Early Vision File Format Standard

Ohad Ben-Shahar

November 2000

1 General Comments

Following the increase in the number of file formats encountered in our group, we have decided to create a new standard file format that (hopefully) can replace them all. This document describes this new format - the Early Vision Format (EVF). Along with the decision to adopt this format it is also agreed that:

1. any new component added to the collection of programs used in our group will be designed and implemented to use this standard format, and if possible, *only* this one.
2. any existing code will be adapted (by means of I/O filters or change in source code) to accept and produce files of this format.
3. the practice of other formats, if needed, will be limited to user interaction (e.g., for display purposes). Filters that convert between EVF and other industry standard formats (like FIG, PS, or PGM) are expected to be developed as a first step towards integrating the new format.

The few "strategic" decisions that shape the general look of the EVF format read as follows:

1. The EVF format should be accessible across platforms with no modifications.
2. One should be able to create, read, alter, or debug an EVF file without any special software (i.e., with a text editor only).
3. The EVF format should be expandable, backward compatible, and as much flexible as possible .

As an immediate consequence, we choose the EVF format to be an ASCII text file (very much in the spirit of, say, Postscript) with the conscious understanding that we pay in size.

It should be noted that in terms of "what is represented", the EVF format is can be tought of a representating a set of points in some high dimensional vector space (whose dimention, coordinate axes, and their properties, are defined within the file itself). This follows the local nature of early vision objects and their processing.

2 The EVF file format grammer

The following is the syntax grammer for the EVF format (without reference to comments which can appear anywhere). Bolded words are tokens and italics are variables. Case of token is not important:

evf	:=	**EVF** *version layout data*
version	:=	*real*
layout	:=	**Layout** *dimensions context* **End Layout**
dimensions	:=	**Dimensions =** *integer axes-spec* **End Dimensions**
context	:=	**Context** *assignments* **End Context** $\mid$ ϵ
axes-spec	:=	*axis axes-spec* $\mid$ ϵ
assignments	:=	*assignment assignments* $\mid$ ϵ
axis	:=	**Axis :** *axis-id axis-class axis-type-spec*
assignment	:=	*var* **=** *val*
axis-id	:=	*string*
axis-class	:=	**Position** $\mid$ **Orientation** $\mid$ ... (see dictionary)
axis-type-spec	:=	*type-id type-modifiers type-range bounding-box*
type-id	:=	**Integer** $\mid$ **Real** $\mid$ **Enumerated** $\mid$ **Quantized** $\mid$ **Sampled**
type-modifiers	:=	(*modifier modifiers*) $\mid$ ϵ
type-range	:=	*range-integers* $\mid$ *range-reals* $\mid$ *range-list* $\mid$ *range-bins*
range-integers	:=	[*integer integer*]
range-reals	:=	[*real real*]
range-list	:=	[*string list*]
range-bins	:=	[*integer integer real real*]
modifiers	:=	*modifier modifiers* $\mid$ ϵ
modifier	:=	**Periodic** $\mid$ **Uniform** $\mid$ ... (see dictionary)
bounding-box	:=	[*number number*]
data	:=	**Data** (*integer* **Records**) *point-list* **End Data**
point-list	:=	*point point-list* $\mid$ ϵ
point	:=	*number point* $\mid$ ϵ
list	:=	*string list* $\mid$ ϵ
var	:=	*string*
val	:=	*number* $\mid$ *string*
number	:=	*integer* $\mid$ *real*
string	:=	[any alphanumeric string (underscores allowed)]
integer	:=	[any integer number]
real	:=	[any real number]

3 The EVF file format - top level structure

While EVF files are extpected to be recognized with the .evf extension (e.g., flower.evf), no such convention should be assumed when trying to interpret the file. The only mean by which EVF files identify themselves is a magic string in their first non comment line.

An EVF file is built of a *layout section* and a *data section*, both of which can contain free text comments (preceeded by the comment character #). The layout section (a.k.a the *header*) contains the specification of the data and how it should be interpreted while the data section is a list of multidimentional points.

The EVF header takes the following general form (with comment lines being a suggestion only and non tokens souranded in < >):

```
#! <path to processing executable>
EVF 1.0
#----------------------------------------------
# Generated by <CLI of generating program>
#----------------------------------------------
Layout

  Dimensions = <D>
    Axis: <Axis spec line>
                :
    Axis: <Axis spec line>
  End Dimensions

  Context
    <context variable>=<value>
        :
    <context variable>=<value>
  End Context

End Layout
#----------------------------------------------
```

The first (optional) comment line is intended to make the file self-executable in the same way that script files are executable in the context of some shell. By specifying the appropriate executabale here, one can make the EVF file self-viewable, for example.

The first non-comment line is a magic word that identifies the EVF as such. We add format version number for future use, and set it initially to 1.0. It is recommended that after the magic word line, a group of comments will describe the creator of the file (by means of the command that created it, with its parameters). This should help, the least, in cases when the file should reproduced with only slight changes.

The `Layout, Dimensions, Axis`, and and `Context`, and their corresponding ending tokens, are expected to appear as shown, although case, identation and white spaces (including line breaks) are not important. The number of axes lines should be equal to `D` ,as specified in the `Dimensions` line. The `Context` section is optional and the number of assignments in it can be arbitrary large.

The EVF body is a list of point objects, all having the same line structure, that is - a list of values separated by white spaces (tabs are ok too). The number of values in each line, and their type (real or integer) should correspond to the specification in the `Dimensions` section. So if the header reads `Dimensions = 4` and includes 4 `Axis` declarations, each data line should contain exactly 4 elements. Comment lines can be integrated freely between data lines, if need arises. Here is an example which corresponds to `Dimension=4`:

```
Data (5 Records)
#-----------------------------------------------
# Here are local measurements related to the
# curve structures in the image
#-----------------------------------------------
113    10.32    45     1.30
1      231.45   67     0.32
#-----------------------------------------------
# Look mom, I can add comments!!!
#-----------------------------------------------
200     25.1    67     3.32
234     0.03    22     2.00   # comments can appear
234     9.73    90     1.10   # at end of lines too
#-----------------------------------------------
# End of object list
#-----------------------------------------------
End Data
```

Note that the total number of data records is specified after the keywork `Data`. This part is *not* optional and must be included. For the sake of readability, it is most recommended that all data lines will be aligned by columns, each column aligned around the decimal point (if applies) and appropriate spacing is used between columns.

4 The `Layout` specification

As mentioned, the layout section specifies the layout of the data and how it should be intertreted. It has two parts, as described below:

4.1 the `Dimensions` subsection

The `Dimensions` section specifies the axes along which the data is drawn. Each `Axis` line takes the following general form:

Axis: ID class type(modifiers) range bounding-box

The **ID** is an arbitrary alphanumeric string that identifies the axis. It is expected that IDs will be unique withing a file and have some descriptive power (so "line" and "contrast" are better than "xg32" or "foo"). Among their other functions, these IDs should be used to distinguish between coordinates that otherwise have identical specification (e.g., between X and Y coordinate of an iname pixel).

The **class** is one of few classes pre-defined in the EVF dictionary. It's role (along woth the ID field) is in associating some meaning and units to the axis and its corresponding data. For example, the class will allow a semantic distinction between position and dimension, both of which may look like real numbers). While the set of suppored classes should be defined in a centrally maintained "dictionary", the addition of new classes should not result in a major coding effort. Rather, the update procedure is expected to include an addition of a new dictionary entry (in the EVF library interface file, e.g. `evf.h`) and recompilation of the library).

The **type** is one of few predefined types that specifies how the data is listed. Current types include integer and real numbers, enumerated types, and quantized and sampled data. The type specification can be further refined with an optinal list of modifiers. Unlike the set of classes, adding new types will require some coding effort and hence should be avoided.

The **range** specification details the range of values permitted within the axis. Its format depends on the types itself:

- ranges of `Real` types are specified by a pair of real numbers - the min and max of the permitted values.
- ranges of `Integer` types are specified by a pair of integer numbers - the min and max of the permitted values.
- ranges of `Quantized` types are specified with 4 numbers - a min and max bin values (both integers), the real value that corresponds to the center of the zero bin, and the step size between adjacent bins. This specification allows both boundary definition and the association of the quantized bin numbers to the real world data from which they emerged.
- ranges of `Sampled` types are specified in the same way of `Quantized` ranges, although now the "bin" numbers do not represent a whole interval but rather a single value on the real axis.
- ranges of `Enumerated` types are a list of permitted values. This data type will not be supported in early implementations.

Finally, the **bounding box** specifies the range of values of this axis that are actually found in the current file. The bounding box is expected to be a subset of the range and the type of both numbers should comply with the type of the first two numbers of the range specification.

Example of a complete `Dimension` section is the following:

```
Dimensions=4
  Axis: X       Position    Sampled             [0 128 0.00  1.00] [ 5 111]
  Axis: Y       Position    Sampled             [0 128 0.00  1.00] [20 90]
  Axis: MyEdge  Orientation Quantized(Periodic) [0  15 0.00 22.50] [ 0 15]
  Axis: k       Curvature   Real                [-5.00 5.00]       [-2.12 4.15]
End Dimensions
```

4.2 the `Context` subsection

This optional part is intended to allow a description of the context in which the data was produce, or important information required to make the correct interpretation of the data. The context is just a list of values assigned to variables. Naturally, in order to useful, the names and values of the context variables should follow some protocol agreed between the data producer and the client.

Example of a context section is the following:

```
Context
  FractalDimension    = Minkowski
  DilationWindowSize  = 10x10
  EdgeOperator        = LogicalLiner
End Context
```

5 EVF keywords dictionary

This section lists all keywords known under the EVF standard, with their corresponding data type and semantic. Note that few of these keywords are relevant only to the `GlobalProperties` section of the header (although nothing in the format description prevent them from appearing as part of the objects too).

Keyword	units	Semantic
Position	mm	the position coordinate of the object
Orientation	degrees	orientation
Curvature	$1/mm$	curvature
Dimension	unitless	local effective dimension
MORE.....	????	???????????????