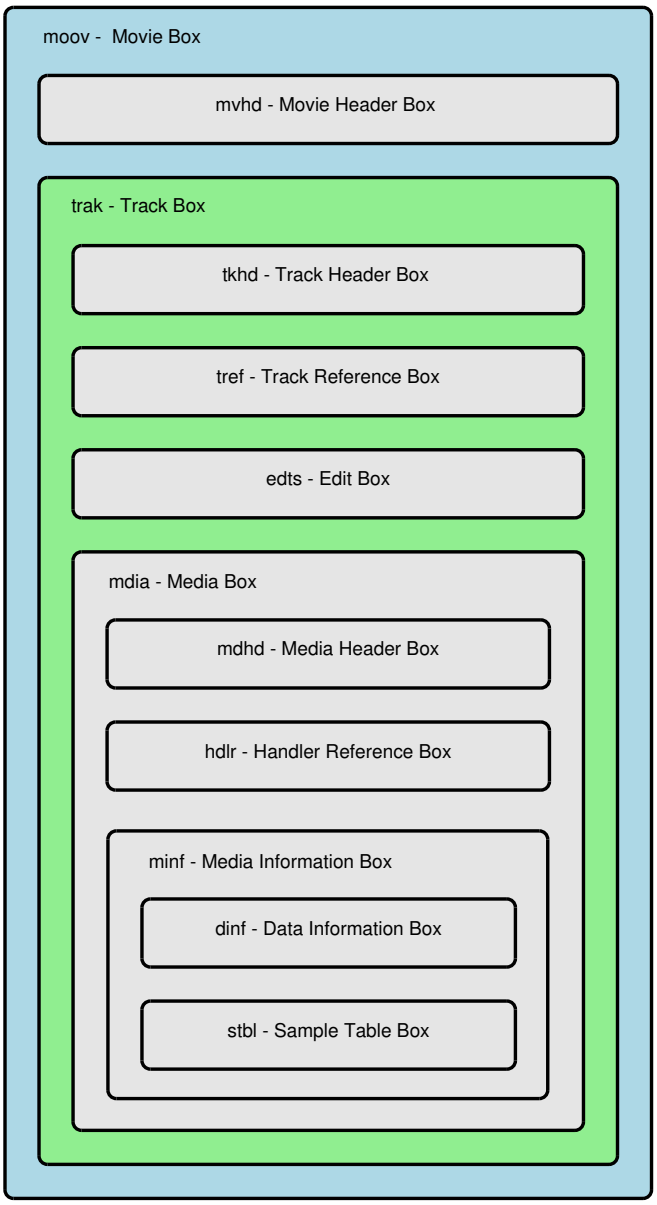


***Figure 3.5*** *Simplified structure of a Movie box in an HEIF file.*

For tracks inter prediction in HEVC bitstream is allowed. Therefore decoding an image in an image sequence may depend on the data of other images in the sequence. Usage of this kind of prediction can significantly reduce the amount of stored data. However, if wanting to view only one such image, the data of all images it is depending from must be first retrieved and decoded. This is more complicated than just

retrieving, decoding and showing only a single image, and especially in a case when dependencies are plenty this also results in inferior performance. A programmer needs to also consider the fact that image sequence decoding order is not necessarily the same as the presentation order.

Image sequences can optionally contain an Edit box **edts** with an Edit List box **elst** inside it. An edit list can be used to manipulate the presentation time-line of images. It is possible to add empty time, prolong display time of an image, or rearrange or repeat sections of several images.

### 3.4.4 Structural format and brands

In addition to specifying the encapsulation of HEVC encoded images, the HEIF specification also defines a structural format, which can be used to derive other codec-specific image formats. The name HEVC Image File Format (HEIC) can be used when referring specifically to the specified HEVC-coded image encapsulation in an HEIF file. [10]

The HEIF standard specifies several brands. These are mif1, msf1, heic, heix, hevc, and hevx. The file name extension recommended for the first two is .heif while others could use .heic. The first two are also so-called structural brands. These impose certain requirements on compliant players, but do not specify an image coding format. Others are HEVC-specific brands. [10]

The storage of image sequences in a way resembling video tracks makes it possible to create so-called dual-brand files. As a result, a dual-branded HEIF file containing an image sequence could be also played back using a regular multimedia player program, which then could interpret image sequence parts of an HEIF as regular video tracks. The image media data itself does not need to be duplicated, so the file size overhead from this is modest.

## 3.5 Use cases

Flexibility of HEIF makes it an intriguing alternative for the most common image file formats currently used in WWW and, for instance, to photography use. Table 3.1 summarizes features and characteristics of HEIF and several other image file formats.

| | HEIF | JPEG /Exif | PNG | GIF (89a) | WebP | JPEG XR / TIFF | JPEG XR / JPX | BPG |
|---|---|---|---|---|---|---|---|---|
| Formats and extensibility | | | | | | | | |
| Base container file format | ISOBMFF | TIFF | - | - | RIFF | TIFF | - [a] | - |
| Lossy compression | HEVC | JPEG | No | No | VP8 | Yes | Yes | HEVC[b] |
| Lossless compression | HEVC | TIFF Rev 6.0 | Yes | Yes [c] | VP8L | Yes | Yes | HEVC[b] |
| Extensible to other coding formats | Yes | Yes [d] | No | No | No | Yes [d] | Yes [e] | No |
| Additional metadata formats | Exif, XMP, MPEG-7 | Exif | - | - | Exif, XMP | Exif, XMP | JPX, (XMP)[f] | Exif, XMP |
| Extensible to other metadata formats | Yes | No | No | No | No | No | Yes (XML-based) | Yes |
| Other media types (audio, text, etc.) | Yes | Audio [g] | No | No | No | No | Yes[h] | No |
| Multi-picture features | | | | | | | | |
| Multiple images in the same file | Yes | No [i] | No | Yes[j] | Yes [j] | No | Yes | Yes [k] |
| Image sequences / animations | Yes | No | No | Yes | Yes | No | Yes | Yes |
| Inter coding | Yes | No | No | No | No | No | No | Yes |
| Derived images | | | | | | | | |
| Multiple-of-90-degree rotations | Yes | Yes | No | No | No | Yes | Yes | No |
| Cropping | Yes | No | No | No | No | No | Yes | No |
| Tiling/overlaying | Yes | No | No | No | Yes | No | Yes | No |
| Extensible to other editing operations | Yes | No | No | No | No | No | No | No |
| Auxiliary picture information | | | | | | | | |
| Transparency (alpha plane) | Yes | No | Yes | No [l] | Yes | Yes | Yes | Yes |
| Thumbnail image | Yes | Yes | No | No | No | Yes | Yes | Yes |

***Table 3.1*** *Summary of features of certain image file formats. Adapted from [9] with some modifications.*

[a] JPX is a box-structured format compatible with ISOBMFF. However, only the File Type box is common in JPX and ISOBMFF.

[b] HEVC Main 4:4:4 16 Still Picture profile, Level 8.5, with additional constraints

[c] Possibly lossy color quantization is applied. The color-quantized image is losslessly compressed.

[d] TIFF as a container format facilitates extensions to other coding formats.

[e] Encapsulation of JPEG-2000 and JPEG-XR have been specified for JPX container. Mappings for other codecs could be similarly specified.

[f] JPX (ITU-T T.800 and T.801) specifies an own metadata schema, but is capable of carrying an XML formatted metadata, such as XMP.

[g] PCM, $\mu$-Law PCM and ADPCM encapsulated in RIFF WAV

[h] JPX can contain media complying with ISOBMFF (or derivatives thereof). No accurate synchronization between JPX animations and other media.

[i] Can be enabled through the MP extension

[j] Only for animations and tiling/overlaying

[k] Only for animations, thumbnails, and alpha planes. Non-timed image collections not supported.

[l] A palette index for fully transparency can be specified

Lossy HEVC compression makes HEIF a competitive alternative for JPEG, as better quality images or smaller data amounts resulting in faster loading times could be achieved. Support for lossless compression and alpha channel support enable HEIF to often be a valid alternative for PNG images. The old GIF format has still remained in some use because it supports animations unlike JPEG and PNG. HEIF supports animations, along efficient encoding and true-color support.

For medical and different archive usages, some of the most valued features are often lossless compression, and flexible metadata content. Both of these are present in HEIF.

# 4. IMPLEMENTATION

This chapter describes the practices used during the development, and the architecture of the implemented HEIF reader and writer software. Section 4.1 about development practices outlines methods used for testing, continuous integration, code and program analysis, and coding practices. Architecture and design is described in Section 4.2. The main points are given about the background of the developed programs, about the current design, and how they function.

## 4.1 Development practices

Development work was carried out in an agile manner, having resemblance to several aspects of the agile manifesto[1], but no any single software development methodology such as Extreme programming (XP) or scrum was followed to the letter. For instance, the typical size of the development team would had been unusually small even for scrum [18]. However, this kind of approach is not exceptional as customization and modification of agile methods have been found in many cases [13]. One of the main priorities, matching well with the agile methodology, was to achieve a high degree of tolerance to changes because standard drafts were still evolving during the development. Tasks such as adding new features, major refactoring and applying updates from changed specifications were divided into sprints. New code was integrated into version control frequently, and builds and tests were automated.

The work described here was carried out as a part of a development team. The thesis author worked as the main software developer of the team for approximately seven months. The core responsibilities were setting up and maintaining the continuous integration server, automatic testings setup, static code analysis setup, documentation, code refactoring and quality improvements, and adding new features as defined and prioritized by the project lead. Eventually, the resulting code base contained code from seven developers although amounts of contributions varied.

[1] `http://agilemanifesto.org/`

### 4.1.1 Testing

To make fast progress possible, automated testing setup was done early in the project. This allowed developers to have confidence that changes done did not break existing functionality. A cross-platform unit testing library Google Test[2] was selected as the testing framework. Along small unit tests, Google Test can be used to perform medium sized integration tests. This enabled convenient automatic end-to-end testing, from HEIF file generation to comparing a H.265 output bitstream from the reader to the input bitstream of the writer. Figure 4.1 shows the terminal output of Google Test after successfully running a part of tests. A separate XML-formatted report was used to integrate Google Test with Jenkins.

```
[==========] Running 16 tests from 2 test cases.
[----------] Global test environment set-up.
[----------] 10 tests from Item_Based_HEVC
[ RUN      ] Item_Based_HEVC.001_Only_Master_Images_Case
[       OK ] Item_Based_HEVC.001_Only_Master_Images_Case (45 ms)
[ RUN      ] Item_Based_HEVC.002_Master_Images_and_One_Thumbnail_Per_Image_Sync_Rate_1_Case
[       OK ] Item_Based_HEVC.002_Master_Images_and_One_Thumbnail_Per_Image_Sync_Rate_1_Case (60 ms)
[ RUN      ] Item_Based_HEVC.003_Master_Images_and_One_Thumbnail_Per_Image_Sync_Rate_2_Case
[       OK ] Item_Based_HEVC.003_Master_Images_and_One_Thumbnail_Per_Image_Sync_Rate_2_Case (53 ms)
[ RUN      ] Item_Based_HEVC.004_Master_Images_and_One_Thumbnail_Per_Image_Sync_Idx_Case
[       OK ] Item_Based_HEVC.004_Master_Images_and_One_Thumbnail_Per_Image_Sync_Idx_Case (48 ms)
[ RUN      ] Item_Based_HEVC.005_Master_Images_and_Two_Thumbnail_Per_Image_Sync_Rate_1_Case
[       OK ] Item_Based_HEVC.005_Master_Images_and_Two_Thumbnail_Per_Image_Sync_Rate_1_Case (118 ms)
[ RUN      ] Item_Based_HEVC.006_Master_Images_and_Two_Thumbnail_Per_Image_Sync_Rate_2_Case
[       OK ] Item_Based_HEVC.006_Master_Images_and_Two_Thumbnail_Per_Image_Sync_Rate_2_Case (102 ms)
[ RUN      ] Item_Based_HEVC.007_Master_Images_and_Two_Thumbnail_Per_Image_Sync_Idx_Case
[       OK ] Item_Based_HEVC.007_Master_Images_and_Two_Thumbnail_Per_Image_Sync_Idx_Case (97 ms)
[ RUN      ] Item_Based_HEVC.008_Primary_Item_3_Test_Case
[       OK ] Item_Based_HEVC.008_Primary_Item_3_Test_Case (43 ms)
[ RUN      ] Item_Based_HEVC.009_1_Master_Images_and_One_Thumbnail_Per_Image_and_EXIF_Storage_Case
[       OK ] Item_Based_HEVC.009_1_Master_Images_and_One_Thumbnail_Per_Image_and_EXIF_Storage_Case (7 ms)
[ RUN      ] Item_Based_HEVC.009_2_Master_Images_and_One_Thumbnail_Per_Image_and_EXIF_Storage_Case
[       OK ] Item_Based_HEVC.009_2_Master_Images_and_One_Thumbnail_Per_Image_and_EXIF_Storage_Case (28 ms)
[----------] 10 tests from Item_Based_HEVC (601 ms total)

[----------] 6 tests from Track_Based_HEVC
[ RUN      ] Track_Based_HEVC.010_All_Intra_Only_One_Image_Sequence_Case
[       OK ] Track_Based_HEVC.010_All_Intra_Only_One_Image_Sequence_Case (42 ms)
[ RUN      ] Track_Based_HEVC.011_All_Intra_One_Image_Sequence_and_Thumbnail_Track_Sync_Rate_1_Case
[       OK ] Track_Based_HEVC.011_All_Intra_One_Image_Sequence_and_Thumbnail_Track_Sync_Rate_1_Case (46 ms)
[ RUN      ] Track_Based_HEVC.014_All_Intra_One_Image_Sequence_and_Two_Thumbnail_Track_Sync_Rate_1_Case
[       OK ] Track_Based_HEVC.014_All_Intra_One_Image_Sequence_and_Two_Thumbnail_Track_Sync_Rate_1_Case (82 ms)
[ RUN      ] Track_Based_HEVC.017_Predictive_Only_One_Image_Sequence_IPPPIPPP_Case
[       OK ] Track_Based_HEVC.017_Predictive_Only_One_Image_Sequence_IPPPIPPP_Case (14 ms)
[ RUN      ] Track_Based_HEVC.018_Predictive_Only_One_Image_Sequence_IPPPPPPP_Case
[       OK ] Track_Based_HEVC.018_Predictive_Only_One_Image_Sequence_IPPPPPPP_Case (11 ms)
[ RUN      ] Track_Based_HEVC.019_Predictive_Only_One_Image_Sequence_IBBBBBBBI_Case
[       OK ] Track_Based_HEVC.019_Predictive_Only_One_Image_Sequence_IBBBBBBBI_Case (14 ms)
[----------] 6 tests from Track_Based_HEVC (209 ms total)

[----------] Global test environment tear-down
[==========] 16 tests from 2 test cases ran. (810 ms total)
[  PASSED  ] 16 tests.
```

***Figure 4.1*** *Google Test example output.*

Having an extensive test suite also enabled the execution of extensive dynamic program analysis. This is important for good results, because dynamic analysis reveals problems only on execution paths which are actually ran.

[2] `https://github.com/google/googletest`

### Unit tests

Unit tests were used to verify functionality when developing especially complicated pieces of code. A unit test type approach was also used in cases where the writer support for certain HEIF file structures was not planned, but when the reader support was required for compatibility reason.

### Integration tests

Majority of test cases were larger integration tests, which followed a pattern of feeding a pre-made input configuration and encoded bitstream or bitstreams to the writer, loading the resulting HEIF file with the reader, and then using the reader API to verify that content corresponds to expected structure.

When reasonable, in addition to this the HEIF file was also fed to a separate reader application which then dumped H.265 bitstreams to the file system. By comparing these outputs with input bitstreams bitwise, it was made sure that integrity of decoder configuration and other data was retained.

As no time-consuming decoding or encoding was needed during tests, it was possible to create, read and analyze tens of HEIF files in a few seconds. This made it convenient to run a full set of tests as often as needed.

### Code coverage

Code coverage is a measure which tells how big part of the source code a test suite executes. Code coverage analysis was done to verify the test suite really executed code as assumed. A special build type was created for this, as this kind of analysis requires the instrumentation of built executables in a way which is not needed during normal operation or development, and prevents the usage of compiler optimization making execution slower.

Again, open-source alternatives were selected as suitable tools were readily available. The GNU Compiler Collection (GCC) includes a tool named gcov which creates reports about how many times each line of a source file has been executed. However, the plain output of gcov for a project of this size is not easy to interpret. To help with

this, a graphical front-end LCOV has been made as a part of the Linux Test Project. LCOV parses gcov output and creates a report in HTML format for easy data exploration and interpretation. These reports were used to interactively identify locations where code coverage lacking and improvements were needed. Unexpectedly missing code coverage in some location can be a strong indicator of a bug either in the test or in the code itself.

### 4.1.2 Continuous integration

Most changes to code were integrated into the version control mainline on a daily basis although few bigger features were first developed in separate branches before integrating them. Jenkins[3], an open source continuous integration tool, was set up to a server. With some help from several plug-ins and scripts, it automatically checked out changes from version control, built the source code, ran tests and generated a report about it, built documentation, and ran static code analysis. The Jenkins system was used to compile the source code by using both GCC and Clang C++ compilers. This made it possible to catch more issues, as these compilers produce different diagnostics messages. Clean builds were required also because both compilers were used during development and for final build targets. For instance, the used code coverage tool chain required GCC to be used as the compiler and Clang was needed by Emscripten build target which was used to generate JavaScript code.

The advantage of the setup was to be able to quickly catch changes which broke the build, if the testing a developer had done was not complete. Such setup also helps catching mistakes like incomplete commits to version control, i.e. if a developer did not commit all necessary modified or added files to the version control. Running automated tests and static analysis helped catching several problems in code quality.

### 4.1.3 Program analysis

Program analysis is generally classified to static and dynamic analysis. Static analysis is performed without executing the program while dynamic analysis is done to a running program. Major objectives for analysis are program correctness and optimization. In this work both were utilized.

[3] `https://jenkins.io/`

In general, static analysis tools are not precise, and might generate false positives and false negatives. On the contrary, dynamic tools are precise, but require a comprehensive set of tests to make sure all wanted code is actually executed. The number of required test cases would become huge when the application has several complex components. A combination of both static and dynamic analysis can help to detect more flaws with higher precision. [1]

### Static code analysis

Static analysis uses source code as its input without needing to compile or execute the code. This means that analysis covers also execution paths those would not be executed during automatic testing, or maybe not even during normal software operation.

A variety of static code analysis tools were employed during development. Level of interaction with them varied from near real-time continuous involvement of the checker of an integrated development environment (IDE) to a single time test of some analyzers.

C++11 version of the C++ programming language was used, which limited selection of available tools as some analyzers still support only C++03 or older standards. Only free-to-use tools were considered and tested. This excluded otherwise interesting commercial products like Coverity[4] by Synopsys and PC-Lint[5] by Gimpel Software.

Cppcheck[6] analyzes C and C++ code. The developer of it underlines that the intention is to have no false positives, and that the tool does not detect syntax errors. Findings are classified as either error, warning, style, performance, portability, or information. Especially performance and style type were considered helpful during development, as such information was not available from other sources.

The Clang Static Analyzer[7] is a part of the Clang LLVM compiler front-end project. It is a static source code analysis tool for finding bugs in programs written with C, C++, and Object-C. The Clang Static Analyzer uses several different checkers.

[4] http://www.coverity.com/
[5] http://www.gimpel.com/html/pcl.htm
[6] http://cppcheck.sourceforge.net/
[7] http://clang-analyzer.llvm.org/

For example, core checkers examine code for structures like division by zero, the addresses of stack memory escaping the function, dereferencing of null pointers, and assigning uninitialized values. This analysis was performed by the continuous integration server running Jenkins. Each build was analyzed and an HTML report generated.

OCLint[8] is a static code analyzer for inspecting C, C++ and Objective-C code. It is also based on Clang. Developers report it is able to e.g. find possible bugs, identify unused, redundant or complicated code, and recognize some bad coding practices.

### Run-time program analysis

Run-time program analysis executes a program in a controlled environment where e.g. the memory accesses of the code can be closely monitored. This can slow down program execution significantly.

Run-time analysis is also unable to catch any errors which do not reside on that particular specific execution path. On the other hand, if anomalies are found there is no chance of false positive findings, and it is guaranteed that checked type issues were not encountered during execution when using given inputs.

Valgrind[9] is a tool suite which contains several debugging and profiling tools. During development the Memcheck tool of Valgrind was extensively used to check for common C++ memory management problems. These include issues such as accessing memory which should not be accessed, using undefined values, an incorrect freeing of heap memory, making precarious memory allocations, and memory leaks.

Even though C++11 features for automatic pointer management were extensively utilized in source code, especially the usage of uninitialized variables, and memory leaks were identified during development.

Usage of uninitialized variables is able to cause undefined behavior. A symptom of this could be for example a test case failing in a random seeming manner. Therefore, instead of continuing development problems of this kind were fixed as soon as possible. Memory leaks might originate from the simple neglect of freeing reserved

[8] `http://oclint.org/`
[9] `http://valgrind.org/`

memory, or be a symptom of a design issue. Latter situation might be reasonable to fix on a better time.

#### Eclipse CDT C++ parser

Most of time Eclipse CDT IDE was used for development. The included C++ parser enables the IDE to provide features such as code navigation, search, and content assist.

This is not actual code analysis, but near real-time syntax highlighting and checking accelerates development, as code has already been "proofread" before it has been even compiled. To assist a developer even further, Eclipse also present so-called Quick Fixes for problems it has identified.

### 4.1.4 Coding practices

The programming language used in the project was C++11 version of C++. C++11 was a major upgrade over C++98/03. It has several new features for performance and convenience, such as range-for loops, automatic type deduction, lambda functions and move semantics [25]. As a result, C++11 makes it possible to write simpler code, and the C++11 standard library is easier to use, when comparing it with earlier C++ standards.

Using C++11 made use of several different development environments straightforward. The chosen CMake[10] cross-platform build management software made switching to different environments simple. Further, building C++11 code for different target environments is relatively easy. Supported targets were Linux, Windows, Android, and JavaScript environment. Latter enabled using the reader code to be embedded in WWW pages. This build was made possible by Emscripten[11] source-to-source compiler which integrated into the CMake build system.

Today programming is often team-based activity, where reading code written by others is a necessity. Easily readable code will be an advantage, at the latest, when the code will be extended and maintained. Clear writing style also tends to

[10] `https://cmake.org/`
[11] `http://kripken.github.io/emscripten-site/`

compensate some inherent complexity of code. Readability is especially important with languages like C++ which have plenty of potentially hazardous features. [24]

Programming style guidelines were adapted from Geotechnical Software Services C++ Programming Style Guidelines[12] with some modifications. These guidelines consider things relating to appearance, such as white space usage, line length and variable naming, but also to code structure (e.g. no assignment operations inside branching condition).

## 4.2 Architecture and design

Along the HEIF reader and writer, several other components are required to complete the toolchain for storing and presenting HEIF image files. Figure 4.2 shows how different components are related when saving an image to a file, and then loading and rendering it. The writer and reader described in this chapter are highlighted with a dashed line.

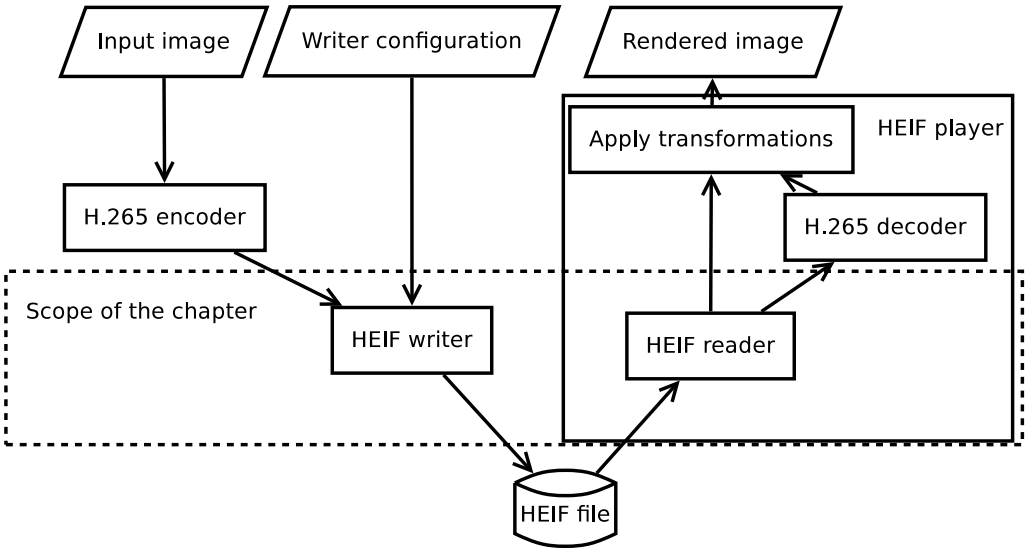


***Figure 4.2*** *A simplified diagram about components used to save and load an HEIF file.*

A separate H.265 encoder is needed for the creation of the compressed image data, which is then used as an input to the HEIF writer. Several input bitstreams like this can be present. Each bitstream can contain one or several images. In addition to these, a configuration file is required to describe the wanted structure and several

[12] `http://geosoft.no/development/cppstyle.html`

other aspects of the file to be generated. The writer then outputs a single HEIF file. Later, a player application can use the HEIF reader to acquire information about the contents of the file. Based on this information, the player can then retrieve the wanted image data and use a separate decoder component to decompress it to a presentable form. Before rendering the image or images, the player should apply transformation as possibly specified in the file. Information about them can be retrieved using the reader.

### 4.2.1 Background

Earlier prototypes of the HEIF reader and the writer were used as a starting point for development although they were completely rewritten. One limitation had been a strong separation between the handling of images and image sequences, which prevented from storing both to the same file. Also features such as Exif metadata storage and properties were missing, and the reader was implemented as an application, not as an API. Thus the usability of it was limited.

However, the low-level handling of ISOBMFF boxes had been implemented in a modular way, so it was possible to utilize a considerable amount of code. Also code for HEVC bitstream parsing was already present. Despite some modifications done to increase robustness and to optimize code, reusing these made development significantly easier.

### 4.2.2 Box handling

Both the HEIF reader and the writer employ common code for the handling of ISOBMFF boxes. On the lowest level writing and reading of ISOBMFF is handled by `BitStream` class. It provides facilities for handling different sized integers, strings, etc.

Boxes defined by ISOBMFF and HEIF are modeled by the classes of `Common` module. As a rule, box classes have at least code for saving and loading (serializing and deserializing) the box in question. Possible boxes contained by boxes are also handled here. This makes it possible to serialize or deserialize contents of even the whole root-level **meta** box with a single method call.

Several methods are offered to inserting data to box classes. The complex **meta** class

offers convenience methods for inserting items and properties, as these operations concern several nested boxes. Handling all related operations at once reduces the possibility of programming errors. Some simple box classes might offer just plain setters and getters.

### 4.2.3 File writer

Figure 4.3 presents core classes of the HEIF writer application and their relationships. The description of the file to be written is read from a JSON file by `WriterConfig` class to an internal data structure defined in `IsoMediaFile`. This arrangement makes it possible to later use the HEIF writer directly as a part of a bigger C++ program without having to create external configuration files.

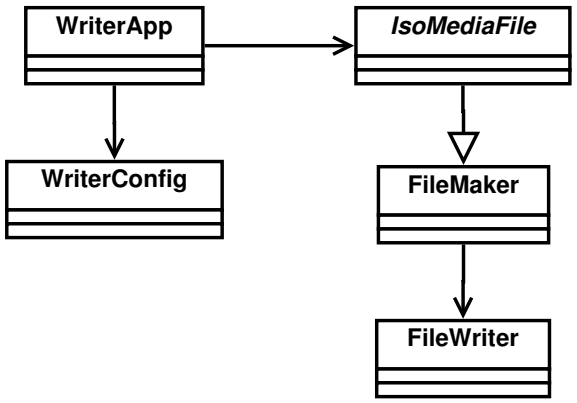


***Figure 4.3*** *Central classes of HEIF writer.*

A simple HEIF file writer description can be found in Appendix A. First the *general* section is parsed. This contains the output file name, brands, and optionally the primary item of the file is identified.

Then the actual image file content array is parsed. One content unit always includes one master section, and optionally thumbnails of different sizes, derived image descriptions, metadata and properties of images. It is possible to reference entities inside the same content rather flexibly, in order to be able to create complex derived images.

Based on the read configuration `FileWriter` creates needed writer classes. Each writer class handles only a single well-scoped part of file data generation, such as a

single thumbnail or auxiliary image section.

Classes derived from `MetaWriter` are in charge of adding content to the root-level metadata box. Classes deriving from `TrackWriter` write content to the root-level Movie Box, each of them creating a new Track box. Along TrackWriters and most MetaWriters `MediaWriter`-derived classes are created. These are responsible for writing the actual encoded payload data, or non-HEIF specific metadata which is given as input and stored as is. Figure 4.4 shows how `FileWriter` uses abstract `MetaWriter`, `TrackWriter` and `MediaWriter` base classes to fill the file content.

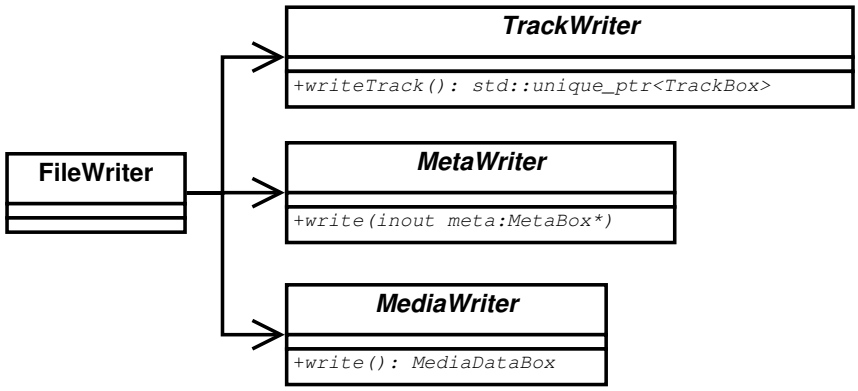


***Figure 4.4*** *FileWriter uses abstract MetaWriter, TrackWriter and MediaWriter base classes to assemble file content.*

ISOBMFF standard defines how many certain type boxes can be present in the root level, or as boxes contained by other boxes. This means an HEIF file can hold zero or one root-level Meta boxes, and zero or one root-level Movie boxes. To cope with this, different writer classes do not directly write output to a file, but add data to classes presenting the nested box structure of the file. At the end of the writing process, contents of these box classes are serialized to the output file.

`FileWriter` then handles the actual file writing in a dual pass manner. First writers participating in the creation of Meta and Movie box are called, but resulting bitstreams are discarded. Only their sizes are saved for calculating box offsets in the file. Thereafter the same writers are called again, now supplied with correct box offsets. Offsets are needed to express exact image data locations in Media data boxes which are located at the end of the file.

Meta box writers are presented in Figure 4.5. Writers derived from `RootMetaImageWriter` handle writing of encoded image items. Other classes handle various operations from

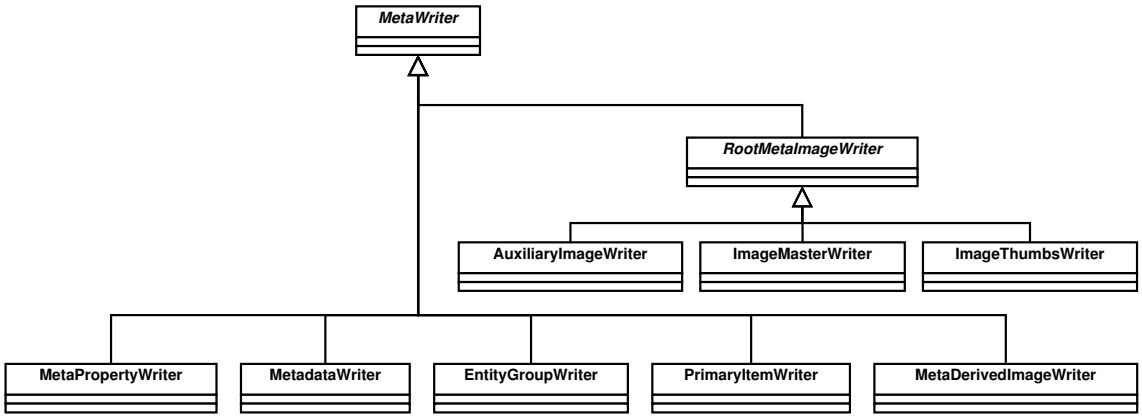


***Figure 4.5*** *Meta box writer classes.*

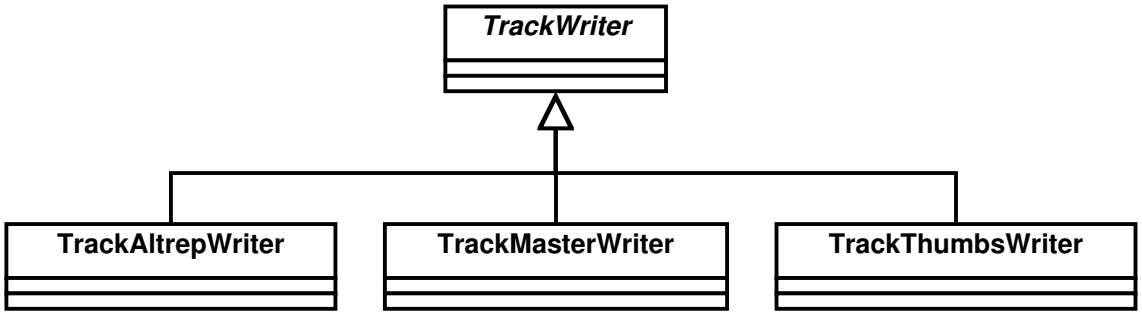


***Figure 4.6*** *Track box writer classes.*

setting up the Primary Item box of the Meta box to creating derived image items with associated properties.

Track box writers are presented in Figure 4.6. These create filled Track boxes, which can then be inserted into the root-level Movie box of the file. This structure of a **trak** box is presented in Figure 3.5. Information about images, in this context samples, is contained in several other boxes contained by the Sample Table box **stbl**. This includes e.g. details about locations, sizes, and types of every sample.

Media data box writers are presented in Figure 4.7. `ImageMediaWriter` objects are created as pairs of Track and Meta writers. Their function is basically to dump the input video bitstream without decoder configurations to a Media data box. Some derived image items such as image grid and image overlay contain data which is located in a Media data box. For those cases `MetaDerivedImageWriter` creating that data can give it to a corresponding `DerivedItemMediaWriter` for writing to Media data box.

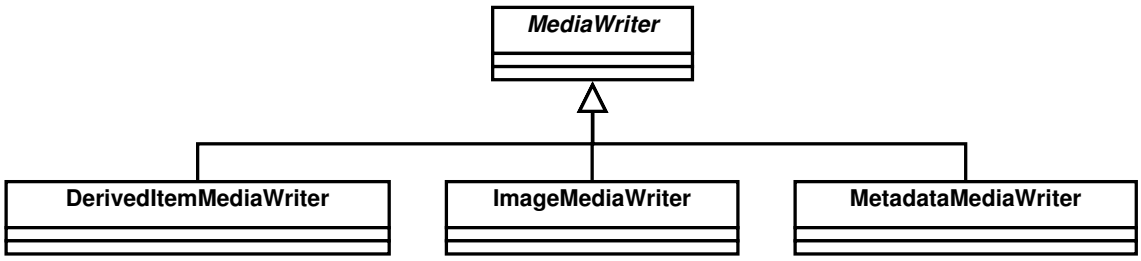


***Figure 4.7*** *Media Data box writer classes.*

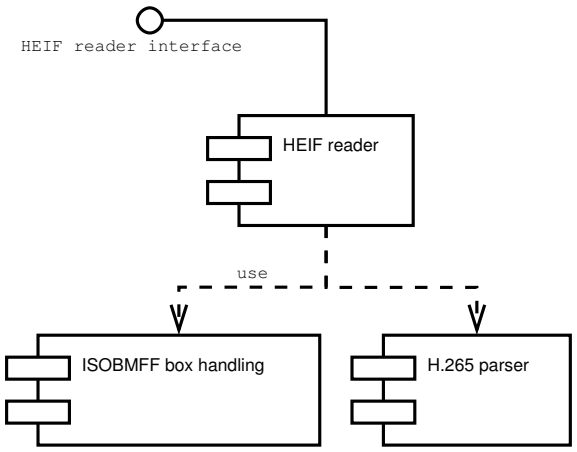


***Figure 4.8*** *Core components of the HEIF reader.*

### 4.2.4 File reader

HEIF reader was implemented as a C++ API, so it could be utilized by players and other applications. A simple reader application was written to use the API, mainly to function as a tool for writer testing. Additionally it was employed by some applications for HEIF demonstration. Figure 4.8 presents HEIF reader components in all its simplicity.

The input HEIF file is parsed box-by-box. Three types of root-level boxes are handled while others, including Media Data boxes, are ignored. Media Data box contents should be considered as binary blobs until it is given an interpretation by information from another boxes which could identify it as encoded image data, for instance.

First, the File Type box **ftyp** is checked to be present and to contain supported compatible brands.

Meta box contains information about image items and other items, the primary resource of the file, item protection scheme and item properties. If the parsing of all contained boxes is successful, the reader fills its internal and exposed reader API data structures with information about the file. This makes the serving of subsequent API calls faster and more robust, as possible later surprises caused by broken file structure can be avoided.

Movie box **moov** is handled in a manner similar to **meta**. The Movie header box **mvhd**, which contains common information about all tracks, and then individual Track boxes **trak** with sub-boxes they contain are read. Again, after successful parsing, information is extracted from the boxes to be filled to both internal and exposed reader API data structures.

A created reader instance must be first initialized by providing it either an input stream or a file name. During initialization, the reader parses the input, and fills data structures as described earlier.

After a successful initialization, a data structure describing file content can be requested from the API. The data structure contains information about possible **moov**, **trak** and **meta** boxes in the file. As an example, this information includes details about every image of image sequences, the relationships of image items, and what kind of features are present in the file (derived images, thumbnails, etc.) This helps the player to request wanted type of data using subsequent API calls.

For example, if a root-level **meta** box is present, the player could then request a list of master image item identifiers in it. A master image, as defined by the HEIF specification, is an image that is stored as an item, and that is not a thumbnail image or an auxiliary image. When master item identifiers are known, the player could then request item data and related H.265 decoder configurations, combine those and pass the result to a H.265 decoder. Before presenting decoded images on display, the player should still verify if some essential properties are mapped to the master images, and apply those as needed.

# 5. EVALUATION AND DISCUSSION

The software resulting from this work has performed without issues as a part of several internal demonstration applications and the promotional HEIF website, having successfully demonstrated the features of HEIF. It has also been used as a basis for building other features in another projects. The developed code was released to a public source code repository[1].

In Section 5.1 the work is evaluated by its code quality, reflected in certain software metrics. Analysis about fixes which have been done to the published source code since its release is done in Section 5.2. As the reader and writer constitute a certain type of an HEIF reference implementation, the standard compliance of the result is discussed in Section 5.3.

## 5.1 Code metrics

Being able to evaluate software quality is important, as inadequate quality can degrade usefulness of a software product. Software quality can be defined as the degree to which software possesses a desired combination of quality attributes [11]. Examples of quality attributes are dependability, performance, maintainability, and safety. The quality attribute maintainability was chosen for closer examination, because it gives a good general impression about how easily new features can be added, and errors can be found and corrected.

IEEE defines software metrics as "A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality." [11]

Metrics used to determine the degree of maintainability were number of lines of codes, lines of comments, code coverage, and static analysis warnings. Certain

[1] `https://github.com/nokiatech/heif`

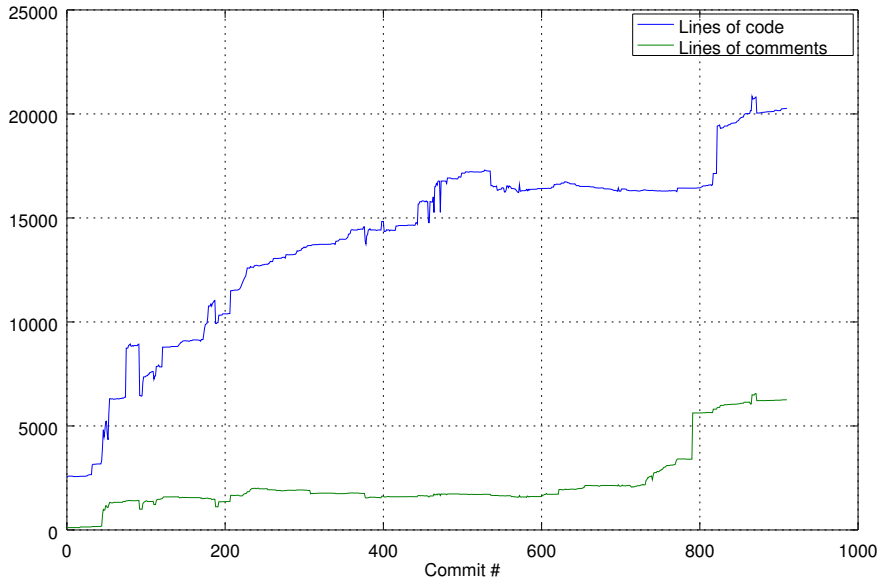


***Figure 5.1*** *Lines of code and comments.*

commits, meaning a set of changes done by a developer, in the source code version history were faulty i.e. an attempt to build the software after applying those stopped to a build system error or a compiler error. Analysis reports generated from such commits were discarded.

The presented metrics are generated from the internal source code repository main branch. The timeline is represented by generated running numbers assigned to each commit. These running numbers do not necessarily correspond to the moment when the commit in question was added to the code repository, because the project used git version control system which allows a rather flexible manipulation of history. In some points code was also developed in parallel by several developers, before being added to the common mainline.

Most analysis here concentrates to time before commit #800, as around that point the code was publicly released. Most active involvement by the thesis author also predates that point.

Code and comment lines of the project were counted using cloc[2] tool. Only C++ source files and headers were taken into account. External libraries, build system

[2] `https://github.com/AlDanial/cloc`

files and test code were omitted from inspection. The result can be seen in Figure 5.1. The code base was growing steadily before stabilizing around the commit #560. Some single refactoring was able to reduce code line number over 700 lines which presented approximately 4% of the code size at the time. Adding new features around the commit #820 caused a significant jump in the graph.

A lot of documentation effort was concentrated relatively near the first public release. This can be seen as quick growth a bit before the commit #800. The approach of doing commenting and coding in batches is recommended in order to reduce burden of constant context switching when alternating between the natural and the programming language [24]. The approach did not seem to have any significant downsides, as the code base was relatively small, and the public API was thoroughly commented since the first versions.

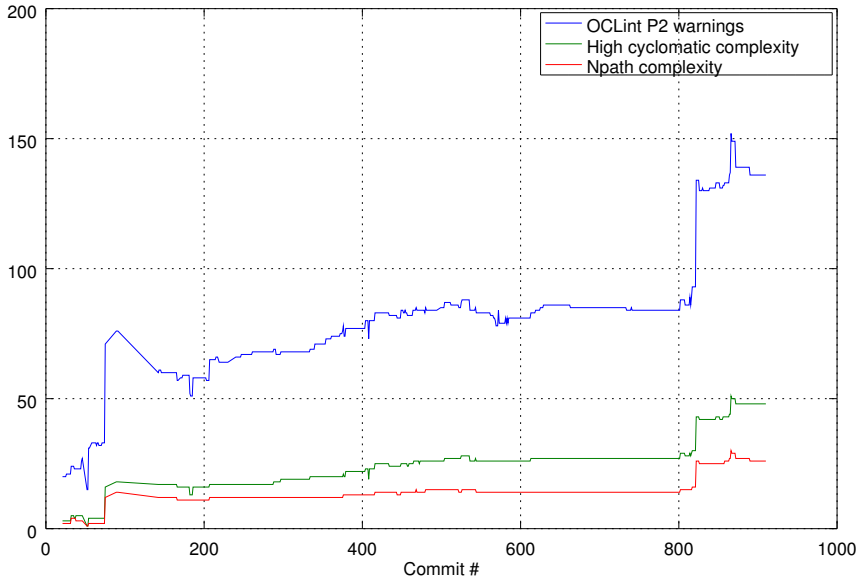


***Figure 5.2*** *Trend of OCLint static analysis violations of priority level 2. Parts of high cyclomatic complexity and NPath complexity violation messages are presented also separately.*

Figure 5.2 shows OCLint static analysis priority level 2 violations trend during the project. Priority level 2 violations were selected for closer inspection, as the most serious priority level 1 violations were not present in the project, and less severe priority 3 violations on big part are related to coding style conventions, several of which were left undefined by the project coding style documentation. Priority 2 violations

are potentially problematic findings, such as empty if or for statements, complicated methods, and bitwise operations in conditionals. These could be thought as potential problems or as structures which might harm the maintainability of the code. Having said that, in some cases these violations tell more about the nature of the code in question than are an alarm sign.

No analysis messages were suppressed or turned off in the source code or configuration. In addition to the OCLint default settings, maximum allowed violation limits were raised so that full analysis would be done always.

Also, Figure 5.2 differentiates violations about especially high cyclomatic complexity and NPath complexity in methods. Cyclomatic complexity is calculated based on decision points in a method. NPath complexity considers also nested conditional statements and boolean expressions with many parts. Both complexity metrics are also included in the priority 2 violations trend. Cyclomatic complexity metrics is especially interesting because of their influence on code testability and maintainability [3]. The figure shows that amount of complexity violations remained rather constant during the project. Most of these are originating from video bitstream parser code.

High cyclomatic complexity is characteristic of parser code, so this is not alarming. Violation suppression could be considered in order to prevent these messages from polluting logs and causing developers to ignore them completely.

The test coverage trend presented in Figure 5.3 is based on data generated by tools gcov and LCOV. The code coverage was determined by the percentage of code lines executed during a full test suite run. LCOV also outputs function coverage percentages, but one line methods like setters, getters and automatically generated functions make those numbers possibly misleading. The plot starts from the middle of version history because test setup was changing during the project. This made the afterward comprehensive collection of code coverage data unfeasible, as numerous manual modifications to early code would have been needed to successfully execute tests and code coverage analysis.

The plot shows, that the code coverage target of 80% was achieved and sustained until the addition of some experimental new code which caused a temporary drop around in #820, before corresponding test cases were added. The plot has several jumps because in most cases tests were added in batches.

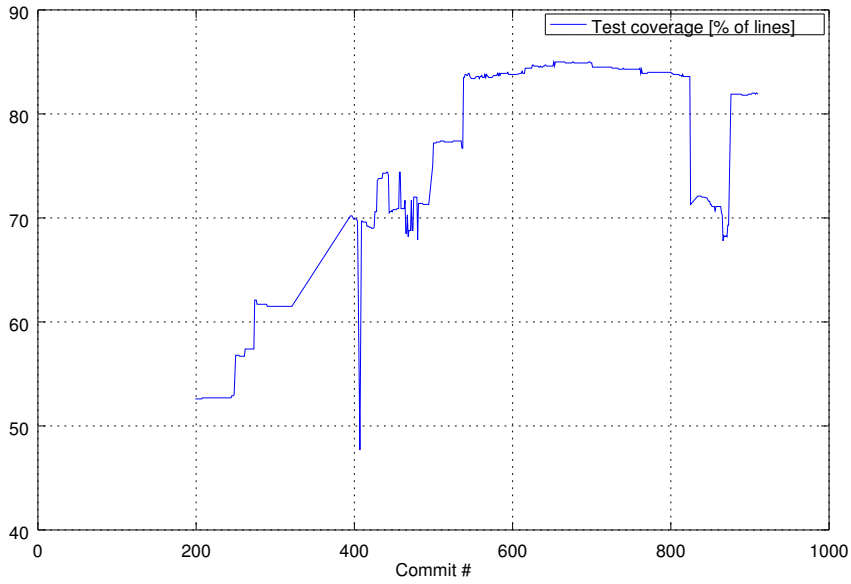


***Figure 5.3*** *Code coverage trend during the project.*

Aforementioned metrics are combined in Figure 5.4. OCLint violations here are shown as relative to number of lines of code. Greatest changes occur at the beginning of project, possibly because changes were then relatively greater and the amount of code was smaller. Perhaps surprisingly, increases in code coverage do not seem to correlate with this OCLint message frequency. Having said that, it looks like the development-stage code added around the commit #820 caused the warnings to code ratio change rather significantly simultaneously with code coverage.

Figures presented here use commit dates to form serial numbers which define the timeline. This might distort observations as sizes of commits might vary considerably, as seen as jumps in Figure 5.1. Alternatively, calendar time could have been used but that has its own problems. For instance weekends and vacation seasons would then distort graphs. The usage of the same analysis tool for metrics extraction and for irregular analysis runs during the development can also have an impact to the results.

## 5.2 Changes in code

Several changes were committed to the public source code repository after the initial release. These were analyzed and classified at Table 5.1 to program logic fixes,

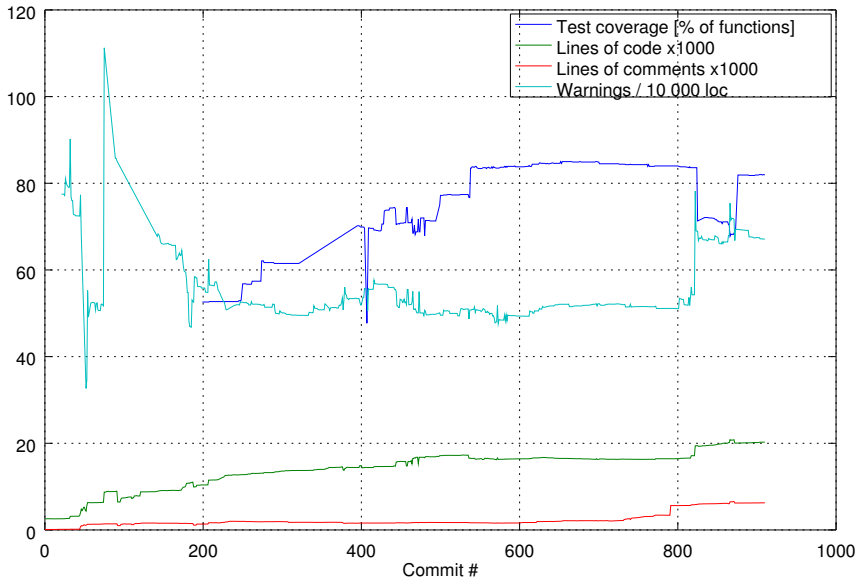


***Figure 5.4*** *Code metrics combined.*

adding new features and fixes related to the compliance with the HEIF and the ISOBMFF specifications. Every change was assigned to the single closest matching category, based on the commit message attached to the change. Changes classified here are not directly mapping to commits, because a single commit might have contained several changes.

Seven changes were identified to relate to specification compliance. In some cases the interpretation of the specification done during the development was not correct, or was incomplete. Some changes were related to changes in the standard specification. For example, in one occasion a change in the HEIF specification had remained unnoticed.

Also seven changes were related to adding new features which were missing from the first released version. These could be considered as project management decisions.

Three changes were made to fix errors in program logic. These were programming errors related to corner cases, and apparent mistakes during development.

| Change | Logic fix | New feature | Compliance fix |
|---|---|---|---|
| #1 | x | | |
| #2 | | | x |
| #3 | | | x |
| #4 | | x | |
| #5 | | x | |
| #6 | | | x |
| #7 | | x | |
| #8 | | x | |
| #9 | | x | |
| #10 | x | | |
| #11 | | | x |
| #12 | | | x |
| #13 | | x | |
| #14 | | x | |
| #15 | | | x |
| #16 | | | x |
| #17 | x | | |
| **Total** | 3 | 7 | 7 |

***Table 5.1*** *Classification of code changes done to the public repository after the initial release. Changes are ordered from the oldest to the newest when the identification of chronological order was possible.*

## 5.3 Compliance with standards

This HEIF implementation is the first public implementation of the HEIF standard. Therefore no reference implementation existed for compliance checking or benchmarking. Despite the whole HEIF standard was not implemented, the result was still complex enough to impose challenges when trying to verify if created files were exactly as intended.

The ISOBMFF and HEIF standard compliance of created HEIF files was verified by comparing the implementation with both specifications. Further, several HEIF files created by the writer were given as input to independent viewer and analysis tools. ISOBMFF compatibility was frequently checked with isoviewer[3] which interprets the contents of ISO 14496-12 and other MP4 files. Several tests were also made with on-line MP4Box.js / ISOBMFF Box Structure Viewer[4].

HEVC bitstream integrity, after storing and retrieving it from an HEIF file, was

[3] `https://github.com/sannies/isoviewer`

[4] `http://download.tsi.telecom-paristech.fr/gpac/mp4box.js/filereader.html`

checked depending on the exact situation. This was done either by comparing it bitwise with the original bitstream, or by inspecting it with hevcesbrowser[5] tool for analyzing HEVC bitstreams. The idea was to verify that any compliant HEVC decoder should be able to decode the result correctly.

Despite these procedures, several deficiencies in standard compliance were later identified. These included issues such as writing a box erroneously twice, using a wrong version of a box definition, filling entries to a box in wrong order, locating a contained box into a wrong parent box, and deriving a box from a wrong type of a box.

Reasons for several issues related to standard compliance were likely caused by the fact that the automated tests used both the self-written writer and the reader. In cases when the same mistake was present in both components no fault was triggered. Understandably, generic tools for ISOBMFF parsing did not identify HEIF-specific structures either, and were either not targeted for automated compliance testing.

Using a hex editor to compare an HEIF file byte-by-byte with extensive specifications in compliance verification purpose is not feasible. Experience seems to show that with even with the help of ISOBMFF-level tools it is hard to be sure that generated files are completely correct and comply with all specifications.

The verification process should be automated, so it could be integrated as a part of the automated tests. This could become easier also because the conformance testing of the HEIF specification will start soon. A set of files generated by the HEIF writer[6] are proposed to become conformance files to help testing other readers and writers.

Furthermore, to help mitigate mistakes from specification interpretation it could be investigated if it was possible to convert relevant parts of specifications to a machine-readable form, and use it as an input to a separate compliance validation program, or possibly use it as an input to generate one. Another approach would be to use automatically generated code already in the writer code. However, this would then reduce redundancy which could otherwise help catching inconsistencies.

[5] `https://github.com/virinext/hevcesbrowser`
[6] `https://github.com/nokiatech/heif_conformance`

# 6. CONCLUSION

In this work, the first public implementation of the High Efficiency Image File Format was created. The implementation consists of a writer application for creating HEIF files and a reader API which can be used to access the files created. A significant part of this work was setting up and maintaining facilities for continuous integration, and improving code quality. The implementation has already been successfully used to demonstrate the features of HEIF in several contexts including MPEG meetings, and for creating a set of conformance candidate files which are proposed to become official test material.

Evaluation of the work shows that it was possible to improve and maintain sufficient software quality during the project. Several code metrics were extracted from the source code repository version history. The metrics show that eventually the written test cases achieved a high code coverage of over 80%. This did not only mean that most of the code was automatically tested, but it also enabled the extensive dynamic analysis of programs. Several problems were quickly diagnosed and fixed thanks to it. When the violations found by static analysis were related to the amount of code, it can be seen that the frequency of violations decreased and stabilized before the first public release.

The combination of dynamic and static analysis seemed to supplement each other like expected, as static analysis logs were completely free of most serious class findings. In order to achieve better code quality it could have been beneficial to integrate static analysis into the continuous integration system in an earlier phase of the project, and use more extensive analysis. Static analysis messages about video bitstream parser code should be suppressed so less false alarms would be present. An automatic reporting of quality issues to developers would further enhanced the effectiveness of the analysis. Now the used process did not provide real-time feedback for developers about changes in static analysis results.

Automatic processing of the HEIF and ISOBMFF specifications on the box-structure

level could be considered, as it seems that manual checking of files by using existing tools and by reading specifications is error-prone. Basic machine-readable representation of specification structures and parsing it would probably be feasible to implement. This would make it possible to automatically analyze generated HEIF files, and enable a better decoupling of testing and the actual application code.

The HEIF is a very competitive image file format by its features and compression efficiency. However, the extensive feature set might also become a burden if supporting it turns out to be impractical, and applications with incomplete feature sets appear and cause compatibility issues which frustrate users. Having said that, the usage of ISOBMFF brands can ease this as they define a minimal player operation and therefore guarantee a minimum set of interoperability.

# BIBLIOGRAPHY

[1] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, vol. 1, Sept 2006, pp. 343–350.

[2] S. Allegrezza, "Flexible image transport system: a new standard file format for longterm preservation projects?" July 2012. [Online]. Available: https://www.vatlib.it/moduli/Allegrezza_EWASS2012.pdf

[3] H. H. Ammar, T. Nikzadeh, and J. B. Dugan, "Risk assessment of software-system specifications," *IEEE Transactions on Reliability*, vol. 50, no. 2, pp. 171–183, Jun 2001.

[4] F. Bellard, "BPG specification," 2014-2015, [Online; accessed 4-February-2016]. [Online]. Available: http://bellard.org/bpg/bpg_spec.txt

[5] F. Dufaux, G. Sullivan, and T. Ebrahimi, "The JPEG XR image coding standard [standards in a nutshell]," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 195–199, 204–204, November 2009.

[6] M. Gelbmann, "The PNG image file format is now more popular than GIF," Jan. 2013, [Online; accessed 15-May-2016]. [Online]. Available: http://w3techs.com/blog/entry/the_png_image_file_format_is_now_more_popular_than_gif

[7] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[8] P. Hanhart, M. Rerabek, P. Korshunov, and T. Ebrahimi, "Subjective evaluation of HEVC intra coding for still image compression," in *Seventh International Workshop on Video Processing and Quality Metrics for Consumer Electronics - VPQM 2013*, 2013.

[9] M. M. Hannuksela, E. B. Aksu, J. Lainema, and V. K. Malamal Vadakital, "Overview of the high efficiency image file format," 2015, [Online; accessed 4-February-2016]. [Online]. Available: http://phenix.int-evry.fr/jct/doc_end_user/documents/22_Geneva/wg11/JCTVC-V0072-v1.zip

[10] M. Hannuksela, J. Lainema, and V. Malamal Vadakital, "The high efficiency image file format standard [standards in a nutshell]," *Signal Processing Magazine, IEEE*, vol. 32, no. 4, pp. 150–156, July 2015.

[11] IEEE, "IEEE standard for a software quality metrics methodology," *IEEE Std 1061-1998*, Dec 1998.

[12] ISO, "Graphic technology — Extensible metadata platform (XMP) specification – Part 1: Data model, serialization and core properties," International Organization for Standardization, Geneva, Switzerland, ISO 16684-1:2012, 2012.

[13] S. Jalali and C. Wohlin, "Agile practices in global software engineering - a systematic map," in *2010 5th IEEE International Conference on Global Software Engineering*, Aug 2010, pp. 45–54.

[14] JEITA, "Exchangeable image file format for digital still cameras — Exif version 2.3," Japan Electronics and Information Technology Industries Association, Tokyo, Japan, JEITA CP-3451C, 2012.

[15] JPEG, "Overview of JPEG 2000," [Online; accessed 15-May-2016]. [Online]. Available: https://jpeg.org/jpeg2000/

[16] R. A. Kirsch, "SEAC and the start of image processing at the national bureau of standards," *IEEE Annals of the History of Computing*, vol. 20, no. 2, pp. 7–13, Apr 1998.

[17] R. A. Kirsch, L. Cahn, C. Ray, and G. H. Urban, "Experiments in processing pictorial information with a digital computer," in *Papers and Discussions Presented at the December 9-13, 1957, Eastern Joint Computer Conference: Computers with Deadlines to Meet*, ser. IRE-ACM-AIEE '57 (Eastern). New York, NY, USA: ACM, 1958, pp. 221–229. [Online]. Available: http://doi.acm.org/10.1145/1457720.1457763

[18] A. Mundra, S. Misra, and C. A. Dhawale, "Practical scrum-scrum team: Way to produce successful and quality software," in *Computational Science and Its Applications (ICCSA), 2013 13th International Conference on*, June 2013, pp. 119–123.

[19] F. Nack, *Encyclopedia of Database Systems*. Boston, MA: Springer US, 2009, ch. Image Metadata, pp. 1362–1368. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-39940-9_1521

[20] J. R. Ohm and G. J. Sullivan, "High efficiency video coding: the next frontier in video compression [standards in a nutshell]," *IEEE Signal Processing Magazine*, vol. 30, no. 1, pp. 152–158, Jan 2013.

[21] D. Salomon and G. Motta, *Handbook of Data Compression*. London: Springer London, 2010. [Online]. Available: http://dx.doi.org/10.1007/978-1-84882-903-9_11

[22] D. Singer, "ISO Base Media File Format," 2011, [Online; accessed 1-May-2016]. [Online]. Available: http://mpeg.chiariglione.org/standards/mpeg-4/iso-base-media-file-format

[23] A. Smith, "Digital paint systems: an anecdotal and historical overview," *Annals of the History of Computing, IEEE*, vol. 23, no. 2, pp. 4–30, Apr 2001.

[24] D. Spinellis, "Reading, writing, and code," *Queue*, vol. 1, no. 7, pp. 84–89, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/957717.957782

[25] Standard C++ Foundation, "C++11 Overview," [Online; accessed 14-May-2016]. [Online]. Available: https://isocpp.org/wiki/faq/cpp11

[26] D. C. Wells, E. W. Greisen, and R. H. Harten, "FITS - a flexible image transport system," *Astronomy and Astrophysics, Supplement*, vol. 44, p. 363, June 1981.

[27] Wikipedia, "Computer Graphics Metafile," 2014, [Online; accessed 4-February-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Computer_Graphics_Metafile&oldid=640192792

[28] ——, "VIDTEX," 2015, [Online; accessed 4-February-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=VIDTEX&oldid=694445047

[29] ——, "JPEG," 2016, [Online; accessed 4-February-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=JPEG&oldid=700084745

[30] ——, "Portable Network Graphics," 2016, [Online; accessed 4-February-2016]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Portable_Network_Graphics&oldid=698339104

# APPENDIX A. AN HEIF WRITER EXAMPLE INPUT

```
{
  "general":
  {
    "output":
    {
      "file_path" : "funny_cats.heic"
    },
    "brands":
    {
      "major" : "mif1",
      "other" : ["mif1", "heic", "hevc"]
    }
  },

  "content":
  [
    {
      "master":
      {
        "file_path" : "funny_cats.h265",
        "hdlr_type" : "pict",
        "code_type" : "hvc1",
        "encp_type" : "meta"
      }
    }
  ]
}
```

***Input configuration 1*** *An example of an HEIF file description for the writer application in JSON format. Encapsulation type meta makes this a single image item, or an image sequence in case the input bitstream contains several images.*