This document is not finalised. Do not use its contents as a reference as they may change in the future as the specification process advances.

Simple File Format Family - SF3

v0.9

Maintainer:	Yukari Hafner ¡shinmera@tymoon.eu¿
Project URL:	https://shirakumo.org/docs/sf3
Specification Source:	https://shirakumo.org/projects/sf3
Discussion Channel:	irc://irc.libera.chat/#shirakumo

Contents

1 Introduction

SF3 (Simple File Format Family) is a family of file format specifications. These file formats all follow a similar scheme and the same principles. They are intended to be easy to read and write, and cover base use-cases of various binary formats.

1.1 Principles

SF3 formats follow these principles:

- **No versioning**
  These formats explicitly do not include any versioning at all. The way they are described in this document is final and will not change. This means the formats are eternally forwards and backwards compatible.
- **No extensibility**
  There are no vendor extensibility blocks or other parts that could be added by third parties. This ensures that a consumer of these formats will always be able to read the full file and know what every single bit in it means.
- **No optional blocks**
  There are no optional blocks or parts in the formats that could be omitted. This means there is no conditional parsing needed and the structure of the files is always clear.
- **Only raw data**
  The data is not compressed, encrypted, or otherwise transformed. Data is always raw. If encryption or compression is desired, the entire file can instead be wrapped in a compression or encryption stream (gzip, lzma, etc).
- **Always little-endian**
  The formats are always little-endian wherever byte order matters. This is compatible with the vast majority of processors and software today and means no byte rearrangement is necessary when loading to memory.
- **Similar layout**
  Each format in the family follows a very similar format of identifier, header, and payload. This ensures that the files remain easy to parse, understand, and debug.

1.2 Nomenclature

The following clarifies how to interpret certain words in regards to this standard:

- **file** – A bounded sequence of octets that should be interpreted as having a structure of one of the formats outlined in this standard.
- **implementation** – A program that is capable of interpreting files in accordance with this specification.
- **must** – If a file should violate this requirement, it is invalid and must be rejected by the implementation.
- **must not** – If a file should meet this requirement, it is invalid and must be rejected by the implementation.

- **should** – It is heavily recommended to follow this requirement, however implementations must be able to handle the case where this requirement is not met.
- **should not** – It is heavily recommended not to follow this requirement, however implementations must be able to handle cases where it is met.
- **may** – The behaviour is optional, however implementations must be able to handle it.

2 Specification Description

The format specifications in this document use a Backus-Naur-Form-style abstract syntax language. The language is defined here:

```
Format       ::= Definition+
Definition   ::= Rule Identifier? Description? '\n'
Identifier   ::= '::=' Sequence
Description  ::= '---' text
Sequence     ::= Composition (' ' Composition)*
Composition  ::= Composable Counter?
Composable   ::= Rule
               | OctetArray
               | Octet
               | Type
               | Switch
               | BitRange
               | BitCount
               | '(' Sequence ')'
Counter      ::= OneOrMore
               | AnyNumber
               | ExactNumber
OctetArray   ::= '[' octet (' ' octet)* ']'
Type         ::= ('int' | 'uint') IntBittage
               | 'float' FloatBittage
               | 'string' ExactNumber
IntBittage   ::= '8' | '16' | '24' | '32' | '64'
FloatBittage ::= '16' | '32' | '64'
Rule         ::= name
OneOrMore    ::= '+'
AnyNumber    ::= '*'
ExactNumber  ::= '{' number '}'
Switch       ::= '<' Rule SwitchCase ('|' SwitchCase)* '>'
SwitchCase   ::= octet+ ':' Sequence
BitRange     ::= Rule ':' (mask ',')? mask
BitCount     ::= Rule '#'

name     --- The name of a rule as a sequence of non-numeric ASCII
             characters.
octet    --- Eight bits expressed as two hexadecimal digits.
text     --- Human-readable textual description of the contents.
number   --- A textual description of the number of occurrences. Can
             make a reference to other rules, in which case the rule's
             content designates a runtime number.
mask     --- A textual description of the number of bits to use.
```

White space unless otherwise mandated may be inserted liberally to aid readability. Each rule ultimately defines a sequence of octets that should be parsed. The `Types` mentioned translate to signed integers, unsigned integers, and IEEE floating point numbers of the given number of bits. `string` designates a null-terminated UTF-8 encoded character sequence with an octet length (including null-terminator) as indicated by the required following `ExactNumber` rule.

A `Switch` acts as a runtime switching based on the value of the referenced `Rule`. This means that when evaluated, the `Switch` should act as if it were the `Sequence` of the `SwitchCase` whose `octet` sequence matches the value of the referenced `Rule`. If the `Rule` evaluates to a value that does not match any of the `SwitchCase`s, the file is invalid. For example, `<x 00: int8 | 01: int16>` with `x` evaluating to `00` would match an `int8`, and when evaluating to `01` would match an `int16`.

A `BitRange` extracts the bits of the specified `Rule` by shifting the integer to the right by the first `mask` number of bits, if given. It then masks the remaining integer such that only the number of bits specified in the second `mask` remain. For example, `x:2,4` with `x` being the binary sequence `10101101` would leave the bits `1011`.

A `BitCount` is the number of set bits of the specified `Rule`.

3 Formats

Each format is made up of the following structure, where a valid file must begin with the `File` rule. The prefix to the `Header` is structured to always take up 16 octets, and each file can only contain a single instance of a format and its payload.

```
File       ::= Identifier Header Payload
Identifier ::= [ 81 53 46 33 00 E0 D0 0D 0A 0A ] format-id checksum [ 00 ]
checksum   ::= uint32 --- A CRC32 checksum of the Payload.
format-id  ::= uint8 --- A single octet identifying the format.
```

The rationale for the ten octets in the identifier is as follows:

- `81` An octet to stop byte-peekers from determining text. The octet lies in the undefined ranges of ASCII, ISO-8859-1, Windows-1252, and SJIS.
- `53 46 33` ASCII sequence spelling `SF3` for human-readability.
- `00` A null octet to stop C-string utilities from trying to munch the rest of the file.
- `E0D0` An invalid UTF-8 octet sequence.
- `0D0A0A` A CRLFLF sequence to catch bad line conversion utilities.

The `Header` and `Payload` will be described by the individual formats.
The values for the `format-id` are interpreted as follows:

- `01` — Archive
- `02` — Audio
- `03` — Image
- `04` — Log
- `05` — Model
- `06` — Physics-Model
- `07` — Table
- `08` — Text
- `09` — Vector Graphic

Any other value for the `format-id` is reserved for future formats in this spec. If an implementation only supports a subset of these formats, it must generate an error when it encounters a `format-id` that it does not support.

3.1 Archive

The Archive format allows storing multiple files in one binary package. The file also includes some metadata so that the files can be stored with a relative path and mime-type, allowing both file-system extraction, and content inspection without explicit extraction.

```
Header       ::= Count MetadataSize
Payload      ::= Metadata Files
Count        ::= uint64
             --- The number of entries.
MetadataSize ::= uint64
             --- The octet size of the Metadata payload.
Metadata     ::= EntryOffset{Count} MetaEntry{Count}
EntryOffset  ::= uint64
             --- The octet offset of the corresponding MetaEntry from
                 the beginning of Metadata.
MetaEntry    ::= ModTime Checksum Mime Path
             --- A descriptor of modification time, CRC checksum, mime
                 type, and path.
ModTime      ::= int64
             --- A unix-time timestamp denoting when the file was last
                 modified. Note the time is signed. Negative numbers
                 designating times before 0:0:0 1.1.1970.
Mime         ::= mime-length string{mime-length}
             --- The mime-type of the corresponding file.
mime-length  ::= uint8
Checksum     ::= uint32
             --- A CRC32 checksum.
Path         ::= path-length string{path-length}
             --- The relative path of the corresponding file.
path-length  ::= uint16
Files        ::= FileOffset{Count} FilePayload{Count}
FileOffset   ::= uint64
             --- The octet offset of the corresponding File from the
                 beginning of Files.
FilePayload  ::= file-length uint8{file-length}
             --- A binary file payload.
file-length  ::= uint64
```

The included `MetadataSize`, `EntryOffset`, and `FileOffset` fields should allow constant-time access to any content within the archive.
If no mime-type is known for a file that should be stored, the corresponding `Mime` should be set to `application/octet-stream`.

Each `EntryOffset` in `Metadata`, and each `FileOffset` in `Files` must be larger than the preceding entry.

3.1.1 Use-Case

This format is useful if you need a way to bundle files together into a single payload, and require constant-time, typed access to individual files without having to extract, decompress, or decrypt.

3.2 Audio

This format is for storing plain audio sample data. It includes support for most types of sample formats and channel numbers out there. The sample channels are interleaved, allowing the file to be written on the fly.

```
Header      ::= samplerate channels format frame-count
Payload     ::= (Sample{channels}){frame-count}
samplerate  ::= uint32
            --- The samplerate in Hz.
channels    ::= uint8
            --- The number of audio channels.
format      ::= uint8
            --- A single octet identifying the per-sample data type.
frame-count ::= uint64
            --- The number of audio frames in the file.
sample-size ::= Format:4
            --- The number of octets per sample.
Sample      ::= <format 01: uint8
                      | 02: int16
                      | 04: int32
                      | 08: int64
                      | 11: uint8
                      | 12: uint16
                      | 14: uint32
                      | 18: uint64
                      | 22: float16
                      | 24: float32
                      | 28: float64>
            ---  A single-channel value in the format indicated
                 by format.
```

The values for `format` are interpreted as follows, declaring the encoding of a single sample:

- `01` unsigned 8-bit integer in the non-linear "A-law" scheme.
- `02` signed 16-bit integer linear PCM.
- `04` signed 32-bit integer linear PCM.
- `08` signed 64-bit integer linear PCM.
- `11` unsigned 8-bit integer in the non-linear "u-law" scheme.
- `12` unsigned 16-bit integer linear PCM.
- `14` unsigned 32-bit integer linear PCM.
- `18` unsigned 64-bit integer linear PCM.
- `22` 16-bit short-float linear PCM.
- `24` 32-bit single-float linear PCM.
- `28` 64-bit double-float linear PCM.

Any other value for `format` is invalid.
The values for `channels` are interpreted as follows, declaring the order and purpose of the channels:

- `1` — FC
- `2` — FL FR
- `3` — FL FR FC
- `4` — FL FR RL RR
- `5` — FL FR RL RR S
- `6` — FL FR FC RL RR S
- `7` — FL FR FC RL RR SL SR
- `8` — FL FR FC RL RR SL SR S
- `9` — FL FR FC RL RR RC SL SR S

Where

- `FL` — Front Left
- `FR` — Front Right
- `FC` — Front Centre
- `RL` — Rear Left
- `RR` — Rear Right
- `RC` — Rear Centre
- `SL` — Side Left
- `SR` — Side Right
- `S` — Subwoofer

Any other value for `channels` is invalid.
The payload must have exactly `FrameCount*Channels*sample-size` number of octets.

3.2.1 Use-Case

This format is useful for raw audio data storage, which means it should be trivial to feed into an audio playback system with minimal overhead. Unlike the traditional uncompressed audio format, Wave, this follows a much clearer and simpler specification with sensible metadata encoding.

3.3 Image

This format is for storing raw image data. Unlike plain data however, it includes a header that completely identifies the pixel data layout and format. The format supports 3D images as well.

```
Header         ::= Width Height Depth channels format
Payload        ::= Layer{Depth}
Layer          ::= Row{Height}
Row            ::= Color{Width}
Color          ::= channel{channel-count}
Width          ::= uint32
Height         ::= uint32
Depth          ::= uint32
format         ::= uint8
               --- A single octet identifying the per-channel data type.
channels       ::= uint8
               --- A single octet identifying the number and order of
                   channels.
channel-count  ::= channels:4
               --- The number of channels indicated by the lower 4 bits
                   of the channels.
format-size    ::= format:4
               --- The number of octets per channel sample.
channel        ::= <format 01: int8
                         | 02: int16
                         | 04: int32
                         | 08: int64
                         | 11: uint8
                         | 12: uint16
                         | 14: uint32
                         | 18: uint64
                         | 22: float16
                         | 24: float32
                         | 28: float64>
               ---  A single-channel colour value in the format
                    indicated by format.
```

Any other value for `format` is invalid.
The values for `channels` are interpreted as follows:

- `01` — `V`
- `02` — `VA`
- `03` — `RGB`
- `04` — `RGBA`
- `12` — `AV`
- `13` — `BGR`
- `14` — `ABGR`

- `24` — `ARGB`
- `34` — `BGRA`
- `44` — `CMYK`
- `54` — `KYMC`

Where

- `V` — Value (Brightness)
- `R` — Red
- `G` — Green
- `B` — Blue
- `A` — Alpha
- `C` — Cyan
- `M` — Magenta
- `Y` — Yellow
- `K` — Black

Any other value for `channels` is invalid.
The payload must have exactly `Width*Height*Depth*channel-count*format-size` number of octets.

The colours are stored in *linear* format without a perceptual colour space or gamma correction in effect. For the floating point formats, values ranged from 0 to 1 correspond to the same intensity of the minimal and maximal values of an unsigned integer format. However, a floating point format file *may* store values beyond that range. Both the floating point and signed integer formats *may* also store negative colour values, though SF3 makes no attempt to specify the perceptual display of these colours. The tone mapping process required to accurately render the colours stored in an SF3 Image in general is application dependent.

3.3.1 Use-Case

This format is useful for storing raw bitmap data that can be directly memory-mapped and read out. This is especially convenient for GPU texture uploads with DirectX, OpenGL, Vulkan, or similar.

3.4 Log

This format is for storing generic logging and event information.

```
Header          ::= StartTime ChunkCount
Payload         ::= Chunk*
Chunk           ::= ChunkSize EntryCount EntryOffset{EntryCount} Entry{EntryCount}
StartTime       ::= int64
                --- A unix-time timestamp specifying when this file
                    begins.
Entry           ::= Size Time Severity Source Category Message
Size            ::= uint32
                --- The size of the remaining log entry in octets.
Time            ::= uint64
                --- The number of milliseconds since StartTime at which
                    this log entry was recorded.
Severity        ::= int8
                --- The severity or importance of the log entry.
Source          ::= source-length string{source-length}
                --- An identifier of the source of the log entry.
source-length   ::= uint8
Category        ::= category-length string{category-length}
                --- An identifier of the category the entry belongs to.
category-length ::= uint8
Message         ::= message-length string{message-length}
                --- A human-readable message describing the event.
message-length  ::= uint16
ChunkCount      ::= uint16
                --- The number of chunks in the file.
ChunkSize       ::= uint64
                --- The octet size of the chunk.
EntryCount      ::= uint32
                --- The number of entries in the chunk.
EntryOffset     ::= uint64
                --- The octet offset of the corresponding Entry from the
                    beginning of Payload.
```

The `severity` should be zero if the message is of neutral importance, positive for increasingly vital information, and negative for increasingly detailed information.

Both the `Source` and `Category` may consist of just a null octet each if the information is not relevant.

The format is designed such that an application can continuously append new entries. To do so, it should behave as follows:

2. Increase the `ChunkCount`
3. Append a new `Chunk` and allocate a number of `EntryOffset`s within the chunk.
4. When a new entry is generated and there are still unused `EntryOffset`s:
   (a) Update the `ChunkSize` and `EntryCount`
   (b) Fill in the current end offset into the corresponding `EntryOffset`.
   (c) Append the new `Entry`.

Otherwise start from 1.

The `EntryOffset`s allow a reading application to scan through the log much more quickly, and perform a binary search to identify date ranges.

3.4.1 Use-Case

This format is useful for basic logging purposes in applications that run for a longer amount of time.The binary format allows quickly skipping ahead in the file to reach interesting messages or to filter out important events.

3.5 Model

This format is for singular triangular meshes only. It does not include a scene graph or the capability for non-triangular or non-static meshes. If animation of the model is desired, animation information can be delivered separately.

```
Header          ::= format material-type MaterialSize
Payload         ::= Material Faces Vertices
MaterialSize    ::= uint32
                --- The octet size of the Material payload.
Material        ::= Texture{material-count}
                --- An array of texture maps for the model's material.
Texture         ::= texture-size string{texture-size}
                --- A relative file path to an image.
texture-size    ::= uint16
Faces           ::= face-count uint32{face-count}
                --- An array of 0-based indices into the Vertices
                    array, every 3 of which describe a face.
face-count      ::= uint32
Vertices        ::= vertex-count float32{vertex-count}
                --- An array of vertices, packed as floats. The count
                    must be a multiple of the float count of an
                    individual vertex.
vertex-count    ::= uint32
Position        ::= float32 float32 float32
                --- A vertex position in model-space.
UV              ::= float32 float32
                --- A texture coordinate in texture-space.
Color           ::= float32 float32 float32
                --- An RGB colour triplet, each channel in [0,1].
Normal          ::= float32 float32 float32
                --- A surface normal, in tangent-space.
Tangent         ::= float32 float32 float32
                --- A surface tangent, in tangent-space.
format          ::= uint8
                --- A single octet identifying the per-vertex format.
material-type   ::= uint8
                --- A single octet identifying the material used.
material-count  ::= material-type#
                --- The number of material textures as indicated by
                    material-type.
vertex          ::= <format 01: Position
                          |  03: Position UV
                          |  05: Position Color
                          |  09: Position Normal
                          |  0B: Position UV Normal
                          |  0D: Position Color Normal
                          |  1B: Position UV Normal Tangent
                          |  1D: Position Color Normal Tangent>
                --- A single-vertex value in the format indicated by
                    format.
```

The values for the vertex `format` encode the set of attributes as a bit set, where:

5. `01` — Position

- `02` — UV
- `04` — Color
- `08` — Normal
- `10` — Tangent

However only the values listed for the `vertex` type are valid.

The values for `material-type` are interpreted as follows, and describe the usage and number of `Textures`:

- `00` — (no material)
- `01` — Albedo
- `03` — Albedo Normal
- `81` — Albedo Emission
- `43` — Albedo Normal Specular
- `83` — Albedo Normal Emission
- `07` — Albedo Normal Metallic
- `1B` — Albedo Normal Metalness Roughness
- `C3` — Albedo Normal Specular Emission
- `87` — Albedo Normal Metallic Emission
- `9B` — Albedo Normal Metalness Roughness Emission
- `3B` — Albedo Normal Metalness Roughness Occlusion
- `BB` — Albedo Normal Metalness Roughness Occlusion Emission

Any other value for `material-type` is invalid. It should be noted that the set of attributes is encoded as a bit set:

- `01` — Albedo
- `02` — Normal
- `04` — Metallic
- `08` — Metalness
- `10` — Roughness
- `20` — Occlusion
- `40` — Specular

- `80` — Emission

The Metallic texture is a combination of Metalness, Roughness, and Occlusion in the R, G, and B channels respectively.

The included `MaterialSize` field should allow constant-time access to the vertex data without having to parse the `Material` structure, if that structure is not needed. If the `Faces` array is empty, then the faces are implicit and every three vertices in the `Vertices` array form a face.

The coordinate system is intended to be right handed with Y+ up, Z- forward.

3.5.1 Use-Case

This format is useful for storing uncompressed, directly accessible 3D geometry data. It is packed in such a way that it should be trivial to upload into vertex-buffers for use with GPU rendering toolkits like DirectX, OpenGL, Vulkan, or similar. For instance, the `format` describes the vertex-array layout, the `Faces` array makes up the element-buffer, and the `Vertices` makes up the vertex-buffer.

3.6 Physics-Model

This format is for storing a series of convex meshes that make up the collision shapes of a more complex model. It is far more efficient at storing this data than the generic model format and allows multiple shapes in one.

```
Header       ::= format hull-count
Payload      ::= mass Tensor shape-count Shape{shape-count}
Tensor       ::= float32{9}
             --- The inertia tensor for the entire model, in row-major
                 order.
Shape        ::= Transform shape-type
                 <shape-type 01: Ellipsoid
                           | 02: Box
                           | 03: Cylinder
                           | 04: Pill
                           | 05: Mesh>
shape-type   ::= uint8
Ellipsoid    ::= float32 float32 float32
             --- The width, height, and depth of the ellipsoid
                 measured from its centre.
Box          ::= float32 float32 float32
             --- The width, height, and depth of the box measured from
                 its centre.
Cylinder     ::= float32 float32 float32
             --- The bottom radius, top radius, and height of the
                 cylinder measured from its centre.
Pill         ::= float32 float32 float32
             --- The bottom radius, top radius, and height of the
                 cylinder measured from its centre.
Mesh         ::= vertex-count Vertex{vertex-count}
             --- A single convex hull as a series of vertices forming
                 its surface.
Transform    ::= float32{16}
             --- The transform matrix describing the offset and
                 orientation of this hull from the model's origin, in
                 row-major order.
Vertex       ::= float32 float32 float32
             --- A single vertex on the convex hull's surface.
shape-count  ::= uint16
             --- The number of shapes that make up the model.
vertex-count ::= uint16
             --- The number of vertices that make up the hull's
                 boundary.
mass         ::= float32
             --- The initial mass of the unscaled model in kg.
```

Each of the implicit shapes (ellipsoid, box, cylinder, pill) are specified with the origin being the shape's centre, and the three values, width, height, and depth, being in X, Y, and Z directions respectively from that centre. Meaning: a cube specified as Width 1, Height 2, Depth 3 has a volume of 48, since each dimension only specifies the half of that side's length. All dimensions must be greater than or equal to zero.

For the cylinder and pill specifically the shapes are oriented Y-up. Meaning: a cylinder's flat sides are oriented Y- and Y+. For the pill the centres of the spheres at its ends are apart from each other by 2*height. If 2*height of the pill is less than the combined radii of both spheres, the collision behaviour is implementation dependent as the shape is no longer well defined.

For the Mesh, only the bounding vertices are specified. In order to recover face data when necessary, an algorithm like Quickhull may be used.

The coordinate system is intended to be right handed with Y+ up, Z- forward.

3.6.1 Use-Case

This format is intended for use in games and other applications that require a convex decomposition of a model for use in collision testing. The packed storage format is ideal for direct use in-engine.

3.7 Table

This format specifies an arbitrary "table" similar to CSV files, albeit with a strict table schema encoded as part of the file.

```
Header        ::= spec-length column-count row-length row-count
Payload       ::= ColumnSpec{column-count} Row{row-count}
spec-length   ::= uint32
              --- The length of the ColumnSpec block in octets.
column-count  ::= uint16
              --- The number of columns per row.
row-length    ::= uint64
              --- The length of every row in octets.
row-count     ::= uint64
              --- The number of rows in the file.
ColumnSpec    ::= name-length string{name-length} column-length Column
name-length   ::= uint16
              --- The length of the column name in octets.
column-length ::= uint32
              --- The length of the column in octets.
Column        ::= column-type <column-type 01: Uint8
                                         | 02: Uint16
                                         | 04: Uint32
                                         | 08: Uint64
                                         | 11: Int8
                                         | 12: Int16
                                         | 14: Int32
                                         | 18: Int64
                                         | 22: Float16
                                         | 24: Float32
                                         | 28: Float64
                                         | 31: String
                                         | 48: Timestamp
                                         | 58: HighResolutionTimestamp
                                         | 61: Boolean>
column-type   ::= uint8
              --- The type of the column data.
element-size  ::= column-type:4
              --- The number of octets per "element" of the column.
Uint8         --- Denotes an array of unsigned 8-bit integers.
Uint16        --- Denotes an array of unsigned 16-bit integers.
Uint32        --- Denotes an array of unsigned 32-bit integers.
Uint64        --- Denotes an array of unsigned 64-bit integers.
Int8          --- Denotes an array of signed 8-bit integers.
Int16         --- Denotes an array of signed 16-bit integers.
Int32         --- Denotes an array of signed 32-bit integers.
Int64         --- Denotes an array of signed 64-bit integers.
Float16       --- Denotes an array of 16-bit IEEE half precision
                  floating point numbers.
Float32       --- Denotes an array of 32-bit IEEE single precision
                  floating point numbers.
```

```
Float64       --- Denotes an array of 64-bit IEEE double precision
                  floating point numbers.
String        --- Denotes an UTF-8 encoded, null-terminated
                  character string.
Timestamp     --- Denotes an array of signed 64-bit integers that
                  encode Unix-time timestamps.
HighResolutionTimestamp
              --- Denotes an array of unsigned 64-bit integers that
                  encode timestamps as the number of nanoseconds
                  since the Unix epoch.
Boolean       --- Denotes an array of unsigned 8-bit integers where
                  0 is denoted "False" and any other value "True".
Row           ::= uint8{row-length}
```

Usually the `column-length` will fit exactly one element, meaning it will be the same as `element-size`, and each cell will not be an array but rather just one value. The `row-length` must be a sum of all `column-length`s and merely serves as a faster way to index into the file. The `column-length` must be a multiple of `element-size`. And thus, the number of logical elements in a column with an array type can be determined by dividing `column-length` by `element-size`.

The `Row` is specified as an opaque blob in the BNF, but can be trivially decoded according to the layout specified by the `ColumnSpec`s. The data for each column must follow in the same order as the `ColumnSpec`s inside the row, without any gaps between. A decoder can thus simply iterate over the `ColumnSpec`s and decode each "cell" by first determining the number of elements, and then decoding each element in the cell according to the `column-type`.

For `String` columns especially it should be noted that, due to the null termination, the string may be *shorter* than the number of bytes reserved by the `column-length`.

3.7.1 Use-Case

This format lends itself well to structured data that follows a precise schema, especially when new data only needs to be appended such as when recording datapoints. The header fields allow constant-time random access to the rows as well as windowing of the data.

3.8 Text

This format allows for a very simple rich text markup. Primitive displays can also ignore all the markup directly and instead display the text plain without needing special processing to strip the markup out.

```
Header          ::= markup-size
Payload         ::= markup-count Markup{markup-count} text-length string{text-length}
markup-count    ::= uint32
text-length     ::= uint64
markup-size     ::= uint64
                --- The size of the markup block in octets
Markup          ::= Start End Option
                --- A singular markup option
Start           ::= uint64
                --- The (0-based) index of the first codepoint being styled.
End             ::= uint64
                --- The (0-based) index of the last codepoint being styled.
Option          ::= option-type <option-type 01: Bold
                                  | 02: Italic
                                  | 03: Underline
                                  | 04: Strike
                                  | 05: Mono
                                  | 06: Color
                                  | 07: Size
                                  | 08: Heading
                                  | 09: Link
                                  | 0A: Target>
option-type     ::= uint8
                --- A description of the text style.
Bold            --- The font weight should be set to "bold"
Italic          --- The font slant should be set to "italic"
Underline       --- A line should be drawn under the text's baseline.
Strike          --- A line should be drawn between the text's baseline and
                    ascent line.
Mono            --- The font should be set to monospaced mode.
Color           ::= float32 float32 float32
                --- The text colour should be set to this R G B triplet.
Size            ::= float32
                --- The text size should be multiplied by this factor.
Heading         ::= Level
                --- The text should be a heading of this level.
Link            ::= Address
                --- The text should be an interactable link to its address.
Target          ::= Address
                --- The text should be a link target for its address.
Level           ::= uint8
                --- The heading level. The higher, the more deeply nested
                    the heading is.
Address         ::= address-length string{address-length}
                --- Some kind of target identifier. Often a URL string.
address-length  ::= uint16
```

The font family, default font size, background colour, foreground colour, and line wrapping mode are all determined by the visualiser. The visualiser may also apply default alternate styling to sections marked up with the `Link` option. If the `Address` of a `Link` is the same as that of a `Target` option, the `Link` markup should, when interacted with, point the user to the text marked up by the corresponding `Target` option. Otherwise the behaviour of interaction with the `Link` text is up to the implementation.

The `Markup` options may be in any particular order with regards to their `Start` and `End`, and the bounds may also overlap arbitrarily.

3.8.1 Use-Case

This format is useful for storing simple rich text documents that don't require complex layouting or processing.

3.9 Vector Graphic

This format offers a relatively simple but capable vector graphic format for scalable images.

```
Header         ::= Width Height Count
Payload        ::= Instruction{Count}
Width          ::= uint32
               --- The width of the visible canvas in units.
Height         ::= uint32
               --- The height of the visible canvas in units.
Count          ::= uint32
               --- The number of instructions to appear.
Instruction    ::= instruction-type
                   <instruction-type 01: Line
                                   | 02: Rectangle
                                   | 03: Circle
                                   | 04: Polygon
                                   | 05: Curve
                                   | 06: Text
                                   | 11: Identity
                                   | 12: Matrix>
instruction-type  ::= uint8
Line           ::= Color Thickness ShapeOutline
               --- A sequence of connected line segments.
Rectangle      ::= ShapeBounds ShapeFill
               --- An axis-aligned rectangle.
Circle         ::= ShapeBounds ShapeFill
               --- An axis-aligned oval circle.
Polygon        ::= ShapeOutline ShapeFill
               --- A many-edged convex polygon.
Curve          ::= ShapeOutline ShapeFill
               --- A cubic Bezier curve directed by its control points.
Text           ::= Point Color Font FontSize String
               --- A single line of text.
Font           ::= font-length string{font-length}
               --- The name of the font family to use to render the
                   text.
font-length    ::= uint16
String         ::= string-length string{string-length}
               --- The string of text to be displayed
string-length  ::= uint16
Identity       --- A reset of the current transform matrix to the
                   1 0 0, 0 1 0 matrix.
Matrix         ::= float32{6}
               --- A coordinate transform matrix to be applied to
                   subsequent instructions.
ShapeOutline   ::= Edges Point{Edges}
               --- A list of edge points of a composite shape.
ShapeBounds    ::= Point Size
               --- The bounding box of a shape.
ShapeFill      ::= Color Color Thickness
```

```
              --- The fill colour, outline colour, and outline thickness.
Point         ::= float32 float32
              --- A position in x/y.
Size          ::= float32 float32
              --- The bounding size of a shape in width/height.
Color         ::= float32 float32 float32 float32
              --- An RGBA colour quadruplet in the range of [0, 1].
Edges         ::= uint16
              --- The number of points to appear in the edge list.
Thickness     ::= float32
              --- The thickness of the outline in units.
FontSize      ::= float32
              --- The size of an em in units.
```

The coordinate system should be defined with X growing to the right, Y growing upwards, and the origin being in the lower left corner of the canvas. Whenever a `Transform` is applied, the given matrix must be applied to all following `Element`s until the next `Transform`. This transformation applies to the shape as a whole, not just the `Point`s that define it in the file.

When an element with a `ShapeFill` should be drawn, the fill must be drawn first, with the outline (if visible) second on top. If the shape is bounded by `ShapeBounds`, the fill should meet the bounds, and the outline should be centred on the bounds when hitting them. If the shape is bounded by a `ShapeOutline`, the outline must be drawn centred on the lines defined by the `Edge`s.

For `Curve` and `Polygon`, if the `Point`s do not form a closed shape, the fill should be drawn as a closed shape by directly connecting the first and last points with a straight line.

In the case of the `Curve`, the points should be interpreted as follows: `EdgePoint ControlPoint (ControlPoint EdgePoint ControlPoint)* ControlPoint EdgePoint`. As such, the `Edges` number for a `Curve` must always match `x+2 % 3 == 0` and must at least be `4`. Any other value is an error. The `ControlPoint` coordinates are relative to their corresponding `EdgePoint`.

In the case of `Polygon` and `Rectangle`, the `Edges` number must be at least `2`. Any other value is an error.

For the `Size`, `Thickness`, `FontSize`, and `Color`, all float components must be positive and real. Infinities, NaNs, and negative numbers are an error.

For the `Point` and `Matrix`, all float components must be real. Infinities and NaNs are an error. The `Matrix` is specified in row-major order, meaning in order the entries are m00 m01 m02 m10 m11 m12.

When rendering `Text`, the `Point` specifies the location of the middle on the baseline of the first character that is rendered.

If a visualiser or editor of a vector graphic file does not have access to the font specified in a `Font` field, it *should* generate an error, but *may* exchange the font for a similar one. In either case, it *must* inform the user of the missing font.

3.9.1 Use-Case

This format is useful for representing vector graphics in a light-weight way that should be easy to write a visualiser for. It intentionally does not specify much about text processing and instead leaves most of this up to the implementation.

4 Metadata

These formats can be delivered as part of a binary stream or deposited in a file system. The following are recommendation for metadata identifiers to distinguish SF3 data without having to parse it.

4.1 Mime-Type

The mime-types for SF3 files should be as follows, according to the format used:

- **Archive** `application/x.sf3-archive`
- **Audio** `audio/x.sf3`
- **Image** `image/x.sf3`
- **Log** `application/x.sf3-log`
- **Model** `model/x.sf3`
- **Physics-Model** `model/x.sf3-physics`
- **Text** `application/x.sf3-text`
- **Vector Graphic** `image/x.sf3-vector`

If a general SF3 file should be designated, the mime-type should be `application/x.sf3`. If/when the IANA registration for an official mime-type is approved, the `x.` prefix may be dropped.

4.2 File Extension

The file extension should always end with `.sf3`. Specifically, for the formats the following extended extensions *may* be used:

- **Archive** `.ar.sf3`
- **Audio** `.au.sf3`
- **Image** `.img.sf3`
- **Log** `.log.sf3`
- **Model** `.mod.sf3`
- **Physics-Model** `.phys.sf3`
- **Text** `.txt.sf3`
- **Vector Graphic** `.vec.sf3`