

Tiny Vector Graphics (Specification)


Abstract: The tiny vector graphics format is a binary file format that encodes a list of vector graphic primitives. It is tailored to have a tiny memory footprint and simple implementations, while lifting small file size over encoding simplicity.


Introduction

Why a new format

SVG is the status quo widespread vector format. Every program can kinda use it and can probably render it right to some extend. The problem is that SVG is a horribly large specification, it is based on XML and provides not only vector graphics, but also a full suite for animation and JavaScript scripting. Implementing a new SVG renderer from scratch is a tremendous amount of work, and it is hard to get it done right.

Quoting the [german Wikipedia](https://de.wikipedia.org/wiki/Scalable_Vector_Graphics) (https://de.wikipedia.org/wiki/Scalable_Vector_Graphics):

 Praktisch alle relevanten Webbrowser können einen Großteil des Sprachumfangs darstellen.

 Virtually all relevant web browsers can display a large part of the language range.

The use of XML bloats the files by a huge magnitude and doesn't provide a efficient encoding, thus a lot of websites and applications ship files that are not encoded optimally. Also SVG allows several ways of achieving the same thing, and can be seen more as a intermediate format for editing as for final encoding.

TinyVG was created to adress most of these problems, trying to achieve a balance between flexibility and file size, while keeping file size as the more important priority.

Features

- Binary encoding
- Support of the most common 2D vector primitives
 - Paths
 - Polygons
 - Rectangles
 - Lines
- 3 different fill styles

- Flat color
 - Linear 2-point gradient
 - Radial 2-point gradient
- Dense encoding, there are near zero padding bits and every byte is used as good as possible.

Format

TVG files are roughly structured like this:

Header
Color Table
Command
Command
Command
Command
End Of File

Files are made up of a header, followed by a color lookup table and a sequence of commands terminated by a *end of file* command.

Concrete color values will only be present in the color table. After the table, only indices into the color table are used to define color values. This allows to keep the format small, as the first 128 colors in the vector data are encoded as only a single byte, even if the color format uses 16 bytes per color. This means in the worst case, we add a single byte to the size of a color that is only used once, but colors that are common in the file will be encoded as a single byte per use + one time overhead. This encoding scheme was chosen as a vector graphic typically doesn't use as much different colors as bitmap graphics and thus can be encoded more optimally.

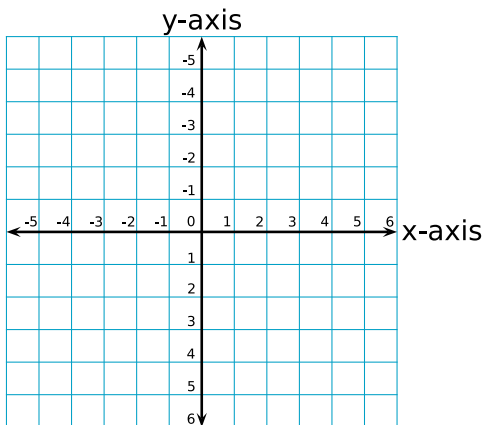
NOTE: The following documentation uses a tabular style to document structures. All integers are assumed to be encoded in little-endian byte order if not specified otherwise.

The *Type* column of each structure definition uses a Zig notation for types and the fields have no padding bits inbetween.

If a field does not align to a byte boundary, the next field will be offset into the byte by the current fields bit offset + bit size. This means, that two consecutive fields **a** (u3) and **b** (u5) can be extracted from the byte by using $(\text{byte} \& 0x7) \gg 0$ for **a** and $(\text{byte} \& 0x1F) \gg 3$ for **b**.

Coordinate system

TinyVG uses the 2-dimensional [Cartesian coordinate system](https://en.wikipedia.org/wiki/Cartesian_coordinate_system) (https://en.wikipedia.org/wiki/Cartesian_coordinate_system) with X being the positive horizontal distance to the origin and Y being the negative vertical distance to the origin. This means that X is going right, while Y is going down, to match the coordinate system of several other image formats:



Header

Each TVG file starts with a header defining some global values for the file like scale and image size. The header is always at offset 0 in a file.

Field	Type	Description
magic	[2]u8	Must be { 0x72, 0x56 }
version	u8	Must be 1. For future versions, this field might decide how the rest of the format looks like.
scale	u4	Defines the number of fraction bits in a Unit value.
color_encoding	u2	Defines the type of color information that is used in the color table .
coordinate_range	u2	Defines the number of total bits in a Unit value and thus the overall precision of the file.
width	u8, u16 or u32	Encodes the maximum width of the output file in pixels. A value of 0 indicates that the image has the maximum possible width. The size of this field depends on the coordinate range field.
height	u8, u16 or u32	Encodes the maximum height of the output file in pixels. A value of 0 indicates that the image has the maximum possible height. The size of this field depends on the coordinate range field.
color_count	VarUInt	The number of colors in the color table .

Color Encoding

The color encoding defines which format the colors in the color table will have:

Value	Enumeration	Description
0	RGBA 8888	Each color is a 4-tuple (red, green ,blue, alpha) of bytes with the color channels encoded in sRGB and the alpha as linear alpha.
1	RGB 565	Each color is encoded as a 3-tuple (red, green, blue) with 16 bit per color. While red and blue both use 5 bit, the green channel uses 6 bit. red uses bit range 0...4, green bits 5...10 and blue bits 11...15.
2	RGBA F32	Each color is a 4-tuple (red, green ,blue, alpha) of binary32 IEEE 754 floating point value with the color channels encoded in sRGB and the alpha as linear alpha. A color value of 1.0 is full brightness, while a value of 0.0 is zero brightness.
3	Custom	The custom color encoding is <i>defined undefined</i> . The information how these colors are encoded must be implemented via external means.

Coordinate Range

The coordinate range defines how many bits a [Unit](#) value uses:

Value	Enumeration	Description
0	Default	Each Unit takes up 8 bit.
1	Reduced	Each Unit takes up 16 bit.
2	Enhanced	Each Unit takes up 32 bit.

VarUInt

This type is used to encode 32 bit unsigned integers while keeping the number of bytes low. It is encoded as a variable-sized integer that uses 7 bit per byte for integer bits and the 7th bit to encode that there is "more bits available".

The integer is still built as a little-endian, so the first byte will always encode bits 0...6, the second one encodes 8...13, and so on. Bytes are read until the upper-most bit in the byte is set. The bit mappings are done as following:

Byte	Bit Range	Notes
#1	0...6	This byte must always be present.
#2	7...13	
#3	14...20	
#4	21...27	
#5	28...31	This byte must always have the uppermost 4 bits set as 0b1000????.

So a VarUInt always has between 1 and 5 bytes while mapping the full range of a 32 bit value. This means we only have 5 bit overhead in the worst case, but for all smaller values, we reduce the number of bytes for encoding unsigned integers.

Example Code

```
fn read() u32 {
    var count = 0;
    var result = 0;
    while (true) {
        const byte = readByte();
        const val = (byte & 0x7F) << (7 * count);
        result |= val;
        if ((byte & 0x80) == 0)
            break;
        count += 1;
    }
    return result;
}

fn write(value: u32) void {
    var iter = value;
    while (iter >= 0x80) {
        writeByte(0x80 | (iter & 0x7F));
        iter >>= 7;
    }
    writeByte(iter);
}
```

Color Table

The color table encodes the palette for this file. It's binary content is defined by the `color_encoding` field in the header. For the three defined color encodings, each will yield a list of `color_count` RGBA tuples.

RGBA 8888

Each color value is encoded as a sequence of four bytes:

Field	Type	Description
red	u8	Red color channel between 0.0 and 1.0, mapped to byte values 0 to 255.
green	u8	Green color channel between 0.0 and 1.0, mapped to byte values 0 to 255.
blue	u8	Blue color channel between 0.0 and 1.0, mapped to byte values 0 to 255.
alpha	u8	Transparency channel between 0.0 and 1.0, mapped to byte values 0 to 255.

The size of the color table is $4 * \text{color_count}$.

RGB 565

Each color value is encoded as a sequence of 2 bytes:

Field	Type	Description
red	u5	Red color channel between 0.0 and 1.0, mapped to integer values 0 to 31.
green	u6	Green color channel between 0.0 and 1.0, mapped to integer values 0 to 63.
blue	u5	Blue color channel between 0.0 and 1.0, mapped to integer values 0 to 31.

The size of the color table is $2 * \text{color_count}$, and all colors are fully opaque.

RGBA F32

Each color value is encoded as a sequence of 16 bytes:

Field	Type	Description
red	f32	Red color channel between 0.0 and 1.0.
green	f32	Green color channel between 0.0 and 1.0.
blue	f32	Blue color channel between 0.0 and 1.0.
alpha	f32	Transparency channel between 0.0 and 1.0.

The size of the color table is $16 * \text{color_count}$.

Custom

The TinyVG specification does not describe the size nor format of this kind of color table. A implementation specific format is expected. A conforming parser is allowed to reject files with this color format as "unsupported".

Commands

TinyVG files contain a sequence of draw commands that must be executed in the defined order to get the final result. Each draw command adds a new 2D primitive to the graphic.

The following commands are available:

Command Index	Name	Short description
0	end of document	This command determines the end of file.
1	fill polygon	This command fills a N-gon.
2	fill rectangles	This command fills a set of rectangles.
3	fill path	This command fills a free-form path.
4	draw lines	This command draws a set of lines.
5	draw line loop	This command draws the outline of a polygon.
6	draw line strip	This command draws a list of end-to-end lines.
7	draw line path	This command draws a free-form path.
8	outline fill polygon	This command combines the fill and draw polygon command into one.
9	outline fill rectangles	This command combines the fill and draw rectangles command into one.
10	outline fill path	This command combines the fill and draw path command into one.

Each command is encoded as a single byte which is split into fields:

Field	Type	Description
command_index	u6	The command that is encoded next. See table above.
primary_style_kind	u2	The type of style this command uses as a primary style.

End Of Document

If this command is read, the TinyVG file has ended. This command must have `primary_style_kind` to be set to 0, so the last byte of every TinyVG file is 0x00.

Every byte after this command is considered not part of the TinyVG data and can be used for other purposes like meta data or similar.

Fill Polygon

Fills a [polygon](https://en.wikipedia.org/wiki/Polygon) (<https://en.wikipedia.org/wiki/Polygon>) with N points.

The command is structured like this:

Field	Type	Description
point_count	VarUInt	The number of points in the polygon. This value is offset by 1.
fill_style	Style(primary_style_kind)	The style that is used to fill the polygon.
polygon	[point_count + 1] Point	The points of the polygon.

The offset in `point_count` is there due to 0 points don't make sense at all and the command could just be skipped instead of encoding it with 0 points. The offset is one to allow code sharing between other fill commands, as each fill command shares the same header.

`point_count` must be at least 2, files that encode a lower value must be discarded as "invalid" by a conforming implementation.

The polygon specified in `polygon` must be drawn using the [even-odd rule](https://en.wikipedia.org/wiki/Even%E2%80%93odd_rule) (https://en.wikipedia.org/wiki/Even%E2%80%93odd_rule), that means that if for any point to be inside the polygon, a line to infinity must cross an even number of polygon segments.

Point

Points are a X and Y coordinate pair:

Field	Type	Description
x	Unit	Horizontal distance of the point to the origin.
y	Unit	Vertical distance of the point to the origin.

Units

The unit is the common type for both positions and sizes in the vector graphic. It is encoded as a signed integer with a configurable amount of bits (see [Coordinate Range](#)) and fractional bits.

The file header defines a *scale* by which each signed integer is divided into the final value. For example, with a *reduced* value of 0x13 and a scale of 4, we get the final value of 1.1875, as the number is interpreted as binary b0001.0011.

Fill Rectangles

Fills a list of rectangles.

The command is structured like this:

Field	Type	Description
rectangle_count	VarUInt	The number of rectangles. This value is offset by 1.
fill_style	<code>Style(primary_style_kind)</code>	The style that is used to fill all rectangles.
rectangles	<code>[rectangle_count + 1]Rectangle</code>	The points of the polygon.

The offset in `rectangle_count` is there due to 0 rectangles don't make sense at all and the command could just be skipped instead of encoding it with 0 rectangles. The offset is one to allow code sharing between other fill commands, as each fill command shares the same header.

The rectangles must be drawn first to last, so in the order they appear in the file.

Rectangle

Field	Type	Description
x	Unit	Horizontal distance of the left side to the origin.
y	Unit	Vertical distance of the upper side to the origin.
width	Unit	Horizontal extent of the rectangle.
height	Unit	Vertical extent of the rectangle origin.

Fill Path

Fills a [path](#). Paths are described further below in more detail to keep this section short.

The command is structured like this:

Field	Type	Description
segment_count	VarUInt	The number of segments in the path. This value is offset by 1.
fill_style	<code>Style(primary_style_kind)</code>	The style that is used to fill the path.
path	<code>Path(segment_count + 1)</code>	A path with <code>segment_count</code> segments

The offset in `segment_count` is there due to 0 segments don't make sense at all and the command could just be skipped instead of encoding it with 0 segments. The offset is one to allow code sharing between other fill commands, as each fill command shares the same header.

For the filling, all path segments are considered a polygon each (drawn with even-odd rule) that, when overlap, also perform the even odd rule. This allows the user to carve out parts of the path and create arbitrarily shaped surfaces.

Draw Lines

Draws a set of lines.

The command is structured like this:

Field	Type	Description
line_count	VarUInt	The number of rectangles. This value is offset by 1.
line_style	<code>Style(primary_style_kind)</code>	The style that is used to draw the all rectangles.
line_width	<code>f32</code>	The width of the line.
lines	<code>[line_count + 1]Line</code>	The set of lines.

Draws `line_count + 1` lines with `line_style`. Each line is `line_width` units wide, and at least a single display pixel. This means that `line_width` of 0 is still visible, even though only marginally. This allows very thin outlines.

Line

Field	Type	Description
start	Point	Start point of the line
end	Point	End point of the line.

Draw Line Loop

Draws a polygon.

The command is structured like this:

Field	Type	Description
point_count	VarUInt	The number of points. This value is offset by 1.
line_style	<code>Style(primary_style_kind)</code>	The style that is used to draw the all rectangles.
line_width	<code>f32</code>	The width of the line.
points	<code>[point_count + 1]Point</code>	The points of the polygon.

Draws `point_count + 1` lines with `line_style`. Each line is `line_width` units wide, and at least a single display pixel. This means that `line_width` of 0 is still visible, even though only marginally. This allows very thin outlines.

The lines are drawn between consecutive points as well as the first and the last point.

Draw Line Strip

Draws a list of consecutive lines.

The command is structured like this:

Field	Type	Description
<code>point_count</code>	<code>VarUInt</code>	The number of points. This value is offset by 1.
<code>line_style</code>	<code>Style(primary_style_kind)</code>	The style that is used to draw the all rectangles.
<code>line_width</code>	<code>f32</code>	The width of the line.
<code>points</code>	<code>[point_count + 1]Point</code>	The points of the polygon.

Draws `point_count + 1` lines with `line_style`. Each line is `line_width` units wide, and at least a single display pixel. This means that `line_width` of 0 is still visible, even though only marginally. This allows very thin outlines.

The lines are drawn between consecutive points, but contrary to *Draw Line Loop*, the first and the last point are not connected.

Draw Line Path

Draws a [path](#). Paths are described further below in more detail to keep this section short.

The command is structured like this:

Field	Type	Description
<code>segment_count</code>	<code>VarUInt</code>	The number of segments in the path. This value is offset by 1.
<code>line_style</code>	<code>Style(primary_style_kind)</code>	The style that is used to draw the all rectangles.
<code>line_width</code>	<code>f32</code>	The width of the line.
<code>path</code>	<code>Path(segment_count + 1)</code>	A path with <code>segment_count</code> segments

The outline of the path is `line_width` units wide, and at least a single display pixel. This means that `line_width` of 0 is still visible, even though only marginally. This allows very thin outlines.

Outline Fill Polygon

Fills a polygon and draws a outline at the same time.

The command is structured like this:

Field	Type	Description
segment_count	u6	The number of points in the polygon. This value is offset by 1.
secondardy_style_kind	u2	The secondary style used in this command.
fill_style	Style(primary_style_kind)	The style that is used to fill the polygon.
line_style	Style(secondardy_style_kind)	The style that is used to draw the outline of the polygon.
line_width	f32	The width of the line.
points	[segment_count + 1]Point	The set of points of this polygon.

This command is a combination of *Fill Polygon* and *Draw Line Loop*. It first performs a *Fill Polygon* with the `fill_style`, then performs *Draw Line Loop* with `line_style` and `line_width`.



The outline commands use a reduced number of elements, the maximum number of points is 64.

Outline Fill Rectangles

Fills and outlines a list of rectangles.

The command is structured like this:

Field	Type	Description
rectangle_count	u6	The number of rectangles. This value is offset by 1.
secondardy_style_kind	u2	The secondary style used in this command.
fill_style	Style(primary_style_kind)	The style that is used to fill the polygon.
line_style	Style(secondardy_style_kind)	The style that is used to draw the outline of the polygon.
line_width	f32	The width of the line.
rectangles	[rectangle_count + 1]Rectangle	The set of points of this polygon.

For each rectangle, it is first filled, then its outline is drawn, then the next rectangle is drawn. This allows to overlap rectangles to look like this:



The outline commands use a reduced number of elements, the maximum number of points is 64.

Outline Fill Path

Fills a path and draws a outline at the same time.

The command is structured like this:

Field	Type	Description
segment_count	u6	The number of points in the polygon. This value is offset by 1.
secondardy_style_kind	u2	The secondary style used in this command.
fill_style	Style(primary_style_kind)	The style that is used to fill the polygon.
line_style	Style(secondardy_style_kind)	The style that is used to draw the outline of the polygon.
line_width	f32	The width of the line.
path	Path(segment_count + 1)	The set of points of this polygon.

This command is a combination of *Fill Path* and *Draw Path*. It first performs a *Fill Path* with the `fill_style`, then performs *Draw Path* with `line_style` and `line_width`.

The outline commands use a reduced number of elements, the maximum number of points is 64.

Style(style_type)

There are three types of style available:

Value	Style Type	Description
0	Flat Colored	The shape is uniformly colored with a single color.
1	Linear Gradient	The shape is colored with a linear gradient.
2	Radial Gradient	The shape is colored with a radial gradient.

Left to right the three gradient types:



Flat Colored

Field	Type	Description
color_index	VarUInt	The index into the color table

The shape is uniformly colored with the color at `color_index` in the color table.

Linear Gradient

Field	Type	Description
point_0	Point	The start point of the gradient.
point_1	Point	The end point of the gradient.
color_index_0	VarUInt	The color at point_0.
color_index_1	VarUInt	The color at point_1.

The gradient is formed by a mental line between point_0 and point_1. The color at point_0 is the color at color_index_0 in the color table, the color at point_1 is the color at color_index_1 in the color table.

On the line, the color is linearly interpolated between the two points. Each point that is not on the line is orthogonally projected to the line and the color at that point is sampled. Points that are not projectable onto the line have either the color at point_0 if they are closed to point_0 or vice versa for point_1.

Radial Gradient

Field	Type	Description
point_0	Point	The start point of the gradient.
point_1	Point	The end point of the gradient.
color_index_0	VarUInt	The color at point_0.
color_index_1	VarUInt	The color at point_1.

The gradient is formed by a mental circle with the center at point_0 and point_1 being somewhere on the circle outline. Thus, the radius of said circle is the distance between point_0 and point_1.

The color at point_0 is the color at color_index_0 in the color table, the color on the circle outline is the color at color_index_1 in the color table.

If a sampled point is inside the circle, a linear color interpolation is done based on the distance to the center and the radius. If the point is not in the circle itself, the color at color_index_1 is always taken.

Path(segment_count)

Paths describe instructions to create complex 2D graphics.

The mental model to form the path is this:

Each path segment generates a shape by moving a "pen" around. The path this "pen" takes is the outline of our segment. Each segment, the "pen" starts at a defined

position and is moved by instructions. Each instruction will leave the "pen" at a new position. The line drawn by our "pen" is the outline of the shape.

The following instructions to move the "pen" are available:

Instruction Index	Instruction	Short Description
0	line	A straight line is drawn from the current point to a new point.
1	horizontal line	A straight horizontal line is drawn from the current point to a new x coordinate.
2	vertical line	A straight vertical line is drawn from the current point to a new y coordiante.
3	cubic bezier	A cubic bezier curve is drawn from the current point to a new point.
4	arc circle	A circle segment is drawn from current point to a new point.
5	arc ellipse	An ellipse segment is drawn from current point to a new point.
6	close path	The path is closed and a straight line is drawn to the starting point.
7	quadratic bezier	A quadratic bezier curve is drawn from the current point to a new point.

As path encoding is hard to describe in a tabular manner, a verbal one is chosen:

1. For each segment in the path, the number of commands is encoded as a `VarUInt`.
2. For each segment in the path:
 1. A Point is encoded as the starting point.
 2. The instructions for this path, the number is determined in the first step.
 3. Each instruction is prefixed by a single tag byte that encodes the kind of instruction as well as the information if a line width is present.
 4. If a line width is present, that line width is read as a `Unit`
 5. The data for this command is decoded.

The tag looks like this:

Field	Type	Description
instruction	u3	The instruction kind as listed in the table above.
<i>padding</i>	u1	Always 0
has_line_width	u1	If 1, a line width is present.
<i>padding</i>	u3	Always 0

Line

The line instruction draws a straight line to the position.

Field	Type	Description
position	Point	The end point of the line.

Horizontal Line

The horizontal line instruction draws a straight horizontal line to a given x coordinate.

Field	Type	Description
x	Unit	The new x coordinate.

Vertical Line

The vertical line instruction draws a straight vertical line to a given y coordinate.

Field	Type	Description
y	Unit	The new y coordinate.

Cubic Bezier

The cubic bezier instruction draws a b ezier curve with two control points.

Field	Type	Description
control_0	Point	The first control point.
control_1	Point	The second control point.
point_1	Point	The end point of the b�ezier curve.

The curve is drawn between the current location and point_1 with control_0 being the first control point and control_1 being the second one.

Arc Circle

Draws a circle segment between the current and the target point.

Field	Type	Description
large_arc	u1	If 1, the large portion of the circle segment is drawn
sweep	u1	Determines if the circle segment is left- or right bending.
padding	u6	Always 0.
radius	Unit	The radius of the circle.
target	Point	The end point of the circle segment.

radius determines the radius of the circle. If the distance between the current point and target is larger than radius, the distance is used as the radius.

When large_arc is 1, the larger circle segment is drawn.

If sweep is 1, the circle segment will make a left turn, otherwise it will make a right turn. This means that if we go from the current point to target, a rotation to the movement direction is necessary to either the left or the right.

Arc Ellipse

Close Path

A straight line is drawn to the start location of the current segment. This instruction doesn't have additional data encoded.

Quadratic Bezier

The quadratic bezier instruction draws a bézier curve with a single control point.

Field	Type	Description
control	Point	The control point.
point_1	Point	The end point of the bézier curve.

The curve is drawn between the current location and point_1 with control being the control point.

Revision History

1.0

- Initial release