OGC® DOCUMENT: 24-042R1

External identifier of this OGC® document: http://www.opengis.net/doc/PER/t20-D014

Open Geospatial Consortium

OGC TESTBED-20 GIMI OPEN SOURCE REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2025-02-14
Approval Date: 2025-06-12
Publication Date: 2025-MM-DD
Editor: Sina Taghavikish

Note

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. The Open Geospatial Consortium shall not be held responsible for identifying any or all such patent rights.

Recipients of this document are requested to submit, with their comments, notification of any relevant patent claims or other intellectual property rights of which they may be aware that might be infringed by any implementation of the standard set forth in this document, and to provide supporting documentation.

CONTENTS

I. OVERVIEW v

II. EXECUTIVE SUMMARY v

III. KEYWORDS vi

IV. CONTRIBUTORS vi

V. FUTURE OUTLOOK vii

VI. VALUE PROPOSITION vii

1. INTRODUCTION 2
   1.1. Aims 3
   1.2. Objectives 3

2. TOPICS 5
   2.1. Dirk Farin Algorithmic Research e K 5
   2.2. Geomatys — Enhancing GeoHEIF Support for Apache SIS 7
   2.3. Ecere — Contribution 9
   2.4. Silvereye Technology — Impact on libheif, GDAL, and GPAC 9

3. OUTLOOK 12

4. SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS 15

BIBLIOGRAPHY 17

ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS 19

ANNEX B (INFORMATIVE) DIRK FARIN ALGORITHMIC RESEARCH E.K. — CONTRIBUTIONS TO LIBHEIF 22
   B.1. Introduction 22
   B.2. Image Tiling 22
   B.3. Multiresolution Pyramids 29
   B.4. Network Streaming Interface 30
   B.5. Encoding Tiled Images with heif-enc 32
   B.6. Example Tiled Image Viewer 34
   B.7. Data Types Support 35
   B.8. Reading and Writing Sequences 36

B.9. Libheif Command Line Tool Support for Image Sequences 42

ANNEX C (INFORMATIVE) ECERE — CONTRIBUTIONS 46

ANNEX D (INFORMATIVE) GEOMATYS — OPEN SOURCE CONTRIBUTIONS 48
D.1. Java implementation 48
D.2. Binding to GDAL 52
D.3. Differences between GDAL and Apache SIS 53

ANNEX E (INFORMATIVE) SILVEREYE TECHNOLOGY — OPEN SOURCE CONTRIBUTIONS 57
E.1. Executive Summary 57
E.2. libheif contributions 57
E.3. GDAL contributions 58
E.4. libavif contributions 59
E.5. GPAC contributions 60
E.6. GStreamer contributions 60

LIST OF TABLES

Table B.1 — Table Encoding formats. 32

LIST OF FIGURES

Figure B.1 — Tiled image with ignored right and bottom border. 27
Figure B.2 — Image has been rotated by 90 degrees. Now the left_offset is greater than 0. 28
Figure B.3 — Interaction between libheif and client application when decoding image tiles on-demand for a streaming file source. 31
Figure B.4 — Tiled HEIF image viewer example application. 35
Figure D.1 — Visualizing the ranges of bytes read from a file 50

I OVERVIEW

In OGC Testbed 20, enhancements were made to libheif to expand its capabilities for the GEOINT Imagery Media for ISR (GIMI) profile. This OGC Report presents the newly introduced `tili` tiling schemes alongside the original `grid` High Efficiency Image File Format (HEIF) tiling method and the 'unci' tiling method defined in the ISO 23001-17:2024 MPEG systems technologies uncompressed codec Further, the Report outlines the API for writing and reading these tiling methods using the libheif library. Additionally, support for multiresolution pyramids in libheif is documented, including decoding and encoding aspects for these structures. The network streaming interface of HEIF/GIMI is also addressed in this Report. Moreover, the encoder and viewer for these new tiling methods are discussed, alongside an account of the extended support for various data types.

As part of the extension work in Testbed 20, HEIF now also supports image sequences. The specification is based on the ISO14496-12 ISOBMFF Standard, incorporating some extensions from ISO23008-12 (HEIF) and GIMI-specific definitions from NGA.STND.0076_1.0.

Furthermore, the document reviews HEIF support for significant open source projects, particularly GDAL and Apache SIS. This Report elaborates on the distinctions between GDAL and Apache SIS, focusing on aspects such as georeferencing, zoom levels, and Coordinate Reference System (CRS) axis order. Lastly, contributions to other open source initiatives, including libavif and GPAC(an open source multimedia project), are briefly summarized.

II EXECUTIVE SUMMARY

This OGC Testbed 20 GIMI Open-Source Report documents the results of the OGC Testbed 20 work performed to evaluate and enhance open source libraries and tools in support of the GEOINT Imagery Media for ISR (GIMI) format.

The GIMI format has the potential to grow and gain popularity by building upon existing open source software. Recent integration work, particularly with libheif, aims to align with the GIMI profile and enhance architectural robustness. The libheif library provides the capability to decode and generate all conformant still-image HEIF/AVIF files, including High Dynamic Range (HDR), and uses the color transformation matrices specified in the color profile. The High-Efficiency Image File Format (HEIF) is a digital container format for storing individual digital images and image sequences.

The libheif library provides a robust implementation of the HEIF format employing various image and video codecs, including High Efficient Video Coding (HEVC), AOMedia Video 1 (AV1), JPEG-2000, and more, while offering region annotations and low-level interface capabilities for customized image properties.

The work performed in the OGC Testbed 20 GIMI initiative has significantly enhanced libheif's functionalities, further supporting the GIMI profile and its architecture, thus benefiting the

broader HEIF user community. This effort included improvements for HEIF file support in prominent projects such as GDAL and the Apache Spatial Information System (SIS). Two modules were developed within Apache SIS for handling a subset of the HEIF file format: A pure-Java GeoHEIF reader, currently in the incubator stage, and a native C/C++ GDAL binding, set to be included in the upcoming Apache SIS 1.5 release. Both modules enable tiling capabilities.

Key features emphasized include multi-resolution pyramids, tool extensions, network streaming, and the introduction of new data types. A novel tiling strategy, along with detailed API specifications for tiled images, has emerged from the Testbed 20 GIMI work.

The Testbed 20 Extension introduced HEIF image sequence support to libheif, particularly for intra-coded sequences in the JPEG2000, HTJ2K, and ISO23001-17 uncompressed formats, including H.264 and H.265. However, the implementation also covers H.264, H.265, H.266, AV1, and JPEG (only decoding for H.264). Each sample can include sample auxiliary information (SAI), with libheif supporting International Atomic Time (TAI) timestamps and textual content IDs from the GIMI standard. Additionally, metadata can be attached to entire tracks in libheif, currently including TAI timestamp configuration blocks and GIMI track content IDs.

The Testbed 20 Extension also added HEIF image sequence encoding support to GStreamer.

For a comprehensive account of enhancements and bug fixes, stakeholders are encouraged to visit the libheif repository (http://libheif.org). However, this Report lists a few very interesting enhancements.

III KEYWORDS

The following are keywords to be used by search engines and document catalogues.

Apache SIS, GIMI, GDAL, GPAC, image sequences, , libavif, libhief, open source, testbed, tiling

IV CONTRIBUTORS

All questions regarding this document should be directed to the editor or the contributors:

NAME	ORGANIZATION	ROLE
Sina Taghavikish	OGC	Editor
Dirk Farin	Dirk Farin Algorithmic Research e K	Contributor

NAME	ORGANIZATION	ROLE
Patrick Dion	Ecere	Contributor
Jérôme Jacovella-St-Louis	Ecere	Contributor
Martin Desruisseaux	Geomatys	Contributor
Brad Hards	Silvereye Technology	Contributor
Carl Reed	OGC	Contributor

V FUTURE OUTLOOK

Preliminary discussions about cloud-optimized GeoHEIF are addressed in the GIMI Coverage Format Selection Report [OGC 24-039]. Future open source efforts could focus on developing libraries for cloud-optimized formats and creating GIMI Compliance Test Tools to ensure format compatibility for streaming. Additionally, enhancing 'tili' to include tiled video sequences and multi-dimensional image applications is suggested, along with developing a viewer for 3-D voxel images and adding a network streaming API to libheif to support interactive image display. Also, motion-predicted video sequences are not supported yet. A broader range of data types will also be a focus for future work.

The GeoHEIF reader in Apache SIS currently supports limited data structures defined by ISO/IEC standards, specifically those used in Testbed 20, prompting the need for broader support.

VI VALUE PROPOSITION

This GIMI Open Source Report details the important recommended changes to GIMI to support the Testbed 20 activities, including a new tiling scheme and support for image sequences while keeping the metadata intact. A central objective for GIMI is the integration of an uncompressed codec into HEIF. As defined in ISO/IEC 23001-17:2024, this codec can decode high bit depth signed and unsigned integers, floating-point, and complex values, along with other data types closer to the sensor's outputs. This encompasses filter array outputs—often known as "raw" or Bayer images—sensor non-uniformity corrections, and bad pixel masks. During Testbed 20, the earlier libheif implementation was enhanced to improve clarity, efficiency, and flexibility. Before Testbed 20, GDAL facilitated access to HEIF files through a raster driver based on libheif. Although this method worked, it was restricted to read-only access, necessitating the decoding of the entire file to access any image segment. The GDAL implementation was improved to allow tiled access to HEIF files, utilizing various tiling formats, including gridded items, codec-

internal tiling similar to that used by the uncompressed codec, and the low-overhead “tili” format. This tiled access considerably enhances the efficiency of retrieving segments of large images, minimizing the data transferred and reducing the decoding requirements, which leads to lower latency for consumers.

The Testbed 20 libheif library activities facilitated the seamless management of both image sequences and still images, offering support for various encoded bitstreams, including H.264 (Advanced Video Coding, or AVC), H.265 (High Efficiency Video Coding, or HEVC), and AV1 (AOMedia Video 1) codecs, while providing support for the inclusion of metadata and TAI time stamps.

During Testbed 20, it became apparent that libavif lacked the necessary API to implement the GeoHEIF design. While libavif can parse the required properties, it discards them instead. This limitation was reported to the libavif project, prompting a response from Google and some members of the libavif core team.

This Testbed Report also highlights the differences between GDAL and Apache SIS, especially regarding their management of data extent, zoom levels, georeferencing, transformation types, the “cell center/corner” convention, and CRS axis order, offering valuable insights for more technical users. For example, while different formats may use different “cell center/corner” conventions either by format specification (as in GeoHEIF) or with a metadata set to a value chosen by the data producer (as in GeoTIFF), libraries often convert the various conventions to a canonical one. The chosen canonical convention varies depending on the library.

1 INTRODUCTION

1 INTRODUCTION

The libheif library and associated API are a robust implementation of the High Efficiency Image File (HEIF) format. This format is recognized for its efficiency in storing and managing images, making it particularly useful in various photo applications. The libheif encompasses not just the commonly used HEIC and AVIF formats, but also supports a variety of other image codecs, such as JPEG-2000 and AVC, among others. Please note that from here on in this report, the libheif library, tools, and API will be referred to as libheif.

Recent activities, specifically the Testbed 20 Geospatial Intelligence Media for Intelligence, Surveillance and Reconnaissance (GIMI) task, provided an excellent opportunity to enhance libheif, particularly in supporting the GIMI profile and improving overall usability. Collaborations have played a crucial role in implementing new features, contributing to the software's evolution and performance.

Overall, libheif represents a significant advance in image storage and manipulation, making it a valuable tool for businesses seeking efficient image management solutions.

The rise of digital imaging formats has necessitated the development of efficient tools for reading and processing these formats. GeoHEIF, an image format designed for geospatial data, is becoming increasingly relevant for use in various applications. To address the demand for effective management of GeoHEIF files, participants of Testbed 20 contributed to the open-source Apache SIS project. This project includes two modules for handling GeoHEIF files: One is a pure-Java reader specialized for GeoHEIF, and the other is a binding to the widely used GDAL native library. As a side benefit of the Testbed 20 work, the latter provides access to other formats supported by GDAL.

The Java implementation, although still in development, is capable of reading GeoHEIF files. The GDAL binding is targeted for release in Apache SIS version 1.5. Both implementations support essential image exploitation features such as tiling, which enhances performance when working with large image datasets.

Active contributions were made by Testbed 20 participants to various open source projects to enhance their functionality and support innovative applications. As part of ongoing development activities in Testbed 20, the focus was on projects critical to image and data processing. Efforts include contributions to libheif, a key library for handling modern image formats, GDAL, a versatile tool for managing geospatial data, and GPAC which offers a suite of media processing tools.

These contributions not only bolster technical capabilities but also help advance the broader image processing ecosystem. This makes implementations of the next-generation imaging framework, known as GIMI, simpler and more efficient. With Testbed participants working collaboratively with contributors or maintainers of open-source projects, the aim is to improve usability and ensure that developments meet the needs of various applications in the industry.

1.1. Aims

To facilitate the broader adoption of GIMI, maximizing the availability of open source support for GeoHEIF was recommended. This Testbed Report documents the advancements in open source initiatives directed towards enhancing the open source support for GIMI through enhancements to libheif.

1.2. Objectives

The objectives of the Testbed 20 GIMI Task were to enhance open source support for GIMI through:

- Focusing on extending and improving the capabilities of libheif for both the GIMI profile and indirectly for other users;
- Supporting HEIF/GIMI in GDAL;
- Supporting HEIF/GIMI in GPAC; and
- Supporting HEIF/GIMI in Apache SIS.

2 TOPICS

2 TOPICS

2.1. Dirk Farin Algorithmic Research e K

Dirk Farin Algorithmic Research is the primary author and maintainer of the “libheif” open source project (http://libheif.org). libheif is a comprehensive implementation of the HEIF (High Efficiency Image File) format, as defined in ISO/IEC 23008-12:2022. It supports not only the common HEIC and AVIF variants used in typical photo applications, but also includes JPEG-2000, uncompressed images (ISO/IEC 23001-17:2024), VVC (Versatile Video Coding), JPEG, and AVC codecs. In addition, libheif enables region annotations and provides a low-level interface for custom image properties, such as those defined in GeoHEIF.

libheif is widely adopted in various software applications and there are pre-built packages for most major Linux distributions, for Windows (via VCPKG and MSYS) and macOS (via Homebrew). Although libheif is implemented in C+&#43, the public API is designed in plain C to facilitate integration with other languages. Language bindings are available for many popular languages, including Python, Rust, JavaScript, Go, and C#. libheif can also be compiled to JavaScript and WebAssembly, enabling client-side image decoding directly in web browsers.

The OGC Testbed 20 activity provided an excellent opportunity to intensify the work on libheif and bring it a big step forward in supporting the GIMI profile while also improving its overall architecture. This will be beneficial to the general user base.

Dirk Farin Algorithmic Research wants to express thanks to Silvereye Technology for the valuable contributions, especially for the ISO/IEC 23001-17:2024 decoder implementation, but also for improvements in various other parts and for the integration work to other software libraries, like GDAL (see Annex E).

2.1.1. Tiling

A major focus of the Testbed 20 GIMI Task was work supporting very high-resolution images. Previously, images were always processed in one piece. This was not only computationally expensive, but it was also limited by the amount of memory available. Images too large to fit into memory could not be processed. This issue can be resolved by adding tile-based access to images. Individual tiles of an image can then be decoded independently, and a high-resolution image can be encoded by successively adding individual tiles to the file.

The design and subsequent implementation started with work on the HEIF grid image type, which is the current standard way to encode tiled images. The Testbed participants observed that several limitations of the HEIF file format restricted the number of tiles to fewer than 65536, which might not be enough for many geospatial imaging applications. Further, the participants discovered that the `grid` image type requires a huge overhead in the file header. This is particularly disadvantageous when a large image should be streamed on-demand over a

network connection, where a large image header (in the order of several megabytes) must be transferred before any image data can be decoded.

To alleviate this limitation, a new image tiling format labeled `tili` was developed. This format requires very little metadata in the file header (less than 100 bytes) and can hold images of practically unlimited size. tili was also designed for streaming the image over a network by holding a table of file extents required for each tile. This table can be loaded on-demand while viewing the image interactively. This approach results in fast loading times.

Details about the `tili` format can be found in Annex B in the Testbed 20 GIMI Lessons Learned and Best Practices Report [OGC 24-040].

The `tili` image type is currently being proposed to MPEG for inclusion in the ISO/IEC 23008-12:2022 standard. The proposal document m70021:Tili proposal was written in cooperation with Joe Stufflebeam of TRAX International Corporation.

Finally, support for the tiling feature of the “uncompressed” ISO/IEC 23001-17:2024 image type was added to libheif. Also the lossless compression methods (Deflate, Zlib, Brotli) were added as options when encoding ISO/IEC 23001-17:2024 images.

These three tiling methods (’grid’, ‘unci’, ’tili’) are transparent to a client application that wants to decode an image. libheif handles all these formats internally via the same public API. For encoding, there are three different functions to initialize images for the three tiling formats, but the API for adding images is also the same for all formats.

2.1.2. Multi-Resolution Pyramid

High-resolution images should often be viewed as downscaled overview images. To reduce the amount of decoding that is required to show a zoomed-out overview, storing downscaled overview images in the HEIF file is possible. A proposal for this is the ‘pymd’ image pyramid entity group, that is currently in ISO/IEC 23008-12:2022 draft state. This draft standard was implemented for both decoding and encoding.

That the libheif implementation of the image pyramid supports using different codecs for each pyramid layer is worth mentioning. This enables using the ‘uncompressed’ codec for the original, high-resolution data, but then able to use lossy compression like H.265 for the overview images. Another advantage is that some applications which might have no support for decoding ISO/IEC 23001-17:2024 images will still be able to show at least the overview images.

2.1.3. Tool Extensions

The `heif-enc` and `heif-dec` command line tools are applications contained in the libheif repository. These commands provide an easy way to encode and decode TIFF, JPEG, or PNG files to HEIF. The commands are extended with functions to encode tiled images from a collection of input images (one image per tile) and to define an image pyramid. See Section Annex B.5 for a detailed description of this function. The `heif-dec` command is also extended with functions to split a tiled HEIF image into single tile images in TIFF, JPEG, or PNG format.

2.1.4. Network Streaming

When streaming a high-resolution image on-demand over a network, the client application needs to know which file ranges it should download from the server. This is so that only the bandwidth for those image parts that are currently displayed is used. libheif already provided for a mechanism for writing custom file readers to provide the input data to libheif. However, the existing libheif API only supported the case where a file is read sequentially from start to end. With tile-based decoding, the requirements changed such that libheif needs to request file ranges at an arbitrary location in the file. Moreover, file-range requests having a reasonable size are desired: Typical requests should include several kilobytes of data instead of just a few bytes. This was achieved by extending the libheif file reader API with functions to request arbitrary file ranges and to control the caching of data in the client application. More information can be found in Annex B.4.1.

2.1.5. Data-Types Support

Previous versions of libheif were limited to working on integer pixel types (1-16 bits per pixel). Since ISO/IEC 23001-17:2024 also defines higher integer bit depths, floating-point, and complex number data types, libheif was extended to support these data types.

2.1.6. Other Contributions

In addition to the above changes, the internal architecture of libheif was refactored. The goal was to clearly separate the HEIF file class, the image item specific code, and the image codecs into clearly separate classes. This will become important during the upcoming work on image sequences, where the hope is to be able to reuse most of the image codecs by packaging the compressed data into sequence tracks instead of assigning the data to image items.

Another contribution is the addition of decoding support for AVC (H.264) coded images. While this is seldom used for images, it will be advantageous to have for the future image sequences implementation.

2.2. Geomatys — Enhancing GeoHEIF Support for Apache SIS

2.2.1. Geomatys Contributions to Apache SIS

Geomatys provide contributions to the Apache SIS project by developing modules for handling GeoHEIF files. Two modules were created: A pure-Java GeoHEIF reader in the “incubator” group and a GDAL-based binding in the “optional” group. The Java implementation is self-contained

and adheres to Apache SIS's design principles. The GDAL-based module relies on the C/C+&#43 GDAL library, which uses libheif, introducing a layered architecture. While the GDAL binding is scheduled for release in Apache SIS 1.5, the Java implementation, still in the prototype stage, is already functional with tiling support but requires polishing.

2.2.2. Unified Design Philosophy in Apache SIS

The Java implementation embodies Apache SIS's philosophy of maximizing code reuse and abstracting common functionalities. GeoHEIF and GeoTIFF readers share a unified foundation through common base classes, reducing development effort by focusing on format-specific nuances such as metadata handling. This differs from GDAL's modular approach, which integrates independent libraries such as `libtiff` and `libpng` through a unified API.

2.2.3. Reading Strategies in Apache SIS

Apache SIS supports two primary strategies for reading image subsets. The first is immediate execution, where users specify the geographic area of interest and the desired resolution, retrieving data promptly. The second, deferred execution, loads and caches tiles dynamically as needed, making this approach ideal for large datasets. The deferred strategy, inspired by the legacy Java Advanced Imaging (JAI) library, is integrated into the SIS JavaFX viewer. An optional panel can visualize the byte ranges of the tiles that are loaded dynamically. While deferred execution offers flexibility, it is less predictable than immediate reading, which is preferred for benchmarking due to its consistency.

2.2.4. Advancements in GDAL Integration

Geomatys implemented a GDAL binding using Panama technology (a new mechanism introduced in Java 22 for invoking functions available in the C language). Panama replaces the older JNI methods. This evolution simplifies application distribution by eliminating the need for intermediate C/C+&#43 code, as Panama allows direct calls to native libraries. The binding integrates seamlessly with Apache SIS's API, encapsulating GDAL functionalities while redirecting error handling to Java loggers. While currently limited to two-dimensional read-only raster data, future updates aim to include multi-dimensional support without altering the user experience, leveraging Apache SIS's inherently multi-dimensional design.

2.2.5. Differences Between Apache SIS and GDAL

Apache SIS and GDAL differ in their handling of user requests and performance strategies. Apache SIS expands user requests to tile boundaries, enabling reuse of tiles (caching) without copying the data returned to the user. In contrast, GDAL strictly returns data for the requested region. Apache SIS also avoids resampling during I/O, ensuring at least the requested resolution, whereas GDAL may choose slightly lower-resolution pyramid levels for faster operations. Additionally, Apache SIS's georeferencing APIs support both linear and non-linear transformations from pixel values to CRS coordinates. Generally, Apache SIS uses abstractions to avoid, as much as possible, copying and transforming data. This is at the cost of providing

data in a way that is less directly accessible to users. For example, accessing pixel values should be done through the standard Java2D API or through Apache SIS's `PixelIterator` instead of looping directly on the array's content. This is because the data layout in arrays may be complex (arbitrary image origin, tiling and pixel interleaving, with possibly deferred loading of data from the file). The use of Java2D or other abstraction APIs is necessary for hiding this complexity. By contrast, GDAL returns arrays reorganized in a uniform data layout easy to process directly in a `for` loop, without an abstraction API.

2.3. Ecere — Contribution

As part of implementing support for encoding output as GeoHEIF in the GNOSIS library and verifying the output in viewers, Ecere made contributions to the libheif and GDAL open source projects.

2.4. Silvereye Technology — Impact on libheif, GDAL, and GPAC

Silvereye Technology contributed extensively to open source projects, enhancing tools and libraries that support advanced image formats and geospatial data processing. Their efforts focused on improving libheif, GDAL, libavif, GPAC, and GStreamer, driving progress in interoperability and implementation efficiency for image and video tools.

2.4.1. libheif Contributions

Silvereye Technology made significant enhancements to libheif, which handles HEIF and AVIF image formats. They extended support for the uncompressed codec as defined in ISO/IEC 23001-17, enabling encoding of advanced data types such as high-bit-depth images, raw sensor outputs, and floating-point values. This refinement improves functionality and flexibility for closer-to-sensor data.

Additionally, Silvereye implemented experimental support for lossless compression methods (e.g., Zlib, Deflate, Brotli) and contributed a low-overhead HEIF variant that reduces file overhead for single, small tiles. These updates were developed collaboratively and incorporated into the codebase, advancing interoperability and readiness for broader adoption.

2.4.2. GDAL Contributions

In GDAL, a critical library for geospatial data handling, Silvereye Technology introduced tiled access for HEIF files, optimizing how large images are processed by reducing data transfer and latency. They also added write support for HEIF files, enabling applications using GDAL

to output this format. Ongoing work includes the integration of GeoHEIF metadata and improvements to overview support, pushing HEIF toward mainstream use in geospatial applications.

2.4.3. libavif Contributions

While working with libavif, a library for the AV1 Image File Format, Silvereye Technology addressed gaps in the API definitions that hindered the implementation of GeoHEIF metadata. They collaborated with the libavif team to propose API changes and validated their flexibility by implementing an early version of GeoHEIF in the GDAL driver for AVIF files. These contributions underline the compatibility of GeoHEIF with both libheif and libavif ecosystems.

2.4.4. GPAC Contributions

Silvereye Technology also contributed to GPAC, an open source multimedia project. They updated tools like MP4Box to handle new metadata for generic compression, enhancing debugging and conformance capabilities. These contributions support advanced video and imagery workflows, particularly within the ISO Base Media File Format context.

2.4.5. Collaborative Impact

Silvereye Technology's contributions are marked by collaboration with experts and open source communities, such as Dirk Farin and Yannis Guyon. Their work strengthens the ecosystem of advanced image formats and geospatial tools, reducing implementation effort for future projects and ensuring compatibility across libraries and applications.

3 OUTLOOK

3 OUTLOOK

The GeoHEIF reader in Apache SIS only supports a limited selection of data structures ("boxes") defined by ISO/IEC 14496-12:2022 and ISO/IEC 23008-12:2022, specifically those utilized in Testbed 20. Consequently, future efforts should consider expanding support for additional data structures. Preliminary discussions regarding cloud-optimized support for GIMI have been covered in the high-level description of the read operation in the Testbed 20 Coverage Format Selection Report [OGC 24-039]. Another potential avenue is developing open source libraries or codecs that efficiently utilize cloud-optimized format capabilities. This would involve creating tools or libraries that integrate data with common data analysis frameworks (such as the Python Data Analysis Library (pandas), xarray, ArcGIS, QGIS) through low-level integration.

Further open source developments could center around creating GIMI Compliance Test Tools designed to verify that data formats comply with requirements for streaming or serving from servers or field workstations.

In OGC Testbed 20, support for handling other datatypes than 8-16 bit integers was added to libheif (floating point, complex numbers, integers with more than 16 bits). However, these datatypes are not yet supported in the ISO/IEC 23001-17:2024 implementation, which is the only codec standard supporting all of these data-types. Since these data types are important for many applications (see Testbed 20 GIMI Coverage Format Selection Report [OGC 24-039] report), implementations would be important to support these applications. It might also be interesting to check whether JPEG-2000 can be used to implement some of these datatypes (integers with high bit-depth) to provide a higher-compression option.

Additional open source work for wider GIMI and GeoHEIF video and image sequence support in common frameworks would increase adoption. For example, it would be useful if GStreamer and FFmpeg supported image sequences and video with GIMI and GeoHEIF metadata, which could bridge the gap between traditional GIS imagery and small UAS derived video.

On the tiled images front, consideration could be given to extending 'tili' to include tiled video sequences, suitable for very high-resolution video or vector-valued data. Additionally, region annotation in three-dimensional 'tili' images could be explored. A viewer application and demo for multi-dimensional images in the 'tili' tiling format could be developed. Possible use cases include:

- Displaying 3-D voxel images, storing multispectral images utilizing the frequency band index as the third dimension;
- Enabling storage of complex numbers or vectors using lossy codecs by employing an additional dimension to differentiate between real and imaginary components; and
- Facilitating the storage of an image series based on various parameters such as exposure duration, capturing angle, or time of day.

Regarding network streaming of images, implementing a read-to-use network streaming API for libheif to support interactive image display from a network URL would be advantageous.

The developments presented in “Annex B.4 Network Streaming Interface ” can be used for implementing remote access to files via HTTP range. Support for HTTP range, in combination with an optimal distribution of boxes in the HEIF file and the proposed tile extension (see Testbed 20 GIMI Lessons Learned and Best Practices Report [OGC 24-040], Annex B) could constitute the bases for a future in Could Optimized HEIF that could be formalized and tested in future Testbeds or activities.

4

SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS

4 SECURITY, PRIVACY AND ETHICAL CONSIDERATIONS

In the context of open source geospatial projects, especially those related to Geo-intelligence formats, there are clear benefits and drawbacks concerning security, privacy, and ethical implications. On the positive side, open source software enhances transparency, enabling developers and users to scrutinize and enhance the code to identify vulnerabilities, which helps reduce possible security threats. Moreover, any party can audit open source projects, ensuring that data privacy and ethical issues, particularly regarding the handling of personal or sensitive data, are upheld. Additionally, the inclusion of contributions from diverse global communities increases the likelihood that these projects will comply with international data privacy standards and security.

Nevertheless, the open source model presents its own set of challenges. A significant concern is the possibility of fragmented security practices. Although open source software can undergo thorough reviews, its security largely relies on the community's commitment and expertise to uphold stringent standards. This dependency may result in uneven implementation of security measures or delayed reactions to new threats. In addition, open source geospatial tools might not always face the same level of commercial scrutiny as proprietary options, potentially leading to gaps in user data privacy and ethical considerations. Moreover, since open source projects are freely accessible, there is a danger that ill-intentioned individuals might misuse them for illicit activities, such as inappropriate geospatial surveillance or other unethical uses. This may be the result of lacking the strict controls found in commercial software. Therefore, while open source geospatial software has numerous advantages, it necessitates continuous vigilance to maintain its security and ethical integrity.

A thorough review was conducted to identify any potential security, privacy, and ethical concerns. After careful evaluation, it was determined that none of these considerations were relevant to the scope and nature of this report. Therefore, no specific measures or actions were required in these areas.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ISO/IEC: ISO/IEC 14496-12:2022, *Information technology — Coding of audio-visual objects — Part 12: ISO base media file format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2022). https://www.iso.org/standard/83102.html.

[2] ISO/IEC: ISO/IEC 23001-17:2024, *Information technology — MPEG systems technologies — Part 17: Carriage of uncompressed video and images in ISO base media file format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2024). https://www.iso.org/standard/82528.html.

[3] ISO/IEC: ISO/IEC 23008-12:2022, *Information technology — High efficiency coding and media delivery in heterogeneous environments — Part 12: Image File Format*. International Organization for Standardization, International Electrotechnical Commission, Geneva (2022). https://www.iso.org/standard/83650.html.

[4] OGC Testbed 20 Coverage Format Selection Report, 2025.

[5] OGC Testbed 20 GEOINT Imagery Media for ISR (GIMI) Specification Report, 2025.

[6] OGC Testbed 20 GIMI Benchmarking Report, 2025

[7] OGC Testbed 20 GIMI Lessons Learned and Best Practices Report, 2025.

[8] Joe Stufflebeam, Dirk Farin: m70021: [HEIF] Tiled item type for very large images, 2024.

A

ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

ANNEX A (NORMATIVE) ABBREVIATIONS/ACRONYMS

Apache SIS	Apache Spatial Information System
API	Application Programming Interface
AV1	Alliance for Open Media Video 1
AVC	Advanced Video Coding
AVIF	AV1 Image File Format
COG	Cloud-Optimized GeoTIFF
CRS	Coordinate Reference System
ESRI	Environmental Systems Research Institute, Inc.
GDAL	Geospatial Data Abstraction Library
GEOINT	Geospatial Intelligence
GeoTIFF	Geographic Tagged Image File Format
GIMI	GEOINT Imagery Media for ISR
GPAC	GPAC Project on Advanced Content (a recursive acronym)
HDR	High Dynamic Range
HEIF	High Efficiency Image File
HEVC	High Efficiency Video Coding
HTML	Hypertext Markup Language
IANA	Internet Assigned Numbers Authority
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization

ISOBMFF	ISO Base Media File Format
ISR	Intelligence, Surveillance and Reconnaissance
JAI	Java Advanced Imaging
JNI	Java Native Interface
JPEG	Joint Photographic Experts Group
JSON	JavaScript Object Notation
MIT	Massachusetts Institute of Technology
MPEG	Moving Picture Experts Group
OGR	OpenGIS Simple Features Reference Implementation
OSGeo	Open Source Geospatial Foundation
SAI	Sample Auxiliary Information
SRS	Spatial Reference System
SWIG	Simplified Wrapper and Interface Generator
TAI	International Atomic Time
VVC	Versatile Video Coding

B

ANNEX B (INFORMATIVE) DIRK FARIN ALGORITHMIC RESEARCH E.K. — CONTRIBUTIONS TO LIBHEIF

ANNEX B (INFORMATIVE) DIRK FARIN ALGORITHMIC RESEARCH E.K. — CONTRIBUTIONS TO LIBHEIF

B.1. Introduction

Dirk Farin Algorithmic Research is the primary author and maintainer of the “libheif” open source project (http://libheif.org). The libheif library, tools, and API are a comprehensive implementation of the HEIF (High Efficiency Image File) format, as defined in ISO/IEC 23008-12:2022. libheif supports not only the common HEIC and AVIF variants used in typical photo applications but also includes JPEG-2000, uncompressed images (ISO/IEC 23001-17:2024), VVC, JPEG, and AVC codecs. In addition, libheif enables region annotations and provides a low-level interface for custom image properties, such as those defined in GeoHEIF.

B.1.1. Contributions

The OGC Testbed 20 activity provided an excellent opportunity to work on libheif enhancements and bring it a big step forward in supporting the GIMI profile and also improving its overall architecture, which will be beneficial to the general user base.

Dirk Farin Algorithmic Research wants to express thanks to Silvereye Technology for the valuable contributions, especially for the ISO/IEC 23001-17:2024 decoder implementation, but also for improvements in various other parts and for the integration work to other software libraries, like GDAL (see Annex E).

B.2. Image Tiling

B.2.1. Types of Image Tiling

When reading or writing high-resolution images keeping the whole image in memory may not be feasible. For those situations, libheif supports reading and writing images as individual tiles. When reading, only the accessed tiles are read from the file and decoded. Similarly, when

writing, each tile can be encoded separately and added to the image file. Writing the image tiles in arbitrary order is also possible.

libheif supports three methods to store tiled images.

- **`grid` images** — This format saves a tiled image as a collection of small images. Tiles are stored in the HEIF file as separate images, which are then combined into a large `grid` image. This format has a rather large overhead because metadata must be stored for each tile image. The format is also limited to only 65535 tiles. This is the format used by most mobile phone cameras that split the image into smaller chunks, usually into 512×512 tiles.
- **`unci` images** — These are "uncompressed" images according to ISO/IEC 23001-17:2024. This format has tiling built into it with little overhead, but 'unci' can only store uncompressed images or images compressed with lossless compression algorithms (Deflate, Brotli).
- **`tili` images** - This is a proposed proprietary HEIF extension that, similarly to 'grid', compresses each tile independently, but avoids the metadata overhead and works for practically unlimited image sizes. 'tili' also supports skipped (no-data) tiles and higher-dimensional images (such as multi-spectral images or 3D volumes). 'tili' is optimized for efficient streaming of image data over networks. More information about the 'tili' file format can be found in GIMI Lessons Learned and Best Practices Report [OGC 24-040], Annex B.

Reading any of these tiling formats with `libheif` uses the same API. Therefore users do not have to care about the format used internally in the HEIF file.

Note: the functions for writing tiled 'unci' and 'tili' images are still in the 'heif_experimental.h' header until the API is considered stable. If libheif is installed through the distribution's package manager, this experimental file might not be installed. Please compile libheif from source and enable 'WITH_EXPERIMENTAL_FEATURES' in the *cmake* config.

B.2.2. API for Writing Tiled Images

Irrespective of the tiling format used, the tiled image will always be generated first and then individual tiles added to it. The API for generating the tiled image depends on the tiling format used. Adding the tiles to the image is always done with the same API call.

B.2.2.1. Writing 'grid' Images

First, generate the grid image with

```
struct heif_error
heif_context_add_grid_image(struct heif_context* ctx,
                            uint32_t image_width,
                            uint32_t image_height,
                            uint32_t tile_columns,
                            uint32_t tile_rows,
                            const struct heif_encoding_options* encoding_options,
```

```
                                struct heif_image_handle** out_grid_image_handle);
```

Listing B.1 — API for encoding 'grid' images

Note that the `image_width` and `image_height` do not have to be an integer multiple of the tile sizes. If they are smaller, the extra pixels at the right and bottom borders are removed from the decoded image.

Then add the image tiles one by one with a call to

```
struct heif_error
heif_context_add_image_tile(struct heif_context* ctx,
                            struct heif_image_handle* tiled_image,
                            uint32_t tile_x, uint32_t tile_y,
                            const struct heif_image* image,
                            struct heif_encoder* encoder);
```

Listing B.2 — API for adding a single image tile

For `grid` images, all image tiles must be filled. Skipped (no-data) tiles are not allowed. You may add the tiles in any order.

B.2.2.2. Writing `unci` Images (ISO 23001-17)

For tiled `unci` images, you first create the `unci` image with

```
struct heif_unci_image_parameters {
  int version;

  // --- version 1

  uint32_t image_width;
  uint32_t image_height;

  uint32_t tile_width;
  uint32_t tile_height;

  enum heif_metadata_compression compression;

  // ...
};

struct heif_error
heif_context_add_unci_image(struct heif_context* ctx,
                            const struct heif_unci_image_parameters* parameters,
                            const struct heif_encoding_options* encoding_options,
                            const struct heif_image* prototype,
                            struct heif_image_handle** out_unci_image_handle);
```

Listing B.3 — API for adding a tiled uncompressed image

The ‘prototype’ parameter is an image with the same color channels and settings as used for the individual tiles. This dummy image is not coded. It is used to specify the image format. Very small image planes (1×1) can be used in the prototype image as their size is not used. Different than for ‘grid’ images, for ‘unci’ images, different from ‘grid’ images, the ‘image_width’ and ‘image_height’ must be an integer multiple of the tile sizes. The ‘compression’ parameter selects the lossless compression algorithm (deflate, zlib, brotli).

Tiles can then be added to the `unci` image as above with

```
struct heif_error
heif_context_add_image_tile(struct heif_context* ctx,
                            struct heif_image_handle* tild_image,
                            uint32_t tile_x, uint32_t tile_y,
                            const struct heif_image* image,
                            struct heif_encoder* encoder);
```

Listing B.4 — API for adding a single image tile

The `tile_x`, `tile_y` parameters specify the tile position as indices (0;0), (0;1), (0;2). These are not pixel coordinates.

B.2.2.3. Writing `tili` Images

For tiled 'tili' images, first create the 'tili' image and then add the individual tiles. The image is first created with:

```
struct heif_tiled_image_parameters {
  int version;

  // --- version 1

  uint32_t image_width;
  uint32_t image_height;

  uint32_t tile_width;
  uint32_t tile_height;

  uint32_t compression_type_fourcc;

  uint8_t offset_field_length;   // one of: 32, 40, 48, 64 (bits)
  uint8_t size_field_length;     // one of:  0, 24, 32, 64 (bits)

  uint8_t number_of_extra_dimensions;  // 0 for normal images, 1 for volumetric
(3D), ...
  uint32_t extra_dimensions[8];        // size of extra dimensions (first 8
dimensions)

  uint8_t tiles_are_sequential;  // (bool) hint whether all tiles are added in
sequential order
};

struct heif_error
heif_context_add_tiled_image(struct heif_context* ctx,
                             const struct heif_tiled_image_parameters*
parameters,
                             const struct heif_encoding_options* options,
                             struct heif_image_handle** out_tiled_image_handle);
```

Listing B.5 — API for adding a `tili` image

The `compression_type_fourcc` corresponds to the image item type usually stored in the HEIF file, e.g., `hvc1` for h.265, `av01` for AVIF. However, this value is not used when encoding an image

as the compression type is set automatically when encoding a tile. This variable is only there when getting this information from an existing file.

The `tili` image contains a table with offset pointers to the individual tiles in the file. The user/developer can choose the bit-length of these offsets and the tile sizes. When setting the `size_field_length` to 0, no tile size will be stored. Note that omitting the tile sizes will force the decoder to load the whole offset table when parsing the file. This may be undesirable when the file should be streamed over the network. When all tiles are added sequentially (top left to bottom right), the size can be omitted and the table can still be read on-demand. The total file size of the table is the combined bit-length of the two fields times the number of tiles.

Now, using the same function as with the other tiling methods, tiles can be added:

```
struct heif_error
heif_context_add_image_tile(struct heif_context* ctx,
                            struct heif_image_handle* tile_image,
                            uint32_t tile_x, uint32_t tile_y,
                            const struct heif_image* image,
                            struct heif_encoder* encoder);
```

Listing B.6 — API for adding a single tile

The same `heif_encoder` with the same settings for all tiles must be used. The `tile_x`, `tile_y` parameters specify the tile position as indices (0;0), (0;1), (0;2). These are not pixel coordinates.

B.2.3. API for Reading Tiled Images

Reading images works the same for all tiling schemes and even for non-tiled images. Non-tiled images will appear as images consisting of a single tile.

Before decoding a tile, the first step should be to get the tiling information with:

```
struct heif_error
heif_image_handle_get_image_tiling(const struct heif_image_handle* handle,
                                   int process_image_transformations,
                                   struct heif_image_tiling* out_tiling);
```

Listing B.7 — API for requesting image tiling information

The Boolean parameter `process_image_transformations` indicates whether `libheif` should take care of all image transformations (rotations, mirroring, cropping) internally, or whether want to handle them. If 'process_image_transformations' is enabled, `libheif` will also convert the tile coordinates such that it looks to the client application as if the image geometry is not transformed.

The above function returns the following tiling information:

```
struct heif_image_tiling
{
  int version;

  // --- version 1

  uint32_t num_columns;
  uint32_t num_rows;
  uint32_t tile_width;
```

```
  uint32_t tile_height;

  uint32_t image_width;
  uint32_t image_height;

  // Position of the top left tile.
  // Usually, this is (0;0), but if a tiled image is rotated or cropped, it may
be that the top left tile should be placed at a negative position.
  // The offsets define this negative shift.
  uint32_t top_offset;
  uint32_t left_offset;

  uint8_t number_of_extra_dimensions;  // 0 for normal images, 1 for volumetric
(3D), ...
  uint32_t extra_dimension_size[8];    // size of extra dimensions (first 8
dimensions)
};
```

Listing B.8 — Image tiling information structure

In general, assume that the `image_width` and `image_height` are no integer multiples of the tile size. This constraint will be the case for 'unci' images, but not for the other tiling types. When a tile extends beyond the image border, the client application should not draw the part of the tile that is outside of the image border. Libheif will not crop the border tiles.

Internally, tiles may extend beyond the right and bottom borders, but when `process_image_transformations`, the image may be rotated and cropped and tiles may extend beyond all four borders. For this reason, the fields `top_offset` and `left_offset` indicate how much of the top and left border should be removed when displaying the image.

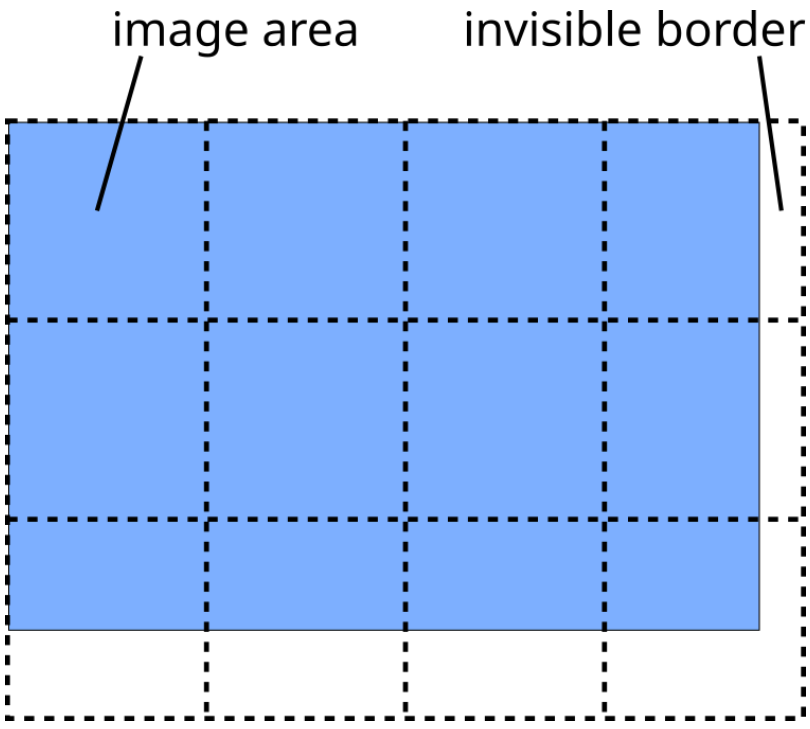

Figure B.1 — Tiled image with ignored right and bottom border.

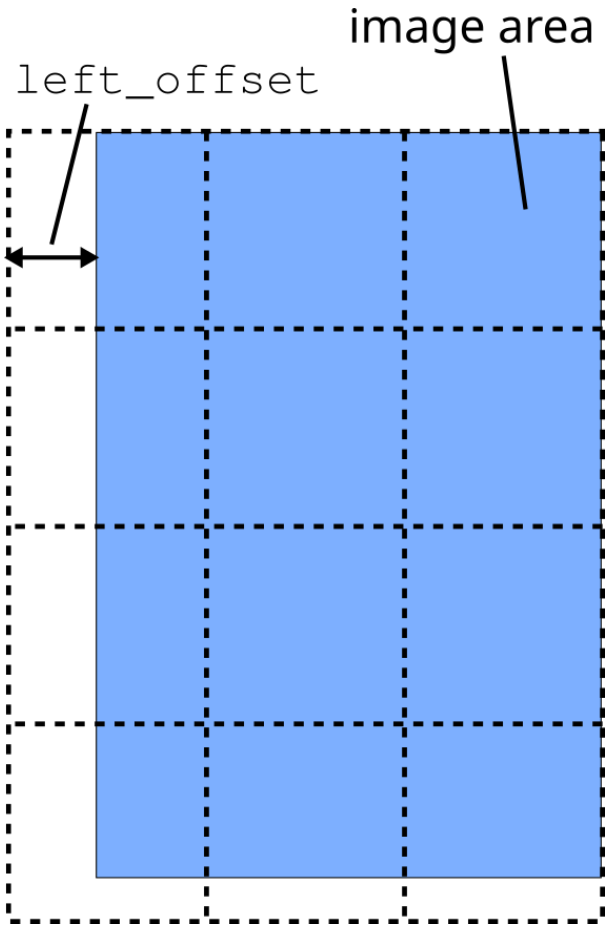

Figure B.2 — Image has been rotated by 90 degrees. Now the `left_offset` is greater than 0.

Now individual tiles can be decoded with:

```
struct heif_error
heif_image_handle_decode_image_tile(const struct heif_image_handle* in_handle,
                                    struct heif_image** out_img,
                                    enum heif_colorspace colorspace,
                                    enum heif_chroma chroma,
                                    const struct heif_decoding_options* options,
                                    uint32_t tile_x, uint32_t tile_y);
```

Listing B.9 — API for decoding a single image tile

As with encoding, `tile_x` and `tile_y` specify the tile position index, not the pixel coordinate. The other parameters are the same as for the usual `heif_image_handle_decode_image()` function. Make sure that the `heif_decoding_options` value `ignore_transformations` is set to the opposite as `process_image_transformations`.

B.3. Multiresolution Pyramids

Closely related to reading high-resolution images is the feature to store lower-resolution overview images in the same file. These make it easier to display zoomed-out views of the image without having to read large areas of the image at the highest resolution and scaling it down.

Multiresolution image pyramids are stored as a set of images, one for each resolution layer. These layer images are combined into a pyramid that is stored as a `pymd` entity group.

A `pymd` entity group also contains some metadata for each layer:

```
struct heif_pyramid_layer_info {
  heif_item_id layer_image_id;
  uint16_t layer_binning;
  uint32_t tile_rows_in_layer;
  uint32_t tile_columns_in_layer;
};
```

Listing B.10 — Data structure for the pyramid layer metadata

This includes the subsampling factor (`layer_binning`) and the number of tiles in the layer. The `pymd` metadata can be requested with

```
struct heif_pyramid_layer_info*
heif_context_get_pyramid_entity_group_info(struct heif_context*,
                                           heif_entity_group_id id,
                                           int* out_num_layers);
```

Listing B.11 — API for reading the image pyramid information

B.3.1. Decoding the Multiresolution Pyramid Information

The `pymd` entity group can be retrieved with

```
struct heif_entity_group
{
  heif_entity_group_id entity_group_id;
  uint32_t entity_group_type;  // this is a FourCC constant
  heif_item_id* entities;
  uint32_t num_entities;
};

heif_entity_group*
heif_context_get_entity_groups(const struct heif_context*,
                               uint32_t type_filter,
          heif_item_id item_filter,
          int* out_num_groups);
```

Listing B.12 — API for getting the entity group information

More specifically, the type_filter to `heif_fourcc('p','y','m','d')` can be set to directly get the entity group of the pyramid if it is present. The 'item_filter' can be set to the primary image ID to get the main image pyramid. However, there will often be only one pyramid in the file, but

consider the case that there are multiple pyramids in one file. The image item IDs ('entities') in the returned entity group are ordered from lowest resolution to highest resolution.

B.3.2. Encoding a Multiresolution Pyramid

First, encode all the resolution layers as separate images. Each layer image in the pyramid can be a tiled image and image types can also be mixed. For example, the highest resolution layer could be an `unci` image that stores the lossless compressed data, while the overview images use `tili` or `grid` with an efficient image codec. It also helps that software that cannot read `unci` images can at least show the overview images.

Once all layer images are created and their item IDs are known, call:

```
struct heif_error
heif_context_add_pyramid_entity_group(struct heif_context* ctx,
                                      const heif_item_id* layer_item_ids,
                                      size_t num_layers,
                                      heif_item_id* out_group_id);
```

Listing B.13 — API for adding an image pyramid

This will group the images into a `pymd` group. The image IDs can be listed in any order. libheif sorts the layer images according to their size and automatically generates the meta-information stored in the `pymd` entity group.

Note that tiles can still be added after the `pymd` entity group has been generated. For example, first create "empty" images with `heif_context_add_tiled_image()` for each layer, generate the `pymd` entity group, and then start encoding the image tiles.

B.4. Network Streaming Interface

In GIMI/HEIF images, there are essentially two top-level boxes: one 'meta' box that contains all metadata and image properties, and one 'mdat' box that contains all the coded image data.

- The `meta` box contains many child boxes that must be read to determine the content.
- The `mdat` box is a large raw data container without obvious internal structure. The image items defined in the 'meta' box point to ranges in the `mdat` box that are required to decode an image. Usually, the data required for one image is saved contiguously in the `mdat` box content (even though that does not have to be the case).

Thus, a decoder will need one or two accesses to read the `meta` box, and can then issue accesses to the `mdat` whenever an image tile should be decoded.

When opening a file, `libheif` will parse the file structure and read the `meta` box. When a single image tile is decoded, `libheif` will only load the `mdat` data of that tile from the file. This is

particularly important when streaming the image over a network without first downloading the whole image.

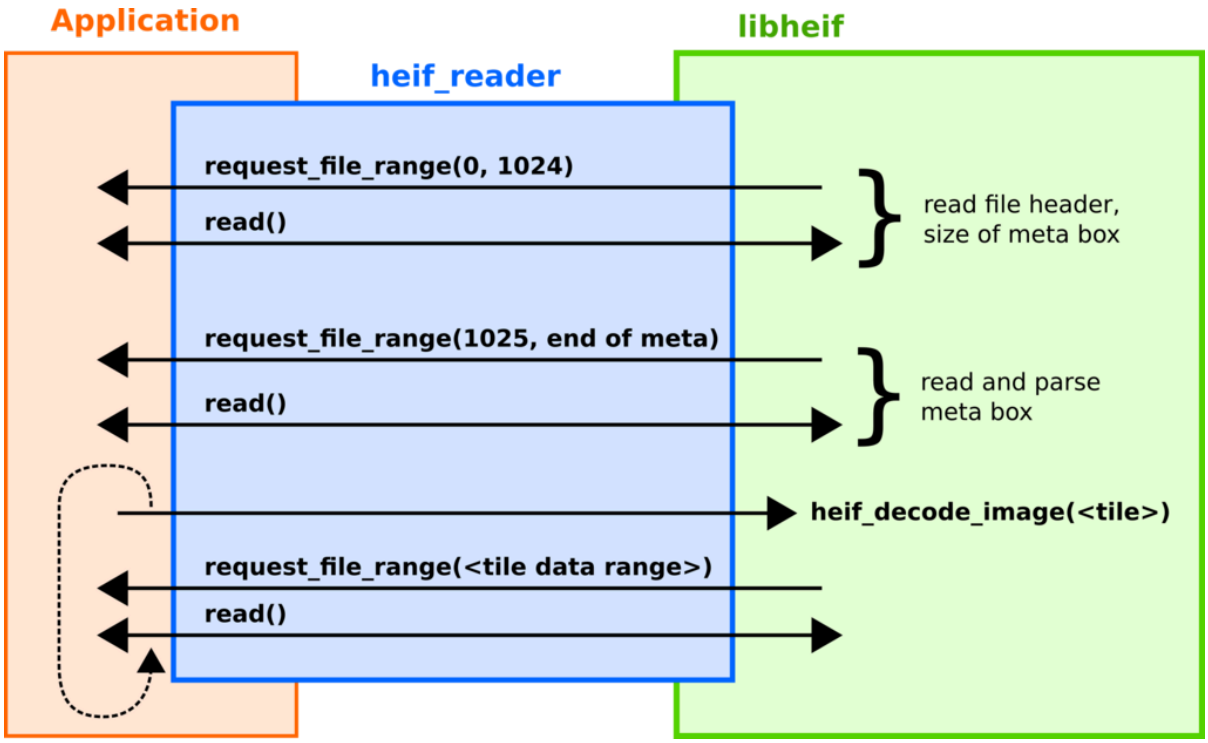

Figure B.3 — Interaction between libheif and client application when decoding image tiles on-demand for a streaming file source.

The reading process is illustrated in Figure B.3. Initially, libheif does a read of the first 1KB of the file to determine the file type and the size of the `meta` box. Then, during a second read (if the 'meta' box is larger than 1KB), it performs another read of the remaining 'meta' box. All subsequent reads are on-demand reads into the `mbox`. With the `tili` format, the `meta` box size is only about 1KB even for very large images and multi-resolution pyramids. With the HEIF grid format, the `meta` box can be several MBs.

B.4.1. Implementing a File Reader Interface

Out of the box, libheif supports reading HEIF files from either a file stored in the filesystem or from memory. However, libheif also provides a generic reader interface that can be implemented by the client application to connect libheif to other data sources. This 'heif_reader' interface can be used to implement a direct download of the HEIF file from a network.

Apart from the usual 'read' and 'seek' functions, which were previously available, the interface was extended in Testbed 20 to a version 2 that also includes the function `heif_reader_range_request_result request_range(uint64_t start_pos, uint64_t end_pos, void* userdata)`. `libheif` will call this function to tell the reader which file range it is going to read next. This should be a blocking call in which you can download the data from the network. The advantage of downloading the file in the `read()` function is that the `read()` function may read many small chunks, while `request_range()` will request larger file ranges that are more efficient

to download. More data than requested may be downloaded and then let libheif know in the result that this data is available. libheif may decide to use that extra data if useful.

Another optional function is 'void preload_range_hint(uint64_t start_pos, uint64_t end_pos, void* userdata)'. With this function libheif indicates that it may need this file range in the future. Contrary to the above, this function should be non-blocking and return immediately to the calling function. So that the content is ready in case the range is requested later a download in the background may be useful.

If you are caching network data, consider the callback void 'release_file_range(uint64_t start_pos, uint64_t end_pos, void* userdata)' which is used to let the application know if libheif does not need a specific file range anymore and the content can be removed from the cache. Note any data can be removed from the cache whenever desired. However, remember that the content may need to be reloaded if libheif requires that content again.

B.5. Encoding Tiled Images with heif-enc

The `heif-enc` command line tool is an example application for converting images to HEIF files. `heif-enc` can read JPEG, PNG, TIFF, and Y4M images and it writes all variants of HEIF like HEIC, AVIF, or JPEG-2000 and ISO/IEC 23001-17:2024 uncompressed content.

The basic syntax is:

```
heif-enc input.jpg -o output.heic
```

Listing B.14

To switch between different encoding formats, use the following command line switches:

Table B.1 — Table Encoding formats.

SWITCH	COMPRESSION FORMAT	SUFFIX
default (no switch)	H.265 (HEVC)	.heic
-A / —avif	AV1	.avif
—vvc	H.266 (VVC)	.vvic
—jpeg	JPEG	
—jpeg2000	JPEG-2000	.hej2
—htj2k	HT-JPEG-2000	
-U / —uncompressed	ISO 23001-17	

Alternatively, if no format is explicitly set, `heif-enc` also sets the compression format automatically when it recognizes the suffix of your output filename.

Despite its name, the 'uncompressed' codec can use lossless compression. The compression algorithm can be specified with `--unci-compression METHOD`, where METHOD is either `deflate`, `zlib`, or `brotli` (support for these formats may vary depending on how the `libheif` build was configured).

B.5.1. Encoding Tiled Images

When encoding a high-resolution tiled image, `heif-enc` expects each tile image in a separate input image with filenames that contain two numbers to denote the tile row and column position. For example: `tile-2-3.jpg` or `image-002-003.jpg`.

To switch `heif-enc` into tile input mode, use the option `-T / --tiled-input` and specify **one** tile image as input. `heif-enc` will scan the directory to search for images with the same name pattern. `heif-enc` will detect the number range for the row and column position. This means that the position numbers may start from 0 or 1 (or any other number). `heif-enc` will also detect whether to use leading zeros for the numbers.

Usually, `heif-enc` expects that the first number is the vertical position and the second number the horizontal position. If swapping is required, use `--tiled-input-x-y`.

The generated tiled image will have a size which is the sum of all the tile widths and heights. However, the input image might have some padding at the right and bottom border. In that case, overwriting the total image size with `--tiled-image-width #` and `—tiled-image-height #`is possible.

B.5.1.1. Tiling Modes

The user/application can choose between three different tiling modes.

- `grid`: This is the default method with the best decoder compatibility. However, this method has the largest overhead and the maximum size is limited to 65535 tiles.
- `tili`: This is a more efficient tiling format with low overhead and supports a nearly unlimited number of tiles. Note: the format is still experimental and currently only supported by `libheif`.
- `unci`: This uses the internal tiling support of the ISO/IEC 23001-17:2024 (uncompressed) image codec. This method can only be used together with that codec. Like `tili`, this method has little overhead and supports larger image sizes.

B.5.2. Multi-Resolution Pyramids

`heif-enc` can also encode a multi-resolution pyramid stack of images. In this case, just specify the input images for each resolution layer on the command line and provide the option `--add-pyramid-group`.

For a hierarchical pyramid of tiled images, provide the option '—tiled-input' and specify the filename of one tile for each pyramid layer. An example command line would be:

```
heif-enc --tiled-input layer1-001-001.jpg layer2-01-01.jpg layer3-01-01.jpg -o
output.heic --add-pyramid-group
```

Listing B.15

The input layer images can be specified in any order; `heif-enc` will sort them by size and add them in the correct order to the pyramid group.

B.6. Example Tiled Image Viewer

An example viewer application for tiled HEIF files was developed during Testbed 20. The viewer is available at https://github.com/farindk/tiled-image-viewer. The viewer supports the display of HEIF images in all three tiling formats and can be used to display images of arbitrary resolution. On that webpage, links to example images can be found.

Figure B.4 — Tiled HEIF image viewer example application.

B.7. Data Types Support

Previously, `libheif` supported image with up to 16 bit unsigned integer bit-depth for each color channel. In Testbed 20, this capability was extended with support for unsigned and signed integers up to 64 bits, 32 and 64 bit floating point, and 32 and 64 bit per component complex numbers. This provides the basis for an implementation of these data-types in the ISO/IEC 23001-17:2024 codec.

However, until that implementation is finalized, the API for creating and reading images with these datatypes can be found in the `heif_experimental.h` header file.

B.8. Reading and Writing Sequences

B.8.1. Introduction

HEIF files can contain image sequences in addition to still images. The HEIF specification builds upon the ISO14496-12 ISOBMFF standard with a few extensions from ISO23008-12 (HEIF) and GIMI specific definitions from NGA.STND.0076_1.0.

During the Testbed 20 GIMI Task Extension, support for HEIF image sequences was added to libheif. In particular, this includes intra-coded sequences in the JPEG2000, HTJ2K, and ISO23001-17 uncompressed format. However, the implementation also covers H.264, H.265, H.266, AV1, and JPEG (only decoding for H.264). Motion-predicted video sequences are not yet supported.

Sequences may consist of several media tracks. A media track is either a visual track or a metadata track. Other track types (e.g., audio) are currently not supported.

A track contains timed samples. Metadata samples can contain any arbitrary data and the track header specifies what kind of data is contained through an URI-based naming scheme. Samples from different tracks do not necessarily come in pairs. Instead, they can be timed independently. For example, while the visual track may carry a 30 fps video stream, the metadata track may contain acceleration sensor data, sampled at a higher rate.

Additionally, each sample (image or metadata) can contain sample auxiliary information (SAI). This is a usually small data package with auxiliary information that is directly attached to the sample. Libheif currently supports TAI timestamps and textual content IDs as defined in the GIMI standard.

Finally, metadata can also be attached to a track as a whole. Currently, libheif support the TAI timestamp configuration block and GIMI track content IDs.

Relations between tracks are defined by track references. For example, a ‘cdsc’ (Content Description) reference from the metadata track to the visual track denotes that this metadata track is further describing the referenced visual track.

The API specific to sequence processing can be found in the libheif/heif_sequences.h header file. This document only describes the basic steps to decode and encode HEIF sequences. For more specific functionality, the reader is referred to that header file.

B.8.2. Reading HEIF Sequences

After opening the HEIF file as usual and obtaining a heif_context object, the application can check whether there is an image sequence in the file with:

```
int heif_context_has_sequence(heif_context*)
```

Listing B.16 — API for checking whether HEIF file contains a sequence

Note that a file can contain still images and a sequence at the same time.

To get an overview what tracks are contained in the HEIF file, the application can use the functions:

```
int heif_context_number_of_sequence_tracks(const struct heif_context*);
void heif_context_get_track_ids(const struct heif_context* ctx, uint32_t out_
track_id_array[]);
```

Listing B.17 — API for getting the sequence track IDs of an HEIF file

To then get access to a specific track with

```
struct heif_track* heif_context_get_track(const struct heif_context*, uint32_t
id);
```

Listing B.18 — API for accessing sequence tracks of an HEIF file

Alternatively, for convenience, you can call heif_context_get_track(ctx, 0) with a zero ID to get the first (and usually only) visual track in the file.

To check what kind of data the track contains:

```
heif_track_type heif_track_get_track_handler_type(struct heif_track*);
```

Listing B.19 — API for getting a sequence track type

This can be an image sequence, a video, or metadata. Other track types (e.g., audio) are currently not supported by libheif, but the raw data can still be read to some extent with the same API as for metadata tracks.

B.8.2.1. Timescale

Each track has a timescale, which specifies the clock ticks per second which are the basis for all timing values given in a track. Note that the timescale can be different for each track and that there is also a timescale assigned to the total sequence, which can also be different than the track timescales. When the application gets the total track duration or a sample duration, it has to divide the duration by the timescale to get the duration in seconds.

B.8.2.2. Reading Tracks

B.8.2.2.1. Visual Tracks

Decoding can optionally start by querying the track resolution with

```
struct heif_error heif_track_get_image_resolution(heif_track*, uint16_t* out_
width, uint16_t* out_height);
```

Listing B.20 — API for getting the resolution of a visual sequence track

Then images can be decoded in temporal order with

```
struct heif_error heif_track_decode_next_image(struct heif_track* track,
                                               struct heif_image** out_img,
                                               enum heif_colorspace colorspace,
                                               enum heif_chroma chroma,
                                               const struct heif_decoding_
options* options);
```

Listing B.21 — API for decoding images from a visual sequence track

The decoded image is stored in out_img. When the last image has been read and the application is reading past the end of the sequence, the error heif_error_End_of_sequence is returned.

The colorspace and chroma parameters can be used to force a specific output color format. If the sequence is coded in a different color format, libheif will convert the image accordingly. When specifying heif_colorspace_undefined and heif_chroma_undefined, libheif will not convert the image but use the most convenient format. Usually, this will be the input video format, but it can also deviate from the codec format because of necessary internal processing.

The image display duration is attached to the image and can be obtained with

```
uint32_t heif_image_get_duration(const heif_image*);
```

Listing B.22 — API for getting the time duration of an image

Remember that the image duration is given in timescale units of the track.

B.8.2.2.2. Metadata Tracks

Reading metadata tracks is similar to reading visual tracks, but instead of images, the raw data is read into heif_raw_sequence_sample objects with

```
struct heif_error heif_track_get_next_raw_sequence_sample(struct heif_track*,
                                                          heif_raw_sequence_
sample** out_sample);
```

Listing B.23 — API for getting the raw metadata sample data

The raw data can then be retrieved from this object with

```
const uint8_t* heif_raw_sequence_sample_get_data(const heif_raw_sequence_sample*,
 size_t* out_array_size);
```

Listing B.24 — API for accessing the metadata sample data

Similar to reading sequence images, the sample duration is attached to the heif_raw_sequence_sample and can be obtained with

```
uint32_t heif_raw_sequence_sample_get_duration(const heif_raw_sequence_sample*);
```

Listing B.25 — API for getting the time duration of a metadata sample

B.8.3. Writing Tracks

B.8.3.1. Visual Tracks

In order to write an image sequence, the application first has to add a visual track with

```
struct heif_error heif_context_add_visual_sequence_track(heif_context*,
                                                         uint16_t width, uint16_t
 height,
                                                         struct heif_track_info*
info,
                                                         heif_track_type track_
type,
                                                         heif_track** out_track);
```

Listing B.26 — API for adding a visual sequence track

The track type can be set to 'heif_track_type_video' or 'heif_track_type_image_sequence'. More information about the track storage configuration is contained in the heif_track_info structure. The easiest way to initialize 'heif_track_info' is to allocate a new structure with

```
struct heif_track_info* heif_track_info_alloc();
```

Listing B.27 — API for allocating a sequence track info configuration structure

As this will fill the structure with the default values, and only the values of interest need to be set. The only field that we are currently interested in is 'uint32_t heif_track_info::track_timescale' defining this track's clock ticks per second. Content of the other fields is described later in this annex. Now that we have a visual track, we can encode images with

```
struct heif_error heif_track_encode_sequence_image(struct heif_track*,
                                                   const struct heif_image*
image,
                                                   struct heif_encoder* encoder,
                                                   const struct heif_encoding_
options* options);
```

Listing B.28 — API for encoding an image into a sequence track

The encoder and options parameters are set similar as for regular HEIF images. They define which codec to use and further encoding options.

Before encoding the image, do not forget to assign the image duration to the image with:

```
void heif_image_set_duration(heif_image*, uint32_t duration);
```

Listing B.29 — API for setting a sequence image time duration

B.8.3.2. Metadata Tracks

Libheif currently uses the "URI Meta Sample Entry" format for metadata tracks. This contains a URI field in the track header that specifies what kind of data is stored in the metadata track, i.e., the binary format of the contained samples. With this information, a new metadata track can be created:

```
struct heif_error heif_context_add_uri_metadata_sequence_track(heif_context*,
                                                               struct heif_track_
info* info,
                                                               const char* uri,
                                                               heif_track** out_
track);
```

Listing B.30 — API for adding a metadata track

Then allocate raw sequence sample objects, copy the raw metadata into them and assign the duration:

```
heif_raw_sequence_sample* heif_raw_sequence_sample_alloc();
heif_error heif_raw_sequence_sample_set_data(heif_raw_sequence_sample*, const
uint8_t* data, size_t size);
void heif_raw_sequence_sample_set_duration(heif_raw_sequence_sample*, uint32_t
duration);
```

Listing B.31 — API for filling a metadata sample with raw data and setting the time duration

Now the sample can be stored in the metadata track:

```
struct heif_error heif_track_add_raw_sequence_sample(struct heif_track*,
                                                     const heif_raw_sequence_
sample*);
```

Listing B.32 — API for adding a raw metadata data packet to the metadata track

B.8.4. Sample Auxiliary Information (SAI)

As mentioned in the introduction, we can attach Sample Auxiliary Information (SAI) to samples in the track. The type of the SAI is specified with a four-character code. Libheif currently supports TAI timestamps and GIMI content IDs as SAIs. The SAI data is assigned to the sample objects. For example, to the heif_image in the case of visual tracks, or heif_raw_sequence_sample in the case of metadata tracks. The SAI data is assigned to the sample objects, i.e., to the heif_image in the case of visual tracks, or heif_raw_sequence_sample in the case of metadata tracks.

When reading a file, content_IDs these can be retrieved from the samples with

```
const char* heif_image_get_gimi_sample_content_id(const heif_image*);
const char* heif_raw_sequence_sample_get_gimi_sample_content_id(const heif_raw_
sequence_sample*);
```

Listing B.33 — API for getting the GIMI content ID from an image or metadata sample

TAI timestamps can be retrieved using:

```
const struct heif_tai_timestamp_packet* heif_raw_sequence_sample_get_tai_
timestamp(const struct heif_raw_sequence_sample*);
struct heif_error heif_image_get_tai_timestamp(const struct heif_image* img,
                                               struct heif_tai_timestamp_packet*
timestamp);
```

Listing B.34 — API for getting the TAI timestamp from an image or metadata sample

There exist similar functions to attach the SAIs to the samples before encoding the samples into the track.

There are also functions to get the list of SAIs that are available in a read HEIF file:

```
int heif_track_get_number_of_sample_aux_infos(struct heif_track*);
void heif_track_get_sample_aux_info_types(struct heif_track*, struct heif_sample_
aux_info_type out_types[]);
```

Listing B.35 — API for getting the TAI timestamp from an image or metadata sample

B.8.5. Track-level Metadata

A track that contains TAI timestamp SAIs also needs a TAIC configuration box. This is assigned to the track as a whole. More correctly, the box is assigned to a cluster of samples, but since there is usually only one cluster of metadata samples, the TAIC is assigned to the whole track. The configuration box can be retrieved using:

```
const struct heif_tai_clock_info* heif_track_get_tai_clock_info_of_first_
cluster(struct heif_track*);
```

Listing B.36 — API for getting the TAI clock information for the track

When writing tracks with SAIs attached to the samples, libheif offers two possibilities as to how the data is structured in the file:

It is also possible to assign a track-level GIMI content ID to the track. This can be obtained with

```
const char* heif_track_get_gimi_track_content_id(const struct heif_track*);
```

Listing B.37 — API for getting the GIMI content ID for the track

and when writing the track, it is also specified in heif_track_info.

B.8.6. SAI Interleaving

When writing tracks with SAIs attached to the samples, libheif offers two possibilities how the data is laid out in the file.

- The SAIs can be stored right after the image/metadata sample. This is advantageous if the file should be streamed, as all sample data is then clustered together.
- All SAIs can be stored in one contiguous block. This means, the file contains a block with all sample data, and one block for each of the SAI types. This results in slightly smaller file

sizes since libheif does not need to save file offsets to the individual sample SAIs as it can make use of a special storage mode when all SAIs are stored contiguously.

Which mode to choose can be specified in the heif_track_info structure.

B.8.7. Track References

Usually, metadata tracks provide additional information related to a visual track. This can be indicated in the HEIF file by setting a track reference. A track reference has a type, which in this case is 'cdsc' (Content Description). When writing a file, the track references can be added with

```
void heif_track_add_reference_to_track(heif_track*, uint32_t reference_type,
heif_track* to_track);
```

Listing B.38 — API for adding a track reference to another track

In order to indicate that a metadata track is describing a visual track:

```
heif_track_add_reference_to_track(metadata_track, heif_track_reference_type_
description_of, visual_track);
```

Listing B.39 — Specifying that a track contains metadata for another track

For the corresponding functions to list the track references when reading files, consult the libheif/heif_sequences.h header file.

B.9. Libheif Command Line Tool Support for Image Sequences

B.9.1. Encoding Sequences with `heif-enc`

Two options were added to the all-around command-line tool for encoding HEIF files 'heif-enc' to encoding image sequences:

```
  -S, --sequence            encode input images as sequence
                            (input filenames with a number will pull in all
files with this pattern).
      --timebase #          set clock ticks/second for sequence
      --duration #          set frame duration (default: 1)
      --fps #               set timebase and duration based on fps
      --vmt-metadata FILE   encode metadata track from VMT file
```

Listing B.40

Option `-S` switches `heif-enc` to sequence encoding mode. The input images are read from individual image files. However, all input files do not need to be specified on the command line. `heif-enc` assumes that the input files are numbered like `image0001.jpg` to `image4200.jpg`.

Only one filename has to be specified. The tool will automatically detect the number part in the filename and search for the first and last number in that directory.

The frame-rate can be specified in two ways.

- Specify the frames-per-second with option '—fps #'. This option will accept non-integer frame-rates. It will also detect the special frame-rate 29.97 and encode that appropriately (as 30000/1001 frames per second).
- Set the clock tick rate with '—timebase #' and the duration of each frame in clock ticks with '—duration #'. In this case, both numbers must be integers.

As an experimental option, `--vmt-metadata FILE` was added. This reads the VMT metadata from the given file and adds it as a separate metadata track. This option will only be available when libheif has been compiled when using the option `-DENABLE_EXPERIMENTAL_FEATURES=on` during cmake configuration.

B.9.2. Decoding Sequences to Separate Images

The existing `heif-dec` command line tool has been extended to decode sequences and write the images to individual images. Decoding of sequences (instead of decoding the still images) is enabled with the command line option `-S` or `--sequence`. Execution of this command will output a numbered list of images.

B.9.3. Getting File Information with `heif-info`

The `heif-info` command line tool will output summary information about the image sequence HEIF file such as the following:

```
MIME type: image/heis
main brand: heis
compatible brands: msf1

sequence time scale: 30 Hz
sequence duration: 24 seconds
track 1
  handler: 'vide' = video
  resolution: 1024x1024
  sample entry type: hvc1
  sample auxiliary information: stai, suid
track 2
  handler: 'meta' = metadata
  sample entry type: urim
  uri: urn:smpte:EXAMPLE
  sample auxiliary information: stai, suid
  references:
    cdsc: track#1
```

Listing B.41

B.9.4. Displaying Image Sequences

A new command line tool `heif-view` was added to show image sequences in real-time. This command is called with the HEIF file and shows the sequence in a UI window. The Simple DirectMedia Layer (SDL) library is required as a compilation dependency for this tool.

The tool has the following options:

```
heif-view  libheif version: 1.19.7

Usage: heif-view [options]  <input-file>

Options:

  -h, --help                    show help
  -v, --version                 show version
      --list-decoders           list all available decoders (built-in and
plugins)
  -d, --decoder ID              use a specific decoder (see --list-decoders)
      --speedup FACTOR          increase playback speed by FACTOR
      --show-sai                show sample auxiliary information
      --show-frame-duration     show each frame duration in milliseconds
      --show-track-metadata     show metadata attached to the track (e.g., TAI
config)
      --show-metadata-text      show data in metadata track as text
      --show-metadata-hex       show data in metadata track as hex bytes
      --show-all                show all extra information
```

Listing B.42

With `--speedup #`, the playback speed can be increased or decreased.

The options starting with `--show*` enable output of the respective data in the terminal window. As metadata tracks contain custom data, it can either be shown as plain text with `--show-metadata-text` or as binary hex dump with `--show-metadata-hex` depending on the metadata format used.

C

ANNEX C (INFORMATIVE) ECERE — CONTRIBUTIONS

ANNEX C (INFORMATIVE) ECERE — CONTRIBUTIONS

As part of implementing support for encoding output as GeoHEIF in the GNOSIS library and verifying the output in viewers, Ecere made contributions to the libheif and GDAL open source projects.

GDAL:

- Fixed an axis-ordering bug in GeoHEIF driver, in particular for the EPSG:4326 CRS.
- Tested improvements on reading GeoHEIF properties and contributed to the review discussion.

libheif:

- Identified a bug with `heif_image_add_channel`.
- Requested support and API usage for outputting a coverage with 1..n non-visual channels using float32.
- Fixed warnings including API headers from C/eC.
- Fixed a missing string conversion in heif-enc.cc and filed issue.
- Identified a bug preventing building without threading support even when `ENABLE_PARALLEL_TILE_DECODING` is disabled.
- Added a g++/gcc-8 compatibility tweak for <bit> library header.
- Identified a bug detecting alpha channels in `unci` tiled output.
- Fixed building with older *libjpeg* versions.

D

ANNEX D (INFORMATIVE) GEOMATYS — OPEN SOURCE CONTRIBUTIONS

ANNEX D (INFORMATIVE) GEOMATYS — OPEN SOURCE CONTRIBUTIONS

Geomatys developed two modules in the open source Apache SIS project for reading GeoHEIF files. A pure-Java GeoHEIF reader in the "incubator" group of modules, and a binding to the C/C++ GDAL native library in the "optional" group of modules. Since GDAL itself delegates to `libheif`, the later implementation has two levels of indirection (SIS → GDAL → `libheif`). The locations of those two modules in Apache SIS source code are:

- `incubator/src/org.apache.sis.storage.geoheif/` for the Java implementation; and
- `optional/src/org.apache.sis.storage.gdal/` for the binding to GDAL native library.

The GDAL binding is targeted for Apache SIS 1.5 release. The Java implementation is not yet ready for release but is capable to read the GeoHEIF files used in Testbed 20. Both implementations support tiling.

D.1. Java implementation

Like any library providing access to many file formats, GDAL and Apache SIS define a single format-neutral API implemented by many drivers (typically one driver per file format). In GDAL, the common API obtained after opening a file is `GDALDataset`. In Apache SIS, it is the `org.apache.sis.storage.DataStore` abstract class. Therefore, implementing a GeoHEIF reader in Apache SIS consists in creating another `DataStore` subclass. Users do not need to know the existence of such subclasses.

The algorithms needed for reading efficiently random subsets of GeoHEIF files have a lot in common with GeoTIFF. Apache SIS implementations in `DataStore` subclasses try to maximize code reuse between different file formats. This strategy is a little bit different than GDAL, which is more like an aggregator of independent projects. For example, `libpng`, `libtiff`, `libnetcdf`, *etc.* are developed independently, each project with its own API, and GDAL puts a translation layer between project-specific APIs and a unique GDAL's API. In Apache SIS, both GeoTIFF and GeoHEIF inherit the same code base, namely the `TiledGridCoverage` abstract class and related classes. This foundation takes care of computing low-level information such as tile indices, band indices, scanline strides, pixel strides, *etc.* from higher-level information such as the area of interest and the desired subsampling. This foundation also manages the cache of

tiles and the downloads of ranges of bytes from HTTP or S3 servers. Therefore, the task of a GeoTIFF or GeoHEIF reader is reduced to file format specificities such as finding and decoding the metadata, with less work to do regarding the use of those metadata. A side-effect of Apache SIS design is that differences in GeoHEIF and GeoTIFF format processing performance are less likely to be caused by differences in implementation details. This is because there are fewer variations in those details between Apache SIS drivers than between GDAL drivers. However, at the time of writing this Report, the Apache SIS implementation of GeoHEIF is still a prototype not yet optimized to a level comparable to the GeoTIFF implementation.

The GeoHEIF reader implemented in Apache SIS supports only a subset of the data structures ("boxes") defined by ISO/IEC 14496-12:2022 and ISO/IEC 23008-12:2022. The reader currently only supports the boxes used in Testbed 20.

D.1.1. Read operations strategies

Apache SIS offers different strategies for reading a subset of the image:

1. The most straightforward strategy is to let users specify their desired geographic area and resolution, then immediately read the (potentially subsampled) values in all tiles that intersect the specified area. This strategy is efficient when the requested data fit in memory and the users know in advance that they will use most of these data. This strategy is also predictable, as I/O operations happen immediately.

2. Another strategy is to act as if the whole image has been read (so users do not need to specify an area of interest), then SIS automatically reads and caches each tile the first time that it is needed. Oldest tiles are discarded when memory is full and will be reread if needed again later. This strategy is convenient when the geographic areas to read are not well known in advance, or when the data volume is too large to fit in memory. This strategy may be slower than strategy 1 if, in the end, the same set of tiles are read but in random order. It is also less predictable because the actual I/O operations may happen at any (potentially unexpected) time after the users made their request.

The latter strategy (deferred execution) is inspired from Java Advanced Imaging (JAI), which was built on top of Java2D. Despite being 25 years old, JAI is still quite advanced even compared to modern libraries. JAI is not maintained anymore, but Apache SIS reuses many of its ideas, including the use of Java2D as a foundation. Apache SIS provides a custom implementation of the Java2D `java.awt.image.RenderedImage` interface. That interface can represent tiled images backed by different types of `java.awt.image.SampleModel` (data layouts) such as banded (each band in a separated array), pixel interleaved (values of each pixel in consecutive elements of the same array), packed (multiple values packed in a single integer using bitmasks), *etc.* The image upper-left pixel is not necessarily associated with the (0,0) coordinates. The complexity of accessing data in such a diversity of layouts is hidden behind standard Java2D API such as `Raster.getSample(x, y, band)`. Apache SIS adds another optional abstraction level inspired from JAI with `PixelIterator`. The Java2D standard library has specialized implementations of `SampleModel` and `Raster` for optimizing the most common data layouts, with delegations to the video memory and GPU when possible.

Strategy #2 (deferred execution) can be seen in action with the SIS viewer. The figure below shows one of the images used in Testbed 20. The rectangle outline on the right of the "ACT2017.tiff" label represents the whole file. The filled rectangles inside the outline represent the ranges of bytes that have actually been read. This representation is updated dynamically as the user zooms and pans the image. This on-demand reading of range subsets works with file systems, HTTP ranges, and Amazon S3. The figure below shows the result after a few of such user's actions.

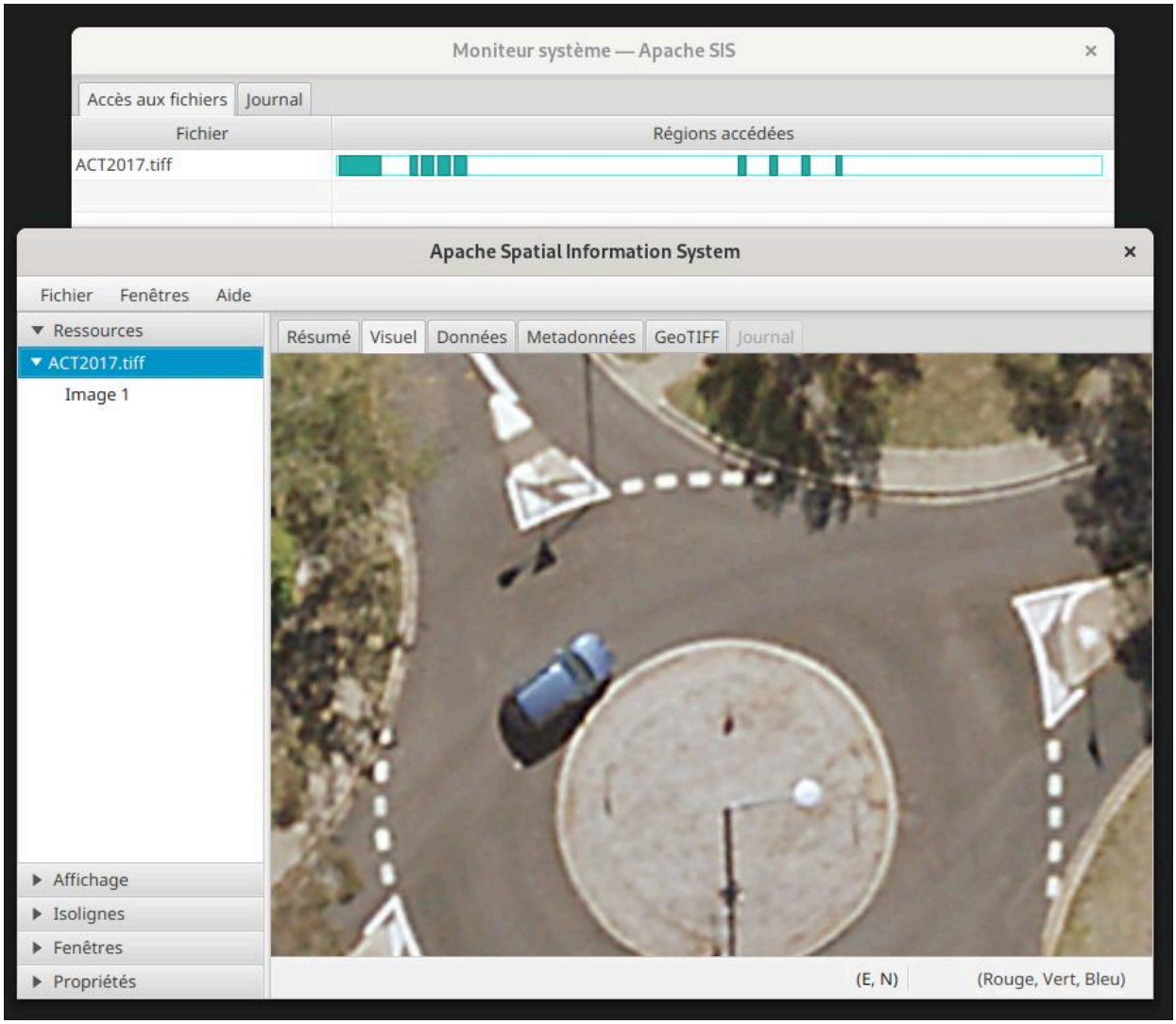

Figure D.1 — Visualizing the ranges of bytes read from a file

Contraryy to the viewer application, the benchmarking described in the OGC Testbed 20 GIMI Benchmarking Report OGC [[_4-041]] uses strategy 1 because of its more predictable nature.

D.1.1.1. HTTP range requests

Apache SIS reads all its data from a 'java.nio.channels.ReadableByteChannel'. A channel may be a file, an HTTP or AWS S3 connection, or anything else an implementer chooses. A channel can optionally be "seekable" thereby allowing jumping to an arbitrary position. However, in

HTTP and S3 cases, jumping to a new position implies closing the current connection and opening a new one specifying a range of bytes, i.e., a start and end positions rather than only the start position. To enable this capability, Apache SIS uses internally a custom sub-type of 'ReadableByteChannel' with a 'rangeOfInterest(lower, upper)' method. The range of bytes specified to that method is a hint that implementations are free to ignore. If that hint is provided, SIS will use it for constructing an HTTP range request when a seek is requested. This mechanism is was initially developed for the GeoTIFF reader and is reused by the GeoHEIF reader. This mechanism is transparent to users..

D.1.2. Implementation perspective on GeoTIFF versus GeoHEIF

The structure of a GeoTIFF file is a table of 32-bits (with standard TIFF) or 64-bits (with BigTIFF) pointers. The data referenced by these pointers can be located anywhere in the file in any order, not necessarily near the locations where the pointer values were provided. These pointers are used for various kinds of data, not only the tile pixel values. For example, the data can be information about bands (number of bits per sample, number of samples per pixel, minimum and maximum values), the color map, and the GeoKeys.

For an in memory GeoTIFF image, dereferencing these pointers is very efficient. For a GeoTIFF image on the local file system, dereferencing can be done with file seek operations, which are still reasonably efficient on modern hardware and operating systems. But for an image read from the cloud, dereferencing is more challenging because bytes are received sequentially and seek operations (closing a connection and sending a new HTTP range request) are costly. In Apache SIS, the data read from the cloud transit in a buffer of arbitrary size (currently 32 kilobytes). All pointers referencing data that are present in the buffer are dereferenced immediately. Other pointers are put on a waiting list and opportunistically dereferenced later, when the buffer is filled with new data. If, for answering a user's request, the value referenced by a pointer on the waiting list become mandatory, and if that value is far from the current stream position ("far" being an arbitrary concept), only then Apache SIS will perform a file seek or HTTP range request. However, it will consult the waiting list for opportunistically grouping nearby values with the same request.

A Cloud-Optimized GeoTIFF (COG) file is a GeoTIFF file organized following a small set of conventions. Some of those conventions are recommendations about the positions in the file of the data referenced by the pointers. Those recommendations aim to reduce the scattering of data, thus giving a chance for implementations to reduce the amount of seek operations or HTTP range requests that they will have to perform. But COG does not completely eliminate the effort that implementations must do.

The structure of a GeoHEIF file is different. Data are stored in boxes of various types, and some boxes can contain other boxes. As each box contains all its data, there are generally no pointers to arbitrary positions in the file except in a few cases designed for large data such as `ChunkOffsetBox` and `ItemLocationBox`. A box can be skipped if its content is not of interest. Skipping a container box skips also all boxes contained inside that container. This structure is easy to implement in an object-oriented language, because each type of box can be mapped to a class in the programming language. There is no need to implement something like a waiting list of pointers as in the GeoTIFF case. However, an inconvenience is that it is not always possible to jump directly to information of interest. A reader can only move from one box to the next in a sequential read, except for large data referenced by pointers in 'ChunkOffsetBox' and similar

special cases. But this inconvenience is mitigated when all the boxes to skip are contained in a larger box, since the latter can be skipped in one operation.

D.2. Binding to GDAL

Geomatys implemented a new "GDAL to Java" binding using the Panama technology (the `java.lang.foreign` package). Panama is available since Java 22 and replaces the Java Native Interface (JNI) which was used since Java 1.1. The official OSGeo GDAL-Java binding has not been used, because that project is based on SWIG which is basically a generator of JNI bindings. SWIG and JNA depend on JNI, while Panama is a new framework. One big advantage of Panama is that, if GDAL is pre-installed on the host computer, only the Java code needs to be distributed. There is no longer the need to bundle (or generate with SWIG) a C/C++ intermediate code between Java and the native library.

NOTE: With JNI, it was necessary to generate or bundle native code together with a Java application, in addition to the native library. This was because it was not possible to invoke a native function directly. Instead, JNI could only invoke a C/C++ intermediate function with a particular signature, which in turn invokes the native function. Distributing that intermediate native code with a Java application was a significant complication compared to the usual "write once, run anywhere" benefit. With Panama, having GDAL pre-installed on the host computer is sufficient. The rest of the application can be pure Java with no need for a C/C++ intermediate layer.

The Panama-based binding is available in Apache SIS 1.5 (not yet released a the time of writing this report) as the `org.apache.sis.storage.gdal` module. Contrarily to the OSGeo GDAL-Java binding, the SIS GDAL-Java binding is not a one-to-one mapping of GDAL functions and does not expose users to the GDAL's API. The binding is encapsulated into a SIS `DataStore` implementation and is generally not directly accessible by users. Only the functions needed by the `DataStore` implementation are bound, and their usages are closely integrated with Apache SIS's API. For example, immediately after loading the GDAL library, Apache SIS invokes the GDAL `CPLSetErrorHandler(…)` function for intercepting all GDAL error and warning messages. Then, any messages emitted by GDAL will be redirected to Apache SIS listeners or to Java loggers. This is transparent for users, who may not even be aware that they are using GDAL. For file formats that are supported by both GDAL and Apache SIS, such as GeoTIFF and GeoHEIF, which implementation is used can be guessed mostly through subtle behavioral differences. For example, metadata are different and the byte ranges visualization shown in Figure D.1 is available only with the SIS implementations.

At the time of writing this Report, the GDAL-Java binding provided by Apache SIS is read-only. Furthermore, the raster support is currently two-dimensional only. The latter restriction is because GDAL provides distinct APIs for two-dimensional data and for multi-dimensional data, for historical reasons. GDAL was initially a 2D library, and *n*-dimensional support was added later. Since SIS is multi-dimensional from the beginning, the addition of multi-dimensional support in a future version of the GDAL-Java binding should be transparent to Apache SIS users. For example, while 'GDALDataset::GetGeoTransform()' always returns a two-dimensional affine transform, that function is mapped to 'GridGeometry.getGridToCRS(…)' in Apache SIS, which supports arbitrary transforms (not necessarily linear neither two-dimensional). Whether GDAL's

two-dimensional of *n*-dimensional API is used is hidden behind Apache SIS's API. This is one of the reasons why Apache SIS binding does not expose GDAL's API directly.

D.2.1. Tiling

GDAL provides a `GDALDatasetRasterIO(…)` function for reading arbitrary sub-regions with arbitrary sub-samplings. If a user's request spans many tiles, that function takes care of reading all needed tiles and assembling the result in a single output array. Alternatively, GDAL also provides the `GDALReadBlock(…)` function for reading on a tile-by-tile basis, which is potentially more efficient. Apache SIS uses the former function, but requests only regions that should correspond to a whole tile or an integer number of tiles. These requests are computed by the same `TiledGridCoverage` Java class as the one used by the pure-Java GeoTIFF and GeoHEIF implementations (Annex D.1), using the tile size given by `GDALGetBlockSize(…)`. This strategy relies on the assumption that `GDALDatasetRasterIO(…)` will detect that the requests match the tile layout and therefore can be executed as efficiently as if `GDALReadBlock(…)` were invoked.

NOTE: Apache SIS does not use `GDALReadBlock(…)` directly because the natural block size of the data set, as given by `GDALGetBlockSize(…)`, is sometime an inefficient way to tile an image. In particular, some data sets use block sizes of *w*×*n* pixels where *w* is the full raster width and *n* a small number, often 1. This is called "strip" in GeoTIFF. When an inefficient natural block size is detected, Apache SIS uses a more efficient tile size instead, at the cost of more work for `GDALDatasetRasterIO(…)`.

The GDAL-Java binding uses the tile cache inherited by Apache SIS `TiledGridCoverage` class. The GDAL tile cache is not enabled by default. The Apache SIS tile cache is based on soft references (a mechanism that supports detecting when a tile is not referenced anymore by the client application). The C/C++ smart pointers would have some similarities (while not fully equivalent) but are not used by the GDAL raster API.

D.3. Differences between GDAL and Apache SIS

As mentioned in Annex D.1, GDAL and Apache SIS differ in their implementation strategy. But the two projects differ also in their behavior. In particular, the two projects do not return the same data for a given request.

D.3.1. Differences in extent of returned data

When the user requests a subregion of a raster, GDAL returns the raster data for exactly the requested subregion. This is necessary because `GDALDatasetRasterIO(…)` copies the raster data in an array allocated by the user. Apache SIS does the opposite: It allocates the array itself and returns the raster data encapsulated in data structures of the `java.awt.image` package. By leveraging the abstraction provided by those data structures, Apache SIS takes the liberty of returning more data than requested if doing so is more efficient. In particular, Apache SIS generally rounds a user request to an integer number of tiles, which enables caching the tiles,

return them directly (without copy) and reuse them (again without copy) if needed in subsequent requests. Compared to GDAL, it is as if the `GDALRasterBlock::GetLockedBlockRef(…)` function (the function for accessing directly the internally cached blocks) was used transparently for all I/O operations with Apache SIS, but without the need to lock/unlock tiles on user's side.

The fact that users may receive something different than what they requested is partially transparent. This is because the image coordinate system is adjusted for having pixel coordinates (0,0) at the beginning of users' request. In other words, the upper-left corner may start at negative pixel coordinates if the image is larger than requested. Users can ignore this detail for some operations such as accessing pixel values or rendering the image. See, for example, listing E.4 in GIMI Lessons Learned and Best Practices Report [OGC 24-040] Annex §E.2.2.1.

D.3.2. Differences in zoom levels

Since users may request an image of any size, `GDALDatasetRasterIO(…)` may resample the image for fitting in the requested region. If the resampling method is nearest neighbor and a pyramid of images at different resolutions is available, GDAL may use a pyramid level slightly less accurate than the requested resolution with a tolerance factor of 1.2. Apache SIS is more conservative: It may apply subsampling but never resamples during I/O operations (resampling is available as a separated operation) and unless there is no such data always selects a pyramid level providing at least the requested resolution. The cost of this strategy is potentially slower I/O operations if Apache SIS selects a finer pyramid level than GDAL This difference is not always visible at rendering time.

D.3.3. Differences in georeferencing

Both GDAL and Apache SIS use affine transforms for describing the conversions from pixel coordinates to CRS coordinates. However, the details differ in three ways.

D.3.3.1. Difference in the type of transform

`GDALGetGeoTransform(…)` returns the six coefficients of a two-dimensional affine transform. Apache SIS returns the information in a more indirect way. The transform can be anything, with any number of dimensions and not necessarily linear (for example, it may be a transform doing interpolations in a grid of ground tie points). In the common case where the transform is affine and two-dimensional, it will be an instance of the `java.awt.geom.AffineTransform` standard class. The 6 coefficients can be extracted from that class using the standard Java2D API.

D.3.3.2. Difference in the "cell center/corner" convention

The affine transform returned by `GDALGetGeoTransform(…)` always maps pixel corners. In Apache SIS, whether the transform maps pixel corners or pixel centers is specified by the user. This is designed that way because the choice depends on the calculation which will use the transform and not on the data (see GIMI Lessons Learned and Best Practices Report

[OGC 24-040] annex §E.1.1). The translation between the user's request and the pixel center/corner convention actually used by the data is handled by Apache SIS.

D.3.3.3. Difference in CRS axis order

The affine transform returned by `GDALGetGeoTransform(…)` does not always compute geographic coordinates in the order declared by the CRS axis definitions. GDAL inserts a *"data axis to SRS axis mapping"* step between the affine transform and the CRS. The mapping is defined by GDAL's heuristic rules aimed at determining (longitude, latitude) axis order. This extra step can be seen in the following functions of GDAL C/C+&#43 API:

- `OGRSpatialReference::setAxisMappingStrategy(…)` for specifying whether to apply heuristic rules.
- `OGRSpatialReference::GetDataAxisToSRSAxisMapping()` for fetching the result of GDAL heuristic rules.

One way to see GDAL's design is to state that, when the *"axis mapping strategy"* enables heuristic rules, GDAL has an *effective CRS* which differs from the *actual CRS* encapsulated by `OGRSpatialReference`. The matrix returned by `GDALGetGeoTransform(…)` maps to the effective CRS, not to the actual CRS. Conversions from effective to actual CRS shall use the information provided by `GetDataAxisToSRSAxisMapping()`. The latter typically returns {2,1} for a geographic CRS, which represents an axis swapping.

The *"data axis to SRS axis mapping"* step does not exist in Apache SIS. Apache SIS relies entirely on the mathematics of the affine transform. If axis swapping is needed, this is handled by swapping rows in the affine transform matrix. Therefore, Apache SIS users can always rely on the fact that the transform maps directly to the CRS axes as they are defined by the CRS, no matter the axis order.

> ***CAUTION***
>
> *The GDAL "data axis to SRS axis mapping" step was introduced in GDAL 3, possibly as a way to reduce disruptive changes for users of GDAL 1 and 2 where (longitude, latitude) axis order was assumed. Newer libraries that do not have such historical constraint should avoid introducing such step, as it is a significant complication. Instead, libraries should either swap matrix rows or derive a CRS definition that matches their desired axis order. For example, a GeoTIFF reader decoding EPSG:4326 can create a `DerivedGeographicCRS` with (longitude, latitude) axis order as suggested by case 3 of OGC directive #14.*

E

ANNEX E (INFORMATIVE) SILVEREYE TECHNOLOGY — OPEN SOURCE CONTRIBUTIONS

ANNEX E (INFORMATIVE) SILVEREYE TECHNOLOGY — OPEN SOURCE CONTRIBUTIONS

E.1. Executive Summary

Silvereye Technology made contributions to a range of open source projects as part of the D120 development activity. These included code contributions to:

- libheif, which is the basis for much of the HEIF support in other open source software;
- GDAL, which is a very widely used geospatial file format abstraction; and
- GPAC, which provides a range of imagery and video tools, especially well-suited to development and debugging.

These contributions progress the ecosystem support, reducing GIMI implementation effort in the future.

E.2. libheif contributions

> libheif is an ISO/IEC 23008-12:2017 HEIF and AVIF (AV1 Image File Format) file format decoder and encoder. There is partial support for ISO/IEC 23008-12:2022 (2nd Edition) capabilities.
>
> HEIF and AVIF are new image file formats employing HEVC (H.265) or AV1 image coding, respectively, for the best compression ratios currently possible.
>
> — https://github.com/strukturag/libheif/

libheif is a widely used implementation of the ISO/IEC 23008-12:2022 image format, which is part of the basis for GIMI.

Silvereye Technology previously contributed to libheif, and the Testbed 20 activity provided an excellent opportunity to extend and improve the capabilities of libheif for both the GIMI profile, and indirectly for other users. A full list of the feature improvements and bug fixes can be seen

in the libheif repository however a sample of the more interesting improvements is described in this section.

An important focus for GIMI is the introduction of the uncompressed codec to HEIF. The uncompressed codec as defined in ISO/IEC 23001-17:2024 allows encoding of high bit depth signed and unsigned integer values, floating point values, complex values. This includes other "closer to the sensor" data, such as filter array outputs (a.k.a. "raw" or Bayer imagery), sensor non-uniformity corrections, and bad pixel masks. Silvereye Technology had previously provided a partial implementation of this capability in libheif. This implementation was extended and updated during Testbed 20 to make the implementation cleaner and more flexible.

As a refinement to the uncompressed codec, Silvereye Technology implemented read support for generic compression, which is proposed as ISO/IEC 23001-17 Amendment 2. This provides lossless compression of data using common algorithms such as Zlib, Deflate, and Brotli. In addition to providing an early implementation, Silvereye Technology also worked with MPEG participants to enhance the proposal prior to implementing the refined design. This work was merged into the libheif code base as an experimental capability, pending approval of the proposed Amendment.

As discussed in OGC Testbed 20 GIMI Benchmarking Report OGC (24-041) , there can be substantial overhead in a HEIF file that is a single, small tile. Silvereye Technology implemented the proposed Low-overhead image file format variant of HEIF as an experimental capability within libheif. At the time of writing this Report, the base code and AV1 codec were incorporated, and the HEVC codec was in review. This demonstrated interoperability with the libavif implementation of this format. The potential advantages are expanded on in OGC Testbed 20 GIMI Benchmarking Report OGC (24-041).

Silvereye Technology acknowledges the work by Dirk Farin Algorithmic Research in review of all of these contributions and the subsequent significant improvements to this work and especially wishes to express appreciation for the collaborative working relationship.

E.3. GDAL contributions

> GDAL is a translator library for raster and vector geospatial data formats that is released under an MIT style Open Source License by the Open Source Geospatial Foundation. As a library, it presents a single raster abstract data model and single vector abstract data model to the calling application for all supported formats. GDAL also comes with a variety of useful command line utilities for data translation and processing.
>
> — https://gdal.org/en/latest/

GDAL is used in a wide range of open source and proprietary geospatial applications, including QGIS and Esri products.

Silvereye Technology contributed features to GDAL/OGR as part of the Testbed 20 activities. The key outcomes of this work exposed key functionality from libheif to the wide range of

software that uses GDAL, and to validate the GeoHEIF design. Together, these contributions bring HEIF closer to becoming a mainstream geospatial format.

Prior to Testbed 20, GDAL supported access to HEIF files through a libheif-based raster driver. While functional, it was limited to read-only access, and the entire file had to be decoded to access any part of that image.

Based on libheif tiling work conduct by Dirk Farin Algorithmic Research (see Annex B), Silvereye Technology adapted the GDAL implementation to support tiled access to HEIF files using a range of tiling formats, including gridded items, codec-internal tiling (as used in the uncompressed codec), and in the low-overhead "tili" format. Tile access provides a much more efficient way to access part of a large image, with reduction in the data transferred, reduction in the data decoded and hence lower latency for the consumer.

Silvereye Technology also provided an implementation of the GeoHEIF metadata as described in the Testbed 20 GIMI Specification Report [OGC 24-038] for reading in both libavif and libheif based drivers.

Silvereye technology also provided write support for HEIF files, so that software using GDAL can output files in this format.

Additional work in progress, but not yet merged, include updates to provide enhanced overview support.

E.4. libavif contributions

libavif is an open source implementation of the AV1 Image File Format (AVIF). AVIF is the HEIF structure using an AV1 codec.

Some of the capabilities provided by libavif are also provided by libheif. Key differences are that libavif only supports various implementations of the AV1 codec, while libheif supports both AV1 and other codecs such as HEVC (H.265), AVC (H.264), JPEG, JPEG-2000, and VVC (H.266). libavif provides some capabilities not present in libheif, such as High Dynamic Range (HDR) images.

Silvereye Technology identified that libavif did not provide the necessary API to implement the GeoHEIF design. While libavif is able to parse the required properties, those properties are discarded. This was identified to the libavif project at Issue 2395. That subsequently resulted in a change by Yannis Guyon (Google, and part of the libavif core team) at Pull Request 2420 which has now been merged. Silvereye Technology followed on from this by extending libavif to support writing of those properties, which is required to enable future GeoHEIF support in GDAL and other geospatial tools that use libavif.

Silvereye Technology acknowledges the support provided by the libavif team, and especially the work by Yannis Guyon, particularly for Pull Request 2420 and review for additional properties support.

E.5. GPAC contributions

GPAC (notionally GPAC Project on Advanced Content) is an open-source project providing a range of tools for audio and video content. The project team is part of MPEG and has a strong focus on ISO Base Media File Format (ISOBMFF). Key tools include the “gpac” pipeline engine, the MP4Box packaging / transcoding tool, and the mp4box.js javascript implementation.

Silvereye Technology made contributions to both the C and javascript projects. These included update to the MP4Box tool to display the new boxes used for generic compression (see ISO/IEC 23001-17 Amendment 2 discussion earlier in this Annex). These changes provide debug capabilities and are used in the ISOBMFF file format conformance suite.

E.6. GStreamer contributions

> GStreamer is a library for constructing graphs of media-handling components. The applications supported range from simple Ogg/Vorbis playback, audio/video streaming to complex audio (mixing) and video (non-linear editing) processing.
>
> Applications can take advantage of advances in codec and filter technology transparently. Developers can add new codecs and filters by writing a simple plugin with a clean, generic interface.
>
> — https://gstreamer.freedesktop.org/

Silvereye Technology extended the existing ISOBMFF multiplexer (“isomp4mux”) to better support the needs of GIMI and GeoHEIF. The primary feature was extending the interface and implementation to support uncompressed (ISO/IEC 23001-17) video in using a range of colour spaces, interleave formats, and subsampling. The formats include “IYU2”, “RGB”, “BGR”, “NV12”, “NV21”, “RGBA”, “ARGB”, “ABGR”, “BGRA”, “RGBx”, “BGRx”, “Y444”, “AYUV”, “GRAY8”, “GRAY16_BE”, “GBR”, “RGBP”, “BGRP”, “v308”, and “r210”. The implementation is generic, and additional formats can be supported without significant rework.

In support of ISO/IEC 23008-12 image sequences, Silvereye Technology implemented an additional multiplexing mode that switches the result from MP4 video to image sequence compliance. This was tested as interoperable with the libheif implementation (intraframe only).

Silvereye Technology also identified and corrected bugs in related parts of GStreamer, including an error in how H.264 (AVC) was encoded and parsed, and fixes in the uncompressed demultiplexing (“qtdemux”) code, including support for a single monochrome plane in component interleave mode.

Silvereye Technology acknowledges the support provided by Sebastian Dröge of Centricular.