FRF - The Flexible Raster Format

Draft 1.0 Specification

Last Modified On:
December 8, 2017

FRF - The Flexible Raster Format

Document Description

This specification defines the *Flexible Raster Format*. This is a format for storing raster image data (as opposed to vector image data) in various integer and floating-point types and depths. One of the core distinguishing characteristics of FRF is the separation of raster data from its visualization, making it possible to use FRF to exchange meaningful raster data, instead of just a specific way of rendering the data. FRF supports arbitrary numbers of layers, and standard raster format features like an alpha channel. It also supports some more unique features like arbitrary bit-depth integer formats (appropriate for raw data from image sensors), per-layer validity masks, and simple geo-tagging and geo-referencing.

Contents

1 Introduction - Why Another Image Format?

When seeing a specification for a new raster image format the first natural (and correct) reaction is to ask: Why do we need another image format? After all, there are plenty of open standards for raster images already, many with freely available libraries with which to use them. The only reasonable answer is that the new format must address deficiencies in the existing standards. Thus, we guide this discussion by exploring the problems with existing formats.

A Format For Data, Not Just Pictures

All raster image formats currently in common use (including JPEG, TIFF, BMP, GIF, and PNG) are display-centric. That is, they embrace the singular goal of defining how to re-create a given visual pattern. This typically involves expressing the greyscale intensity at each pixel using a pre-defined scale (e.g. $0 - 255$) or the color and intensity of each pixel using some known colorspace (e.g. RGB). This approach is generally fine where re-creating a desired image is the only goal. However, there are times when raster data has some implicit meaning that is not directly related to visualization. For instance, imagine that we want to communicate a height or depth image. Each pixel has a value associated with it that represents a distance in some units. To save this data in an existing format requires mapping the range of distances to a pre-defined pixel intensity range. Then, to correctly interpret the image later, we need to know what mapping we used so that we can invert it. There is generally no standard way of encoding such a mapping in the image itself, so it must be separately communicated to the entity trying to interpret the image. One may be tempted to have a fixed, hard-coded mapping, but while this alleviates the need to store and communicate the mapping separately from the raster data, it frequently wastes much of the available dynamic range in your pixel representation on unused values and introduces quantization noise. Not only does this deficiency add complexity to data interchange, but it makes proper data interpretation more error-prone because the key needed to map back from pixel intensities to meaningful values can become separated from the corresponding image file.

FRF addresses this issue by taking a data-centric approach instead of a display-centric approach. Instead of defining pixel intensities directly, you can assign arbitrary values to each pixel using a variety of numeric formats and bit depths. Then, separately, you define a *visualization* that expresses how to render your values in an RGB colorspace. In our previous example of trying to save a height or depth map, we would save the raw distance values associated with each pixel (perhaps as a floating-point number). Then our visualization would tell us to associate the color black with a given distance and the color white with another distance. The distances in between will implicitly be mapped to different shades of grey and distances outside the specified range will saturate to white or black. From a standpoint of rendering a greyscale image, this is no different than using an existing image format. The difference, however, is that the "actual" distance values are still accessible in the FRF file, whereas if we use another format, we only have access to pixel intensities and we have to try to reconstruct the original distance values from that.

The Fixed Bit-Depth Problem

Another common problem with existing raster image formats is the use of fixed, hard-coded bit depths. For instance, the JPEG image format can only use 8-bit unsigned integers to represent the intensity of a single pixel or 8 bits per red, green, and blue color channel for color images (modern revisions also support a few fixed higher bit-depths, although these are not universally supported). Many imagers in use today have higher bit-depth analog to digital converters (12 and 14-bit sensors are common, and 16-bit sensors are not unheard of). To save such data with a fixed-depth format, you must truncate down to the depth of the format by removing some number of bits[1]. This caps your dynamic range and frequently results in non-reversible saturation. Even the formats that offer a choice of bit depths do not solve this problem entirely because you must generally choose between a small handful of options. This forces users to round up to the nearest allowed depth, padding the values at each pixel to fill out the unused bits. Since the padding is required for

[1]Usually you drop the least-significant bits because these are the noisiest.

every pixel this wastes an appreciable amount of space. The fixed bit-depth limitation is frequently what drives professional photographers to use proprietary *raw* image formats when acquiring images.

FRF eliminates this limitation by providing a host of numeric formats to choose from when saving raster data. In addition to offering floating-point layers, FRF supports arbitrary bit-depth (up to 64 bits) unsigned integer formats. Values are not padded to consume an integer number of bytes. This allows for the compact representation of raw sensor data, regardless of the sensor depth.

Is Storing N Channels Really Harder Than Storing 3?

Many existing raster image formats allow you to specify greyscale images using 1 channel or color images using 3 channels (perhaps 4 with an alpha layer). While there are some exceptions (notably, TIFF), these are frequently the only options. Unfortunately, there are many situations where this is not sufficient. The biggest problem with this arises when trying to save multi-spectral or hyper-spectral imagery. Such an image consists of many layers, each corresponding to a specific spectral band. Using a format that locks your number of channels to 1 or 3 requires you to save each spectral band (or groups of 3 bands) as separate images.

FRF effectively removes this restriction and allows you to save up to 2048 separate channels. The numeric type is specified per channel, so layers of different types and depths can be saved together in a single file.

Layer Validity

Raster image formats generally only allow for rectangular images. Unfortunately, data doesn't always come in square packages. When we have irregularly shaped images, we need to pack them into larger rectangular rasters. With a conventional raster format we need to use some default value when filling in "unused" pixels. With existing formats, it is impossible to distinguish between "dummy" pixels that were assigned the default value and legitimate pixels that just happened to already have that same value. In other words, we can't tell which pixels are "valid". This is a consequence of the display-centric nature of existing formats. Since these formats focus on what to display for each pixel, and you can only display a fixed set of colors (you can't display *nothing*), it doesn't make sense to talk about the validity of a pixel. In other words, since you have to display something for each pixel, every pixel is valid by definition. The data-centric nature of FRF, on the other hand, assumes we are visualizing some type of data, and that data may or may not be defined for a given pixel. While we will need to display something for undefined pixels, we can still keep track of pixel validity. FRF has a mechanism for marking pixels as invalid, or undefined. For floating-point layers, the value NaN is understood to have this meaning. For integer layers, one can specify an optional mask layer. A mask layer uses 1 bit per pixel to mark which pixels in the layer are valid and which are not.

Layer validity is a distinct concept from transparency. FRF supports an optional global alpha channel. This is separate from layer validity masks.

The Geo-Tagging and Geo-Registration Mess

Frequently imagery is associated with particular locations and times. It is very handy to be able to embed this type of information in an image. There are two types of such metadata and it is important to distinguish between them. Sometimes, we know the location and/or orientation of a camera when we take a picture. When we attach this information to an image we call this "Geo-Tagging" or "Pose-Tagging". At other times, we will literally associate each distinct pixel in an image with a location on Earth. This is called "Geo-Registration". Importantly, if imagery is Geo-registered, one cannot compute a pose tag from the registration data (in fact the data might not correspond to a real-world camera so a pose tag may not even make sense). Also, if an image is Geo-tagged, the tags (even if position and orientation are provided) are not enough to Geo-register the imagery without additional information, which may or may not be known. Thus, Geo-tagging and Geo-registration are not equivalent, even though the terms are often sloppily used interchangeably. Many raster formats do not provide any mechanism for Geo-tagging or Geo-registration. Those that do often support one but not the other. JPEG, for instance, allows for Geo-tagging with camera position (orientation is not supported) through EXIF metadata. Geo-registration, on the other hand, is not supported. TIFF is special amongst existing formats because it supports Geo-tagging (position only) through EXIF metadata as well as Geo-registration through a separate Geo-TIFF extension. The issue with this is that the Geo-TIFF standard is a mess. It provides several ways of tying raster data to a planetary coordinate system, including the use of two tie points, one tie-point and scale-factors, and multiple tie points using an unspecified interpolation scheme. It also supports countless planetary coordinate systems to choose from, and ways of specifying custom systems in case your favorite system isn't on the list. It even lets users specify their own units for their preferred coordinate systems. The format was apparently designed to make it as easy as possible to *create* a Geo-TIFF image. Unfortunately it places an enormous burden on the software responsible for *reading* Geo-TIFF images since it must be able to correctly process the embedded registration data and convert it to whatever format is used internally by the software, regardless of the choice of representation in the Geo-TIFF file. Since there are so many different ways of encoding registration data with Geo-TIFF, the unfortunate situation is that there are really no complete implementations of the standard. It is common for standard-compliant Geo-TIFFs to open just fine in one program and then end up on the other side of the Earth in another program because it chose an unsupported combination of options. The only *safe* Geo-TIFFs use the simplest, most standard, lowest-common-denominator options.

FRF supports optional Geo-Tagging as well as optional Geo-Registration. Both features are implemented with the philosophy that there should be exactly one way to embed the associated metadata, and that way should be as simple and straightforward as possible. This splits the burden between supporting the *creation* of FRF files and supporting the *reading* of FRF files.

Visualization Consistency

The FRF format permits high-bit-depth and floating-point imagery. It also makes a distinction between the value of a pixel and the way that value is visualized or rendered. It is important when making such a distinction that we don't just focus on the data and ignore the visualization. For instance, OpenEXR is a high-dynamic range format with support for floating-point images. OpenEXR is sometimes chosen to exchange raster data when high dynamic range is required. Unfortunately, this format does not explicitly specify how the values attached to a given pixel should be rendered in grayscale or color and different viewers use different algorithms for deciding that rendering. The result is that someone can produce an image in the OpenEXR format and give that file to someone else, and have that person see a completely different image when they view the image with their preferred software.

An FRF file *always* includes a default visualization. Additional visualizations can (optionally) be provided.

Even if an FRF file encodes data that is not specifically meant for human viewing, there is a guarantee that when two people open the same FRF file with two different standard-compliant viewers, they will see the same thing.

2 FRF Overview

FRF is a raster image file format for saving data in various floating-point and integer formats, including arbitrary bit-depth integers. FRF makes a distinction between data and visualizations of data. This makes FRF a good choice when data has meaning, and possibly units, attached to it that you don't want to be lost when saving the data to a file. FRF supports an arbitrary number of layers, global alpha, Geo-tagging and Geo-Referencing, and several other features. In the initial specification, FRF does not support compression, although compression support may be added in a future version of the specification.

FRF starts with a file header. This contains general properties of the image and a manifest of included layers. This also includes all visualizations. The layers follow the file header, each layer padded out on the end to contain an integer number of bytes. Layers do not contain their own headers. All information about the image layers is saved in the layer manifest in the file header. The file header begins with some simple, mandatory information, and then a series of information blocks. There are several different types of information blocks, each with a pre-defined structure. Some blocks are mandatory (like the block holding the layer manifest), while other blocks are optional (like blocks for holding pose-tagging or Geo-registration data).

Before defining the header structure and listing the information blocks, we need to go over some of the basics of image coordinates and data types.

3 Image Coordinates

FRF is a 2D raster format. It discritizes a two-dimensional rectangular region into "pixels" and assigns values to each pixel. When such an image is viewed in a standard way, such as through an image viewer, there will be an upper-left, upper-right, lower-left, and lower-right corner for the image. In FRF we adopt the convention that the upper left pixel in the image is the origin of the 2D image coordinate system. The X axis is horizontal pointing towards the upper-right corner. The Y axis is vertical and points down to the lower-left corner. This coordinate system will be used later in the document in various places. Whenever we refer to a pixel using coordinates, this is the implicit coordinate system that we are using. Figure 1 depicts the image coordinate system graphically.

Sometimes we may refer to a single row or column of an image by number. When we refer to row n, we are referring to all pixels in the image with coordinates $(*, n)$, starting with pixel $(0, n)$. When re refer to column m, we are referring to all pixels with coordinates $(m, *)$, starting with pixel $(m, 0)$.

When dealing with real-valued image coordinates, point (x, y) corresponds to the center of pixel (x, y). Non-integer coordinates are understood to linearly map to points in between the centers of adjacent pixels. For instance, point $(3.5, 7)$ corresponds to a point exactly half way between pixels $(3, 7)$ and $(4, 7)$. In this way, real-value image coordinates are just an extension of integer-valued pixel coordinates.

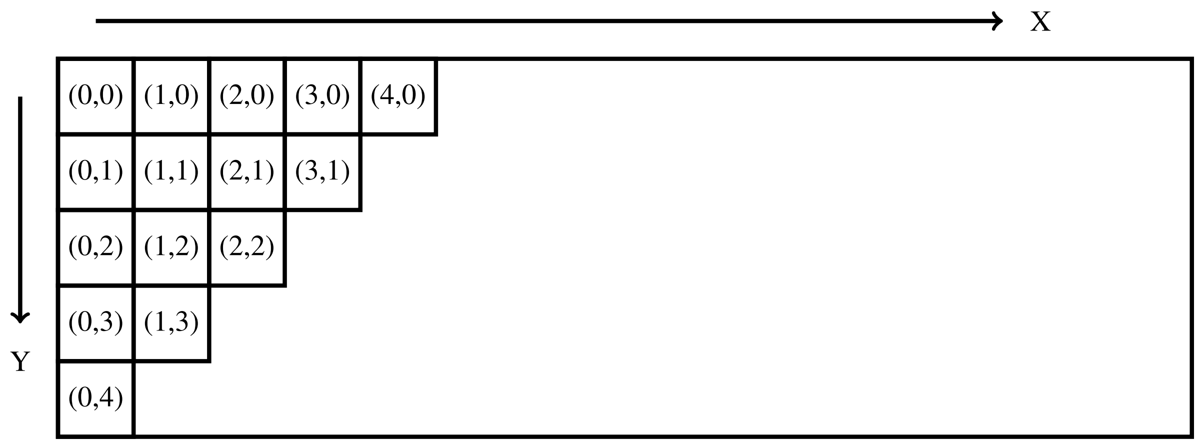

Figure 1: Image coordinate system

4 Data Types

FRF supports storing image data using common floating-point and signed/unsigned integer types, as well as less-common arbitrary bit-depth integer types. Table 1 lists the supported standard data types. Each type has a unique (unsigned integer) type code associated with it. Each layer represents data using one of the standard types. Different layers may use the same or different types. Each layer has two coefficients attached to it, α and β, which are always 64-bit floating point numbers. These are used when interpreting the value of a pixel at a given location in a given layer. There are two definitions to keep in mind:

- **Raw Value:** This is the number represented by the bit pattern at a given location in a given layer, interpreted according to the data type for the layer. For instance, if you choose Int8, the raw value will be an integer between -128 and 127. If you choose Uint17 (17-bit unsigned int), the raw value will be an integer between 0 and $2^{17} - 1$.
- **Value:** Let x be the *raw value* for a given pixel in a given layer. The *value* is defined to be $\alpha x + \beta$.

While one can get their hands on both the value and the raw value when decoding an FRF image, the raw value should only be viewed as a temporary quantity used in computing the value. There may be good reasons to occasionally operate on raw values since one may be able to avoid some casts and arithmetic in some cases. This is permitted, but one should remember that the value is the important quantity, so any operations done on raw values must be done in such a way that they have the desired effect on the values (after applying α and β).

This encoding scheme allows you to represent real numbers in any desired range using essentially any desired resolution by selecting α, β, and the correct depth unsigned integer type to go along with it. One may observe that this scheme makes signed integer support somewhat redundant, since the same range of values can be expressed by scaling and translating an unsigned integer. However, we deliberately support 2's-compliment signed integers with standard depths for usability. If users have signed integer raster data, they can skip the unnecessary conversions needed to use a scaled integer representation.

Type Code	Type Name	Description
0		Reserved for future use
1	Uint1	Unsigned integer, 1 bit
2	Uint2	Unsigned integer, 2 bits
3	Uint3	Unsigned integer, 3 bits
⋮	⋮	⋮
64	Uint64	Unsigned integer, 64 bits
65	Int8	Signed (2's-compliment) integer, 8 bits
66	Int16	Signed (2's-compliment) integer, 16 bits
67	Int32	Signed (2's-compliment) integer, 32 bits
68	Int64	Signed (2's-compliment) integer, 64 bits
69		Reserved for future use
70	Float32	IEEE 754 floating-point number, 32-bits
71	Float64	IEEE 754 floating-point number, 64-bits

Table 1: Standard data types

In addition to these standard types, FRF defines a set of compound data types. These are not used for storing image data, but they are used for storing metadata. That is, they are used in the image header to store things like camera calibration and pose information. Table 2 lists the compound types supported by FRF.

Type Code	Type Name	Description
75	Vec2	2-element vector of 64-bit floats
76	Vec3	3-element vector of 64-bit floats
77	Vec4	4-element vector of 64-bit floats
78	Mat2	2-by-2 matrix of 64-bit floats
79	Mat3	3-by-3 matrix of 64-bit floats
80	Mat4	4-by-4 matrix of 64-bit floats
81	MatX	N-by-M matrix of 64-bit floats
82	String	UTF-8 encoded string
83	GPST	GPS Time

Table 2: Compound data types

4.1 Storage of Primitive and Compound Types

We have given names and type codes to a large set of different data types. In this section we define how these different types are serialized for storage in a file. We begin with a short discussion of some background material.

In a computer the atomic unit of memory is the byte. A single byte consists of 8 bits, numbered 0 through 7. Each bit can take on two values, 0 or 1. Bit 0 is called the least-significant bit (LSb) and bit 7 is called the most-significant bit (MSb). Memory is organized as a large array of bytes, each byte with its own memory address. When data is stored in a file, one writes a sequence of full bytes. Bytes in a file can also be numbered, with byte 0 being the first byte written.

Because the byte is the atomic unit of memory, it is quite common for data types to take up an integer number of bytes. For instance, an ASCII character takes up a single byte. A single-precision floating point number takes exactly 4 bytes. This simplifies data retrieval because you only have to identify which bytes contain your data and you don't need to worry about your data taking up only part of some bytes. This is not usually a problem. However, when saving image data where the value for a pixel does not natively occupy an integer number of bytes, we end up converting the value to a larger type, which can waste a significant amount of space. For instance, if we have 10-bit values for our pixel data, we end up padding each value and saving them as two-byte unsigned integers. This wastes 6 bits per pixel, which can end up being a substantial amount of space for large images.

All of our compound types and many of our standard types use integer numbers of bytes. In the header for an FRF file we exclusively use such data types. This simplifies data retrieval in these areas. However, we do allow for data types that use fractional bytes when storing image data. We must carefully define how this data is organized.

In the header, all data is byte-aligned and all data types take up an integer number of bytes. All that must be done to specify how data is stored in the file is specify byte ordering for each type. For primitive standard types FRF uses Big-Endian representation. That us, when storing an item that consists of multiple bytes, the most-significant byte (MSB) is stored first, towards the front of the file, and the least-significant byte (LSB) is stored last, towards the end of the file. Compound types are composed of collections of objects of standard types. For these items, each field is stored most-significant byte first. We specify the ordering of the fields in the next sub-section.

4.1.1 Compound Type Definitions

$\mathbf{v}_1$	$\mathbf{v}_2$
8	8

Vec2 Serialization This shows the serialization of a 2-vector denoted $\mathbf{v}$. All fields are of type Float64. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$
8	8	8

Vec3 Serialization This shows the serialization of a 3-vector denoted $\mathbf{v}$. All fields are of type Float64. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$	$\mathbf{v}_4$
8	8	8	8

Vec4 Serialization This shows the serialization of a 4-vector denoted $\mathbf{v}$. All fields are of type Float64. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

$\mathbf{M}_{11}$	$\mathbf{M}_{12}$	$\mathbf{M}_{21}$	$\mathbf{M}_{22}$
8	8	8	8

Mat2 Serialization This shows the serialization of a 2-by-2 matrix denoted $\mathbf{M}$. All fields are of type Float64. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

$\mathbf{M}_{11}$	$\mathbf{M}_{12}$	$\mathbf{M}_{13}$	$\mathbf{M}_{21}$	$\mathbf{M}_{22}$	$\mathbf{M}_{23}$	$\mathbf{M}_{31}$	$\mathbf{M}_{32}$	$\mathbf{M}_{33}$
8	8	8	8	8	8	8	8	8

Mat3 Serialization This shows the serialization of a 3-by-3 matrix denoted $\mathbf{M}$. All fields are of type Float64. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

$\mathbf{M}_{11}$	$\mathbf{M}_{12}$	$\mathbf{M}_{13}$	$\mathbf{M}_{14}$	$\mathbf{M}_{21}$	...	$\mathbf{M}_{24}$	$\mathbf{M}_{31}$	...	$\mathbf{M}_{34}$	$\mathbf{M}_{41}$	...	$\mathbf{M}_{44}$
8	8	8	8	8	...	8	8	...	8	8	...	8

Mat4 Serialization This shows the serialization of a 4-by-4 matrix denoted $\mathbf{M}$. All fields are of type Float64. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

Rows	Cols	$\mathbf{M}_{11}$	$\mathbf{M}_{12}$	...	$\mathbf{M}_{1N}$	...	$\mathbf{M}_{M1}$	$\mathbf{M}_{M2}$	...	$\mathbf{M}_{MN}$
4	4	8	8	...	8	...	8	8	...	8

MatX Serialization This shows the serialization of an M-by-N matrix denoted $\mathbf{M}$. Fields "Rows" and "Cols" are of type Uint32. All other fields are of type Float64. An empty matrix of this type will consist of exactly 8 bytes: 4 to encode the number of rows (0) and 4 to encode the number of columns (0). The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

StrLength	B0	B1	B2	...
4	1	1	1	...

String Serialization This shows the serialization of a string. The "StrLength" field is of type Uint32 and holds the number of bytes that follow in the string. The full string object consists of $4+$StrLength bytes. The bytes that follow the "StrLength" field should be interpreted as a UTF-8 byte stream. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

Week	TOW
4	8

GPST Serialization This shows the serialization of a time object. Times are saved as unambiguous GPS time. Week rollovers should be corrected for before saving a time object. The field "Week" has type Uint32. The field "TOW" has type Float64. "Week" is the number of integer weeks that have passed since the GPS time-0 epoch (a week being defined as exactly $7 * 24 * 60 * 60$ seconds). "TOW" is the number of seconds into the remaining week. The top row provides field names and the bottom row provides field widths (in bytes). The leftmost field is closest to the front of the file.

4.1.2 Storage of Image Data

In the preceding parts of this section we defined how data types that consist of an integer number of bytes are organized in an FRF file. FRF also has some simple types that do not occupy integer numbers of bytes. These types may only be used when storing image data. We also note here that every layer in an FRF file is padded at the end with the smallest number of 0 bits possible to make the layer occupy an integer number of bytes. Thus, even if one of these types is used for storing the data in a given layer, the layer itself will be byte-aligned, even though individual values in the layer may not be.

When writing image data for a layer, the start of the layer is always byte-aligned. We write the bit patterns for each pixel, one after another, row-by-row (starting with pixel $(0, 0)$) until all values for the layer have been written. When the first value is written, it is aligned with the MSb of the first byte of the buffer. If the data type for the layer uses less than 8 bits per value then there will be left over bits (the least significant bits) in the first byte. If the data type uses more than 8 bits, the value must spill over into next byte (using the most significant bits first). The second value will start with the most significant unused bit. Whenever a value rolls into the next byte, it is aligned with the most significant bit of that byte. Figure 2 shows an example of a Uint10 layer with 6 pixels (2 rows by 3 columns).

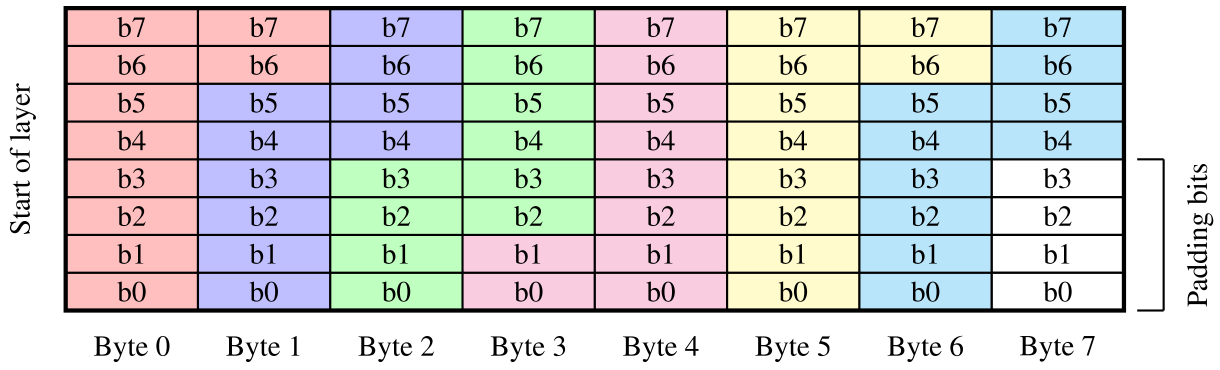

Figure 2: This example shows the serialization of a Uint10 layer consisting of 6 pixels (2 rows by 3 columns). Each color corresponds to a single 10-bit value for one pixel (Red = pixel (0,0), Blue = (0,1), Green = (0,2), Magenta = (1,0), Yellow = (1,1), Cyan = (1,2)). We write the most significant bits of each value first. In this example, we will refer to the value for pixel (0,0) as value 0. Bit b9 of value 0 is placed in bit b7 of byte 0 for the layer. Bit b8 of value 0 is placed in bit b6 of byte 0, and so on. Bit b2 of value 0 is placed in bit b0 of byte 0. Then bit b1 of value 0 is placed in bit b7 of byte 1 of the layer, and bit b0 of value 0 is placed in bit b6 of byte 1. The next value starts in bit b5 of byte 1.

With this method of packing data, we always align with the most significant unused bit of the lowest-number byte. In this situation, it can be convenient to think of memory as a sequence of bits instead of as a sequence of bytes, starting with the bits of Byte 0 (MSb first), followed by the bits of Byte 1 (MSb first), and so on. Then, if our data type uses N bits, we simply take the first N bits for the first value, the next N bits for the second value, and so on. Then we pad out the end with 0 bits until we have a multiple of 8 bits total for the layer. Figure 3 shows the same example as Figure 2, except it shows memory as an array of bits instead of an array of bytes. This can help with visualizing the data packing, but keep in mind that the atomic element of memory is the byte, not the bit. Thus, Figure 2 is a more accurate representation of actual memory.

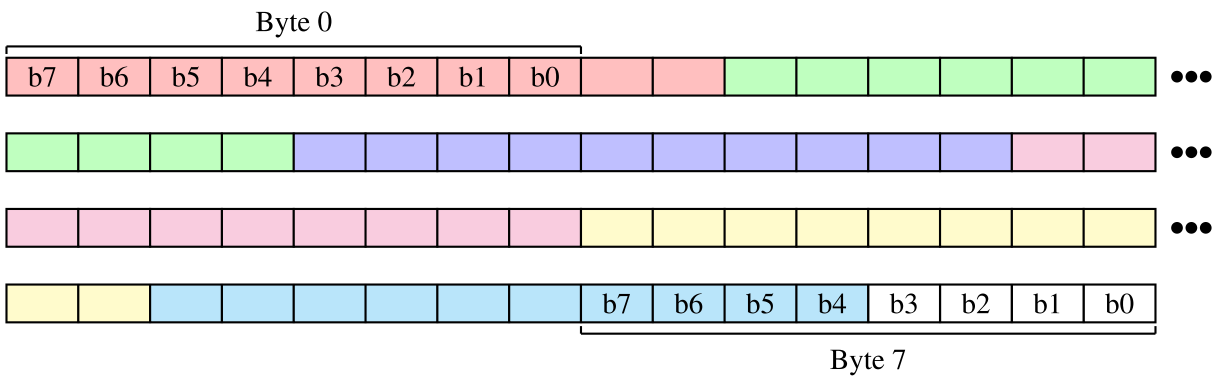

Figure 3: Packed storage of image data. This is the same example as Fig. 2 (6 pixel Uint10 layer). This figure displays memory as an array of bits instead of an array of bytes, with each byte drawn MSb first.

5 The FRF Header

Now, we define the structure of the FRF header. Recall that the header contains a few fixed fields, and then a sequence of information blocks. The fixed portion of the header has the following structure:

Field	Size	Offset	Type	Description
MagicNumber	8	0	Uint64	All FRF files start with the same 8 bytes. This can be used to verify that a file is in fact an FRF file (Value = 3197395143525533696U)
MajorVersion	2	8	Uint16	Major version of the FRF specification that the file complies with. Software for reading FRF files should maintain backwards compatibility. That is, if a reader is designed to support version x, it should also support versions below x.
MinorVersion	2	10	Uint16	Minor version of the FRF specification that the file complies with.
Width	2	12	Uint16	Width of image, in pixels.
Height	2	14	Uint16	Height of image, in pixels.

Table 3: Fixed fields of the FRF file header. All multi-byte fields are stored most-significant byte first (towards the beginning of the file). Fields in grey (in this table only) have their definitions locked for all future versions of the FRF specification to ensure that version checking is always possible. All other fields are subject to change in future versions of the specification.

5.1 Information Blocks

This section defines each of the information blocks that are supported in the FRF header. Each block can appear at most one time in the FRF header. The blocks that are mandatory are marked as such. In general, blocks can appear in any order in the header of an FRF file. The only exception is that the mandatory End-of-Header block must be the last block to appear. Blocks have the following general structure:

Field	Size	Offset	Type	Description
BlockCode	2	0	Uint16	Code indicating the block type
Size	4	2	Uint32	The total size of the block, in bytes (including the BlockCode and Size fields)
Payload		6		Payload for the block. The structure and size of the payload depends on the block type. The payload may be empty (0 bytes).

Table 4: General block structure. All multi-byte fields are stored most-significant byte first (towards the beginning of the file).

Now we define the payload structures for each of the supported block types:

Information Block	Mandatory	Block Code
Layer Manifest Block	Yes	0U
Visualizations Block	Yes	1U
Geo-Tagging Block	No	2U
Geo-Registration Block	No	3U
Camera Information Block	No	4U
Custom Block	No	5U
End-of-Header Block	Yes	6U

Table 5: Supported Header Information Blocks

5.1.1 Layer Manifest Block (Mandatory)

This block is used to define each of the layers that appears in the file. It provides layer names, descriptions, data types, and scale factors for each layer.

Field	Size	Offset	Type	Description
AlphaLayer	2	0	Uint16	Index of alpha layer. Set to $2^{16} - 1$ if there is no alpha layer.
				Start of repeated section
Name			String	Layer name (Max of 48 unicode symbols)
Description			String	Layer description (Max of 1024 unicode symbols)
Units	4		Int32	A code specifying units for values in this layer (see §6.1).
TypeCode	1		Uint8	Code for data type used to encode raw values for layer data
α	8		Float64	Coefficient for mapping raw value to value
β	8		Float64	Coefficient for mapping raw value to value
HasMask	1		Uint8	1: The layer has a validity mask. 0: No validity mask.

Table 6: Layer Manifest Block payload structure.

Notice that the number of bytes used for each layer in the manifest is variable due to the string fields. The number of layers in an FRF file must be between 1 and 2048 ($1 \leq$ value ≤ 2048). A file with no layers or more than 2048 layers is considered invalid. Each layer has an index that is implicitly defined by the ordering of layers in the manifest (the first layer to appear has index 0). Layers appear in the file in the same order as the layer sections appear in the manifest. The number of layers is not saved numerically in the manifest. The number of bytes in the block is known due to the *Size* field for the block. Thus, the size of the payload is known (*Size* − 6 bytes). When done decoding a section of the layer manifest, you check to see if there are any bytes left in the manifest. If there are, then you know there are more layers. If there aren't, then you know that you just decoded the final layer.

Also notice that the number of bytes that the layer data occupies is not saved numerically. This is also implicit and must be computed. The data type is known and the image resolution is known (the image width and height are mandatory fields in the header). The number of bytes is computed from this, of course rounding up to the nearest byte. If the layer does not have a validity mask ($HasMask = 0$), the number of bytes needed to store the layer data is $\lceil Width * Height * \text{BitsPerPixel}/8 \rceil$. If the layer has a validity mask ($HasMask = 1$), the layer data is stored first, padded out to an integer number of bytes. Then the layer validity mask immediately follows it, one bit per pixel, stored row by row (same order as the image data). A value of 1 marks a pixel as valid and a value of 0 marks a pixel as invalid. The validity mask is also padded out to an integer number of bytes. If a layer has a validity mask the total number of bytes used to store the layer data (including the mask) is $\lceil Width * Height * \text{BitsPerPixel}/8 \rceil + \lceil Width * Height/8 \rceil$. Note that validity masks are not allowed for floating point layers since the value NaN is already reserved to mark invalid data.

The first field in the payload for this block is the index of the alpha layer. Setting this field to $2^{16} - 1$ (the largest possible value for a Uint16) indicates that there is no alpha layer. If an alpha layer is used, it is interpreted as global opacity. That is, it applies to any visualization of the image. An alpha layer must still have a normal section in the layer manifest. You can fill in the *Name* and *Description* fields if you please. The *Units* field should be set to 0 (dimensionless quantity). The alpha layer can use any data type and the α and β coefficients will be interpreted and used the same as any other layer for mapping raw values to values. The values for the alpha layer will be interpreted as follows. A value of 0.0 (and any lower value) corresponds to being fully transparent, and a value of 1.0 (and any higher value) corresponds to being fully opaque. Alpha is a distinct concept from validity. Alpha is only used for visualization and compositing different images. The opacity of a pixel has no bearing on whether the pixel is valid on a particular layer.

5.1.2 Visualizations Block (Mandatory)

Visualizations communicate ways of displaying the raster data in an FRF file. A single file may contain several different visualizations. The visualizations block is mandatory and at least one visualization is mandatory. The first visualization found in the visualizations block is called the *default visualization*. This is used whenever a specific visualization has not been selected (for instance, when first opening an FRF file with an image viewer).

All visualizations encode ways of rendering raster data in the FRF file in an RGB color space. A visualization may relate to a single layer in the file, or it may use multiple different layers.

Field	Size	Offset	Type	Description
				Start of repeated section
Name			String	Visualization name (Max of 48 unicode symbols)
Description			String	Visualization description (Max of 1024 unicode symbols)
VisCode	4		Uint32	A code specifying the type of visualization.
VisPayloadSize	4		Uint32	Number of bytes in the *VisPayload* field.
VisPayload				Payload for the visualization. Structure depends on *VisCode*.

Table 7: Visualization Block payload structure.

The following tables lists the supported visualization codes:

VisCode	Name	Description
0	RGB	Map 3 layers in the file to red, green, and blue, respectively.
1	Colormap	Map values in a single layer to different colors in a gradient.

Table 8: Visualization types and associated codes.

Now we define the payload structure for each of the visualization types.

RGB Visualization: This visualization simply grabs 3 layers in the file and uses them as the red, green, and blue channels of the image. A single layer can be used for multiple channels. The channel *Min* and *Max* fields specify the values corresponding to the channel being “off” and the channel being at maximum contribution, respectively.

Field	Size	Offset	Type	Description
RedIndex	2	0	Uint16	Index of layer used for red channel.
RedMin	8	2	Float64	Value corresponding to no red.
RedMax	8	10	Float64	Value corresponding to maximum red.
GreenIndex	2	18	Uint16	Index of layer used for green channel.
GreenMin	8	20	Float64	Value corresponding to no green.
GreenMax	8	28	Float64	Value corresponding to maximum green.
BlueIndex	2	36	Uint16	Index of layer used for blue channel.
BlueMin	8	38	Float64	Value corresponding to no blue.
BlueMax	8	46	Float64	Value corresponding to maximum blue.

Table 9: RGB Visualization payload structure.

Colormap Visualization: This visualization is for rendering a single layer in false color. It consists of a collection of set points, each of which has a value and a corresponding color. When rendering the layer, each value is assigned a color by interpolating between the closest set points on either side of the value at hand. When a value doesn’t have set points on both sides, it saturates to the color of the nearest set point. The set points do not need to appear in any particular order in the visualization payload. The number of set points is known due to the *VisPayloadSize* field (each set point occupies 32 bytes).

Field	Size	Offset	Type	Description
LayerIndex	2		Uint16	Index of layer being visualized.
				Start of repeated section
Value	8		Float64	Value associated with set point.
Red	8		Float64	Red intensity of set point (0.0 to 1.0).
Green	8		Float64	Green intensity of set point (0.0 to 1.0).
Blue	8		Float64	Blue intensity of set point (0.0 to 1.0).

Table 10: Colormap Visualization payload structure.

5.1.3 Geo-Tagging Block

The Geo-Tagging block is used when the raster data corresponds to imagery collected with a real-world camera at a known location and/or orientation. This optional block provides a way of tagging the image with the absolute camera position and/or camera attitude and/or time of acquisition. Position and time are relatively simple to handle. Camera attitude is somewhat more complicated. So far we have only defined a 2D coordinate system for the raster image (see §3); we have not dealt with 3D coordinate systems. In order to define camera attitude unambiguously we must first define a coordinate system for the real-world camera corresponding to an image. The definition must be complete in the following sense. If we are given a camera and an image from that camera, there should be exactly one coordinate system for the camera that is consistent with our definition. If there is any ambiguity in the definition of this frame, it means that attitude data can be interpreted in multiple ways, rendering it effectively useless.

We now define the "implicit camera frame". This is a 3D coordinate system fixed to the real-world camera. The origin of this frame will coincide with the camera center. The directions of the X and Y axes of the camera frame are induced by the axes of our image coordinate system. Imagine drawing a perfectly straight arrow on a hard surface and placing that surface in front of the camera so that the surface is parallel to the focal plane of the camera. Now imagine rotating that surface about the cameras optical axis while monitoring the images produced by the camera. If you rotate the surface so that the image of the vector is parallel to the X axis of the image coordinate system (and pointing in the same direction), then the vector drawn on your hard surface is parallel to the X axis of the implicit camera frame. Now rotate the surface until the image of the vector is parallel to the Y axis of the image coordinate system (and pointing in the same direction). Now the vector drawn on the surface is parallel to the Y axis of the implicit camera frame. The Z axis of the implicit camera frame is defined to be the cross product X $\times$ Y. Figure 1 illustrates this definition.

This definition of the implicit camera frame is deliberately based on the functionality of a camera as opposed to the construction of the camera. We could also have defined the implicit frame in terms of the positions of pixel elements in the focal plane array. The X axis could point from pixel element (0,0) to pixel element (1,0), for instance. Such a definition is quite problematic. A camera with optics that flip the image in all directions, followed by an imaging sensor that flips both directions digitally is indistinguishable from a camera which does neither of these, but they would have different implicit camera frames if we used this

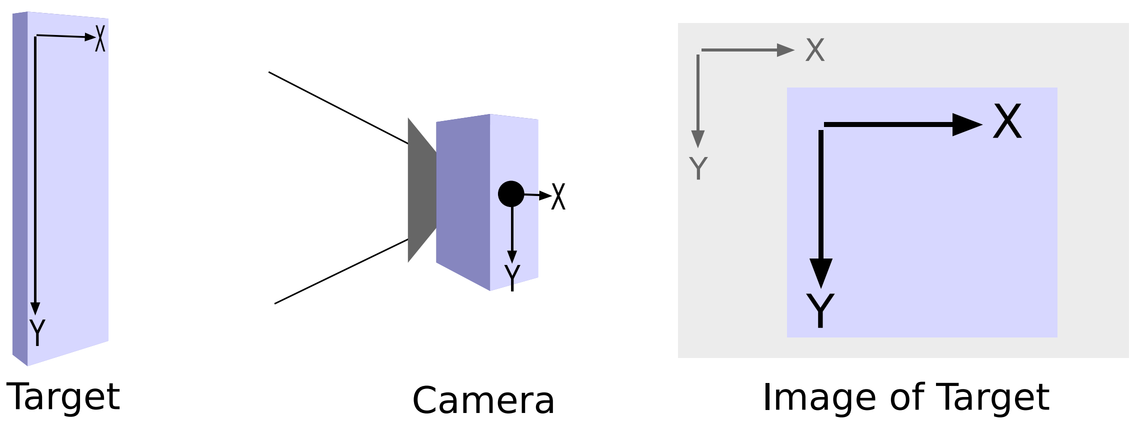

Figure 4: The implicit camera frame. The image of a vector parallel to the X axis of the camera frame should be parallel to the X axis of the image coordinate system. Likewise, the image of a vector parallel to the Y axis of the camera frame should be parallel to the Y axis of the image coordinate system.

candidate definition. Knowing how a camera is constructed should not be necessary in order to define a coordinate system for the camera.

Most cameras are designed to produce "non-inverted" images. If a camera flips its images along the X or Y axes (but not both), then it produces "inverted" images. You can tell whether a camera is producing inverted or non-inverted images using the following test. If you hold up an image produced by a camera in the same location from where the image was taken, a non-inverted image can be rotated until the image appears to match the scene. An inverted image cannot; it will always be backwards in some direction. The implicit camera frame for a non-inverting camera will always have the Z axis in the direction of the optical axis of the camera. The implicit camera frame for an inverting camera will have its Z axis in the direction opposite the cameras optical axis. **The Geo-Tagging Block is only defined for non-inverting cameras**. This ensures that the implicit camera frame always has its Z axis agree with the cameras optical axis. This is not to say that you cannot produce a tagged FRF image using an inverting camera. You can, but you must flip the image about one axis before saving it as an FRF image to simulate a non-inverting camera.

Now that we have a definition of the implicit camera frame, we can encode camera attitude and position by storing the rotation and translation that takes vectors in some fixed, absolute coordinate system to the camera frame. We use the Earth-Centered, Earth-Fixed coordinate system [1] as our absolute frame.

Let $\mathbf{X}$ be an arbitrary position vector. Let $\mathbf{X}_{\text{ECEF}}$ denote the vector $\mathbf{X}$ coordinatized in the ECEF frame (in meters). Let $\mathbf{X}_{\text{Cam}}$ denote the vector $\mathbf{X}$ coordinatized in the implicit camera frame (also in meters). Let $\mathbf{P}_{\text{ECEF}}$ be the position of the camera center in the ECEF frame (in meters). Now we define the orthogonal rotation matrix $\mathbf{C}_{\text{ECEF}}^{\text{Cam}}$ so that for all such vectors $\mathbf{X}$ we have:

$$\mathbf{X}_{\text{Cam}} = \mathbf{C}_{\text{ECEF}}^{\text{Cam}} * (\mathbf{X}_{\text{ECEF}} - \mathbf{P}_{\text{ECEF}}) \tag{1}$$

The vector $\mathbf{P}_{\text{ECEF}}$ holds the position of the camera, and the matrix $\mathbf{C}_{\text{ECEF}}^{\text{Cam}}$ holds the orientation of the camera, both with respect to a known coordinate system fixed to the earth. These are the two quantities that we save in the Geo-Tagging block for storing camera pose. Additionally, we allow for storing the time of acquisition for the image with a GPS Time field. In general, one may know one or more of these quantities without knowing others (position information is much more common that attitude information, for instance). We do not require all fields to be known in order to have a valid Geo-Tagging block, although at least one field should be filled in if this block is to be used.

We are ready to define the payload structure for this block:

Field	Size	Offset	Type	Description
$\mathbf{P}_{\text{ECEF}}$	24	0	Vec3	Position of the camera center in ECEF coordinates (in meters). In practice this will often be approximated by the position of the phase center of a GPS antenna that is positioned close to the camera. Set all coordinates to NaN if position is unknown.
$\mathbf{C}_{\text{ECEF}}^{\text{Cam}}$	72	24	Mat3	Rotation between ECEF and Implicit Camera Frame. See precise definition above. Set all elements to NaN if orientation is unknown.
Time	12	96	GPST	The time instant corresponding to the beginning of the image acquisition interval. Set Week subfield to 0U and TOW subfield to NaN if time is unknown.

Table 11: Geo-Tagging Block payload structure.

5.1.4 Geo-Registration Block

The Geo-Registration block is used when each pixel in an image corresponds to a specific, known point on the Earth (we will call such a point a geo-location). This is different from Geo-tagging, where we store information about camera position and orientation.

We start with some requirements for an encoding of geo-registration data:

- The registration should be unambiguous. For a given pixel in a given image we must have exactly one geo-location corresponding to it. We must be able to compute this geo-location with very high accuracy. We must fully define the mapping that takes pixel coordinates to geo-locations (we can't leave things like interpolation schemes up to implementation).
- We must be able to invert the mapping from pixel coordinates to geo-locations. That is, for a given geo-location we need to be able to find the pixel or pixels in the image that correspond to it, if any exist. In the event that the mapping is non-injective, we should at least be able to find one of the pixels corresponding with a given geo-location, should it contain a non-empty pre-image.
- It must be possible to efficiently evaluate the forward and inverse mappings between pixel coordinates and geo-locations.

We must make a decision about whether to use 2D or 3D representations of geo-locations. There is some appeal to using a 3D representation, such as a 3-element vector holding position in the ECEF coordinate system. This provides some interesting functionality, like making it possible for us to represent slices of arbitrary volumes above or even inside the earth. For instance, if one wanted an image to represent a cross-sectional slice of the planet, this type of representation would be a good choice. It does, however, complicate the format and implementation. In particular, since FRF is a 2D raster format each image will correspond to a 2D surface in world-space. Consequently, using 3D representations of geo-locations makes it much more difficult to invert the mapping from points on Earth to pixel coordinates since the image takes up no volume in 3D world coordinates. That is, virtually all geo-locations we might be given will not correspond to locations in the image. We would need to handle tolerances on how close to the image surface a point needs to be before we consider it to have a valid inverse and we need to handle projections onto arbitrary 2D manifolds to make accurate inversion possible. From this standpoint, 3D representations of geo-locations would be more appropriate for a 3D raster format, which is beyond the scope of FRF.

We also note that the vast majority of use cases for geo-registration do not require the level of flexibility offered by using 3D representations of geo-locations. Typically, imagery is meant to be associated with the surface of the earth, the surface of the reference ellipsoid, or a plane at a fixed altitude. Given these considerations we elect to use 2D representations of geo-locations. This is not without problems. Any parameterization of the surface of the earth will involve distortions of distances, discontinuities (e.g. the line of 0 longitude), and/or over-parametrized points (e.g. the point at $90°$ latitude). These non-idealities must be handled as gracefully as possible.

We elect to use latitude/longitude pairs (in radians) to represent geo-locations. These are understood to represent locations with respect to the WGS84 reference ellipsoid. We will use a separate tag to record whether the image corresponds to the surface of the earth, a plane at fixed altitude, etc.

We use bi-linear interpolation on a regular grid to define the function that maps pixel coordinates to geo-locations. In the simplest (and probably most common) case, a file will contain geo-locations for the 4 corners of the image. If the image spans a very large area, or is close to a pole where distortion may be problematic, or if one just desires extremely high registration accuracy, the grid can be shrunk down from the full image size by any integer divisor in each dimension. As is typical for bi-linear interpolation, a pixel location is mapped to a geo-location using interpolation on the corners of the cell in which the point lies. Arbitrarily high registration accuracy can be achieved by shrinking the grid appropriately and forward mapping is always extremely fast. The challenge with this is that inverting grid-based bi-linear interpolation is non-trivial and not nearly as fast as evaluations in the other direction. This can be overcome, however, by a sufficiently clever implementation. The interpolation scheme should map the latitude and longitude of each of the 4 corners of a given cell to a common principal range that avoids any longitude discontinuity before doing the interpolation.

Observe that nothing about the registration scheme enforces injectiveness of the registration function. That is, we can identify multiple pixel locations with the same geo-location. In this case, the registration function does not have an inverse, strictly speaking. However, it should still be possible to retrieve a point in the pre-image of any geo-location in the range of the registration function. That is, if one tries to lookup the pixel coordinates corresponding to a given geo-location, and if there are multiple pixel coordinates that map to the given location, an implementation is required to retrieve exactly one of those pixel locations. The implementation is allowed to select which one to return. This makes it possible to make a single registered image with full global coverage. We would simply overlap the image slightly on the seam, repeating values as necessary. We are guaranteed that when we lookup a pixel corresponding to a geo-location along the seam, we will get one of the correct pixel locations associated with that point, and if we created the image carefully it won't matter which one we use to sample the image. An implementation may attempt to warn the caller about the existence of multiple pixel locations associated with a given geo-location, but it should still return one of the solutions. This should not be considered an error.

A known limitation of the registration block we are defining is that it will not handle images that cover either the North or South pole well.

We lead the registration block with a field with a hard-coded value. This represents the version or type of registration block. The sole purpose of this is to make it simple and clean in future versions of the specification to offer alternate types of registration data. The main rationale for this would be if people are running up against the limitations of the block defined here, such as wanting to register imagery along planetary cross sections, or against other planets. These use cases are not accommodated by the scheme defined here, but we leave the door open to add them in the future.

Field	Size	Offset	Type	Description
RegistrationType	2	0	Uint16	Value = 0U.
Altitude	8	2	Float64	WGS84 altitude (meters) associated with the imagery. A value of 0.0 corresponds to the surface of the WGS84 reference elipsoid. The special value NaN is reserved to indicate that the imagery is associated with the surface of the earth, even though the altitude of the surface may be different at each point. Note that the surface altitude does not need to be known to register imagery against the surface.
GridWidthDivisor	2	10	Uint16	The number of columns the image is broken into for registration bi-linear interpolation. This field cannot be 0U.
GridHeightDivisor	2	12	Uint16	The number of rows the image is broken into for registration bi-linear interpolation. This field cannot be 0U.
Start of repeated section				
$P_{i,j}$-*Lat*	8		Float64	Latitude (radians) of grid point (i, j)
$P_{i,j}$-*Lon*	8		Float64	Longitude (radians) of grid point (i, j)

Table 12: Geo-Registration Block payload structure.

The repeated section gives the geo-location corresponding to a single grid point. The total number of grid points is $(\textit{GridWidthDivisor} + 1) * (\textit{GridHeightDivisor} + 1)$. The grid points appear in the Geo-registration block one row at a time, starting with the upper-left corner of the image. That is, grid point $P_{0,0}$ is first, followed by point $P_{1,0}$, and then $P_{2,0}$, and so on through point $P_{\textit{GridWidthDivisor},0}$. Then we start with the next row: $P_{0,1}$, $P_{1,1}$, and so on. The image coordinates corresponding to grid point (i, j) are:

$$\begin{pmatrix} (\mathit{Width} - 1) * (i/\mathit{GridWidthDivisor}) \\ (\mathit{Height} - 1) * (j/\mathit{GridHeightDivisor}) \end{pmatrix}$$

5.1.5 Camera Information Block

If an FRF file is being used to store imagery collected with a real-world camera, it can be useful to embed information about the camera in the image file. This optional block is for storing this type of metadata. Due to the large variety of information that may be known about a camera, little in this block is static. Instead, it defines a collection of tags that may be specified.

Field	Size	Offset	Type	Description
Start of repeated section				
TagCode	2		Uint16	A code identifying the tag.
TagPayloadSize	2		Uint16	Number of bytes in the tag payload.
TagPayload				Structure depends on *TagCode*.

Table 13: Camera Information Block payload structure.

In Table 14 we list the supported Information Tags. Notice that there are no tags for specifying conventional radiometric calibration quantities like dark current or sensitivity. This is deliberate. The “proper” way to store radiometrically calibrated imagery to an FRF file is to save each band to a separate layer where the value corresponds to scene irradiance (or scene spectral irradiance). The units field for each layer should be set accordingly. In the event that the camera does not do any sophisticated image processing at the time of collection, one can simply save the raw Digital Numbers (DNs) as the raw values for each layer and then set the coefficients α and β as needed to map these DNs to irradiance. Setting these coefficients is equivalent to specifying the dark current and sensitivity for a band, but it is saved in a more accessible form for one reading the imagery. If the mapping from DN to irradiance is non-linear (perhaps due to some image processing steps during acquisition), one will need to apply the necessary transfer function and save the resulting irradiances. Notice that if we just saved raw DNs along with dark current and sensitivity, it would be impossible to correct for a non-linear imager transfer function, making the approach used by FRF more general.

It is also possible to save “raw” imagery to an FRF file by saving all raw DNs to a single image layer. The RawInfo tag makes it possible to interperet such data without additional knowledge about the camera or color filter array (CFA) design. It is even possible to save radiometrically calibrated imagery in raw form through the use of this tag, although this approach is somewhat less flexible that saving each band to a different layer (you cannot correct for arbitrary non-linear image transfer functions).

TagCode	Name	Type	Size	Description
0	Make	String		Camera manufacturer.
1	Model	String		Camera model string.
2	BodySerial	String		Serial number of camera. If camera has removable optics, this should be the serial number of the camera body.
3	ImagerMake	String		Manufacturer of imaging sensor.
4	ImagerModel	String		Part number of imaging sensor.
5	LensMake	String		Lens manufacturer.
6	LensModel	String		Lens model number.
7	LensSerial	String		Lens serial number (if camera has removable optics).
8	FNumber	Float64	8	Lens F number.
9	FocalLength	Float64	8	True focal length of lens (m).
10	XPitch	Float64	8	Horizontal pixel size (m).
11	YPitch	Float64	8	Vertical pixel size (m).
12	ExposureTime	Float64	8	Image exposure time (s).
13	CamMatrix	Mat3	72	Camera calibration matrix - See below.
14	RadialDistortion		48	Radial distortion coefficients - See below.
15	SpectralProfile			Sensitivity curves for each band - See below.
16	RawInfo			Info needed to interpret raw image data - See

Table 14: Camera Information Block - Supported Tags.

Camera Calibration Matrix: This matrix characterizes how the real world projects onto the image plane. This matrix depends on the camera optics and on the geometry of the image sensor. In the absence of radial distortion, this matrix is all that is needed to compute how a world point in the implicit camera frame (defined in §5.1.3) projects onto the image sensor and to compute the resulting pixel coordinates.

Let $\mathbf{X}_{\mathrm{Cam}}$ be the position of a point in the implicit camera frame, in meters and expressed in homogeneous coordinates (so it is a 4-element vector). Let $\mathbf{x}$ be the projection of $\mathbf{X}_{\mathrm{Cam}}$, expressed in homogeneous image coordinates (so $\mathbf{x}$ is a 3-element vector). Let $\mathbf{K}$ be the CamMatrix in Table 14. Then, the following holds:

$$\mathbf{x} = \mathbf{K}\,[\mathbf{I} \mid \mathbf{0}]\,\mathbf{X}_{\mathrm{Cam}} \tag{2}$$

This tag is the preferred way of saving optical calibration information. When this tag co-exists with other tags related to optical calibration (such as FocalLength), CamMatrix should be used whenever possible over other quantities.

Radial Distortion Coefficients: This tag lets you encode coefficients for correcting radial distortion in the imagery. When this tag and a CamMatrix tag are both present, it is assumed that the CamMatrix is valid *after* radial distortion has been corrected. This tag has the following payload structure:

Field	Size	Offset	Type	Description
C_x	8	0	Float64	Distortion center, X-coordinate.
C_y	8	8	Float64	Distortion center, Y-coordinate.
a_1	8	16	Float64	Distortion coefficient.
a_2	8	24	Float64	Distortion coefficient.
a_3	8	32	Float64	Distortion coefficient.
a_4	8	40	Float64	Distortion coefficient.

Table 15: Radial Distortion Tag Payload Structure

Let I represent the uncorrected image, as a function of image coordinates. Let $\tilde{I}$ represent the corrected image, also as a function of image coordinates. Then the following equation expresses how we correct for radial distortion:

$$\tilde{I}\begin{pmatrix} x \\ y \end{pmatrix} = I\left(\begin{pmatrix} x \\ y \end{pmatrix} + L(r)\left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} C_x \\ C_y \end{pmatrix}\right]\right) \tag{3}$$

Where:

$$\begin{aligned} L(r) = &\quad 1 + a_1 r + a_2 r^2 + a_3 r^3 + a_4 r^4 \\ r = &\quad \sqrt{(x - C_x)^2 + (y - C_y)^2} \end{aligned}$$

Spectral Profile (TagCode = 15): This tag is used to communicate the spectral profiles of different measurement bands for a camera. This tag is not implemented in the draft specification, but we list requirements for future implementation. We need to be able to specify a normalized sensitivity function for one or more "bands". These bands should either map to layers in the image or, in the case of raw imagery, to filter indices in the CFA (these indices should correspond to the indices used in the RawInfo tag, if present).

Raw Info (TagCode = 16): This tag is used to save information about the design of the color filter array (CFA) if the FRF file contains raw imagery. This tag is not implemented in the draft specification, but we list requirements for future implementation. This tag will pertain to raw imagery saved in a single layer of an FRF image. This layer may or may not be the only layer in the file, but only one layer can be described by this tag. We need to specify the dimensions of the repeating CFA block pattern (i.e. 2x2 Bayer-style, 3x3, 4x4, etc.). We need to provide the pattern using filter indices. For each filter index referenced in the pattern, we need to provide coefficients α and β to map the de-mosaiced values to irradiance or spectral irradiance, or something else (whatever is indicated in the units field for the raw layer). These override the α and β set in the Layer Manifest for the raw layer, and allow us to apply different mappings for each band.

This can be used in conjunction with a SpectralProfile tag to save the specific sensitivity profiles for each filter in the CFA.

5.1.6 Custom Block

Custom blocks are provided so that additional information can be embedded in an FRF file, beyond what is explicitly supported by the standard. Software is not required to be able to parse and understand the payloads of custom blocks in order to comply with the FRF standard. Thus, custom blocks should be used only as a last resort. If you define a custom block and use it when saving files, you may be the only one capable of reading and understanding that data.

The only rule for custom blocks is that the payload must begin with an 8-byte field, to be interpreted as a 64-bit unsigned integer (stored most significant byte first). This field is referred to as the "Custom Block Code" and is used to identify your custom block and distinguish it from other custom blocks. Custom block codes are not regulated, although a list will be maintained of known custom block codes. It is highly recommended to check with this list when selecting a code for a custom block to avoid collisions with other custom blocks.

As a rule of thumb, custom blocks should make use of standard primitive and compound data types already provided by the FRF spec whenever possible, and they should respect the Endianness of those types. This allows you to use already-provided serialization and deserialization methods instead of needing to create your own from scratch.

5.1.7 End-of-Header Block (Mandatory)

This block marks the end of the FRF file header. The payload for this block is empty (0 bytes). The first byte of image data for the first layer begins immediately following this block.

6 Appendix

6.1 Layer Units

The layer manifest has a 32-bit "units" field for each layer. The field is mandatory, although there is a code reserved for unspecified units. This field can be used when data has a specific meaning with units. This specification does not attempt to provide all possible units for all possible types of measurements. Instead, it aims to provide one standard, SI-derived set of units for any quantity, and then provide a single unambiguous representation of these units.

The bit pattern for the unit code is broken into eight smaller fields. Seven of these fields are used to store exponents (positive or negative) for each of the seven SI base units. The remaining 1-bit field is called the "special" flag. When this flag is set, the other 31 bits are used to represent special units that are not representable using the normal SI base units. The unit code is decomposed as follows:

b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16	b15
spc	cd-exp				mol-exp				K-exp				A-exp			

b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
s-exp					kg-exp					m-exp				

Figure 5: Decomposition of the unit code. When the special flag (denoted "spc" above) is cleared, the other fields are all interpreted as signed, 2's-compliment integers. When spc is set, the remaining 31 bits are used to represent special units.

When the special flag is cleared, the remaining seven fields are each interpreted as a 4-bit or 5-bit 2's-compliment integer, and represents an exponent for one of the base SI units. The units for a given set of exponents are defined to be:

$$\mathbf{cd}^{\text{cd-exp}}\mathbf{mol}^{\text{mol-exp}}\mathbf{K}^{\text{K-exp}}\mathbf{A}^{\text{A-exp}}\mathbf{s}^{\text{s-exp}}\mathbf{kg}^{\text{kg-exp}}\mathbf{m}^{\text{m-exp}}$$

The above expression uses these standard abbreviations for the SI base units:

Abbreviation	Unit	Measure For
cd	candela	luminous intensity
mol	mole	Amount of matter
K	kelvin	thermodynamic temperature
A	ampere	electric current
s	second	time
kg	kilogram	mass
m	metre	length

Since we choose bit b31 for the special flag bit, it is convenient to interpret the unit code as a signed integer. When we do this, non-negative values correspond to normal SI-derived units, and negative values correspond to special units.

Each exponent has upper and lower limits, depending on the size of the field. This imposes some limitations on the units that can be represented in this way, but in practice it is quite unusual to need larger powers of a base unit in either the numerator or denominator than is expressable with these exponents.

Some may find it a bad choice to offer only SI-derived units. However, the α and β scaling coefficients attached to each layer makes offering multiple unit options unnecessary. A unit conversion can be accomplished by simply adjusting these coefficients. The raw values do not need to be changed and no resolution is lost. For instance, if we have a layer representing distances in inches, we can effectively convert to m by replacing α with 0.0254α. Since α and β are both 64-bit floating point values, they can absorb enormous scale factors without losing any precision (up to $10^{\pm 308}$). Thus, if our preferred units are, for example, cubic km (km^3), we have to re-scale the layer to use cubic meters instead (m^3). This requires multiplying each value in the layer by 10^9. Rather than changing the raw value for each pixel, this is easily accomplished by replacing α for this layer with $10^9\alpha$. Since α is double precision, this will generally result in no loss of precision.

We demonstrate how to compute units with an example. A common set of units for irradiance are $\mu\mathrm{W}/\mathrm{m}^2/\mathrm{nm}$. Assume we have a floating point image representing irradiance with these units and we want to save this image as an FRF layer with the units field set correctly. We must compute the decomposition of these units into SI base units:

$$\frac{\mu\mathrm{W}}{\mathrm{m}^2\mathrm{nm}} = \frac{\mu\mathrm{W}}{\mathrm{m}^2\cancel{\mathrm{nm}}} * \frac{10^9\cancel{\mathrm{nm}}}{\mathrm{m}} = 10^9 * \frac{\mu\mathrm{W}}{\mathrm{m}^3} = 10^9 * \frac{\cancel{\mu\mathrm{W}}}{\mathrm{m}^3} * \frac{\mathrm{W}}{10^6\cancel{\mu\mathrm{W}}} = 10^3 * \frac{\mathrm{W}}{\mathrm{m}^3} = 10^3 * \frac{\mathrm{kg\ m}^2}{\mathrm{m}^3\mathrm{s}^3} = 10^3\mathrm{s}^{-3}\mathrm{kg}^1\mathrm{m}^{-1}$$

This shows us that the SI-derived units we need to use are $\mathrm{s}^{-3}\mathrm{kg}^1\mathrm{m}^{-1}$. These are not exactly the same as the units our image is currently stored in, but they only differ by scale factor. Thus, when we save the layer, we will need to multiply every value by 10^3 (or set $\alpha = 10^3$) and then use the following exponents to compute the units field:

Exponent	cd-exp	mol-exp	K-exp	A-exp	s-exp	kg-exp	m-exp
Value	0	0	0	0	-3	1	-1
Bit Pattern	0000	0000	0000	0000	11101	00001	11111

This set of exponents results in the following hexadecimal bit pattern (we need to make sure that the special flag is cleared): 0x0000743F. To get the unit code, we interpret this as a signed integer: 29759.

As mentioned earlier, we reserve values with bit b31 set for special units. Since we interpret this code as a signed integer, this is the same as saying that negative values are reserved for special units. In the initial version of the specification, there is only one special value. A code of -1 (in hex: 0xFFFFFFFF) is used to indicate unknown or unspecified units. This is not the same as a dimensionless quantity. A dimensionless quantity corresponds to a code of 0 (in hex 0x00000000). That is, the special flag is cleared and all exponents are set to 0.

We end with a table of common units and their corresponding codes.

Measure For	Units	SI Decomposition	Unit Code (Hex)	Unit Code (Int)
Unspecified Units			0xFFFFFFFF	-1
Dimensionless Quantity			0x00000000	0
Distance	m	m^1	0x00000001	1
Area	m^2	m^2	0x00000002	2
Volume	m^3	m^3	0x00000003	3
Time	s	s^1	0x00000400	1024
Speed	m/s	$\mathrm{m}^1\mathrm{s}^{-1}$	0x00007C01	31745
Acceleration	m/s^2	$\mathrm{m}^1\mathrm{s}^{-2}$	0x00007801	30721
Mass	kg	kg^1	0x00000020	32
Surface Density	kg/m^2	$\mathrm{kg}^1\mathrm{m}^{-2}$	0x0000003E	62
Density	kg/m^3	$\mathrm{kg}^1\mathrm{m}^{-3}$	0x0000003D	61
Power	W	$\mathrm{kg}^1\mathrm{m}^2\mathrm{s}^{-3}$	0x00007422	29730
Force	N	$\mathrm{kg}^1\mathrm{m}^1\mathrm{s}^{-2}$	0x00007821	30753
Irradiance	$\mathrm{nW}/\mathrm{m}^2/\mathrm{nm}$	$\mathrm{kg}^1\mathrm{m}^{-1}\mathrm{s}^{-3}$	0x0000743F	29759

Table 16: Common units, their decompositions into SI base units, and corresponding unit codes. This is by no means an exhaustive list. If none of these units are appropriate for a given situation, you will need to compute the unit code yourself (see the example in this section).

It might seem that the unit code is overly restrictive and cannot represent some important units of measurement. For instance, what if we want to have a layer that represents population density in persons per unit area? A person is not an SI dimension. Thus, in such a case, the units field would be set to represent inverse area (m^{-2}). The only reason this is unsettling is that in this case the units alone are not sufficient to communicate the full meaning of the layer. While one can often guess the meaning of a layer just by looking at the units, this is not the intent of the field. The layer manifest also encodes a name and description for the layer that can be used to embed the full meaning of the layer. Another example might be a layer representing solid area measurements in steradians. Technically, the steradian is a dimensionless quantity (it is a ratio of areas, much like the radian is a ratio of distances). In this case we would set the unit code to 0 to indicate a dimensionless quantity. We might chose a layer name that helps clarify the meaning of the data, like “Solid Angle (steradians)”.

We end this section with a short philosophical defense of the treatment of units in FRF. The purpose of a standard is to enable information exchange. People must be able to put data into a defined format and hand it to others. Without explicit help from the creator, the receiving party should be able to read and correctly interpret the data. This is the minimum requirement for a standard and it is worth noting that many formats that people use today to exchange data fail to meet even this minimalist definition. For instance, if you put meaningful data into a JPEG image (like an elevation map), the receiver will not be able to make sense of the data without express help from the one who created the file.

In addition to meeting this definition, a good standard will not overburden those creating data or those interpreting data and it will attempt to balance what burden there is between these parties. A legitimate complaint one might level against the FRF units field is that it requires much of one trying to create an FRF file because they must manipulate their data to conform to the one set of appropriate, available, SI-derived units for their data. We argue that this burden is more than offset by the fact that it is comparatively effortless to correctly interpret such data once stored correctly. It can even be done in a completely automatic system and if one wants to convert to their preferred non-SI-derived units, they can easily interpret the data, read the units field, and apply the necessary conversion factor. Compare this to the simplest, least-structured, alternative approach to the units field: a human-readable units string. While it alleviates the burden on those creating data (they can just keep whatever units their data is naturally in and come up with an appropriate units string), it makes autonomous interpretation of data much harder. Different people may use different preferred units for the same quantity. As a result, one reading this data must support all of the possible units for each possible quantity and store conversion factors for all of them. Even then, there may be units unknown to those reading the data, making correct interpretation impossible. Even worse from an implementation perspective is the fact that different people may use the same units but come up with different ways of expressing those units in string form. Thus, not only does an implementer need to have a database of all known units for a given measurement type, with conversion factors, but they need to have a database of all reasonable representations of those units in string form. Even then, there are bound to be some standard-compliant files that cannot be read and correctly interpreted. It seems self-evident that the small burden that FRF places on those creating data is justified in light of the dramatically reduced burden on those reading and interpreting data.

For an even more extreme example, consider how GeoTiff embeds registration data. One can use any projected or non-projected coordinate system that has ever been defined and any units. As a result, it is trivial to create a standard-compliant GeoTiff, but there has likely never been a complete implementation of the GeoTiff standard that can read and correctly interpret all valid GeoTiff images. We do not want to emulate this failing.

References

[1] Wikipedia. Ecef — wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=ECEF&oldid=673101265, 2015. [Online; accessed 28-December-2015]. 18