Specification

(Working Draft)

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 extent. 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:

> 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 an intermediate format for editing as for final encoding.

TinyVG was created to address 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.

Recognizing TinyVG

TinyVG is using the .tvg file extension and should use the image/tinyvg mime type.

The textual representation should use the .tvgt file extension and the text/tinyvg mime type.

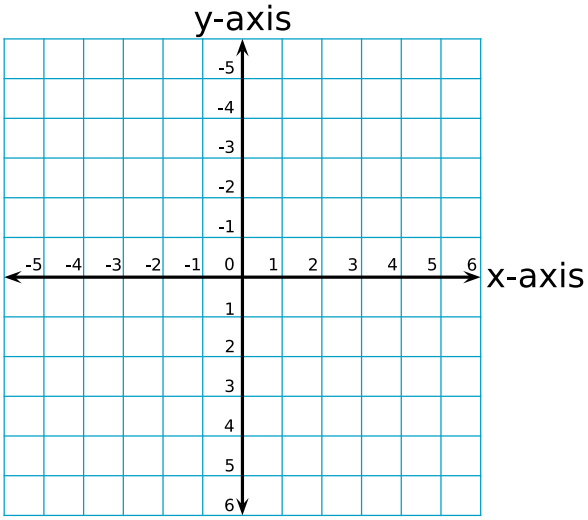
Display Units

Contrary to pixel graphics, vector graphics don't have a inherent unit system. While pixels in a bitmap map 1:1 to pixels on a screen, a vector graphic unit does not have this requirement.

TinyVG uses an abstract unit called *display unit* which is defined to be a 1/96th of an inch. This matches the CSS pixel definition so a TinyVG graphic with 48x48 display units will match a typical 48x48 bitmap.

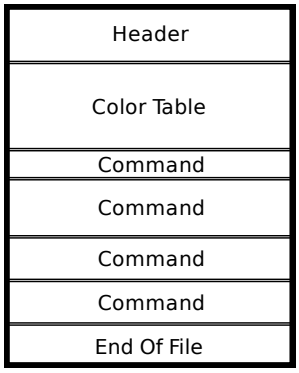
Coordinate system

TinyVG uses the 2-dimensional 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:



Binary Encoding

TinyVG files are roughly structured like this:



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 many different colors as bitmap graphics and thus can be encoded more optimally.

Notes

- 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 in between. 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 (byte & 0x7) >> 0 for **a** and (byte & 0xF8) >> 3 for **b**.
- If not specified otherwise, all coordinates in TinyVG are absolute coordinates, including path nodes and gradients.

- A lot of encoded integers are encoded off-by-one, thus mapping 0 to 1, 1 to 2 and so on. This is done as encoding these integers as 0 would be equivalent to removing the element from the file. Thus, this can be used to encode some more elements with less bytes. If this is the case, this is signaled by the use of value+1.

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 *display units*. 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 *display units*. 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. This color also uses the sRGB color space.
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 scRGB and the alpha as linear alpha. A color value of 1.0 is full intensity, while a value of 0.0 is zero intensity.
3	Custom	The custom color encoding is *defined undefined*. 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 16 bit.
1	Reduced	Each Unit takes up 8 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 uppermost bit in the byte is not 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 0b0000????.

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.

Encoding Examples The following table contains some examples on how a VarUInt is encoded as a byte sequence. The byte sequence is written in hexadecimal to allow uniform notation.

Integer Value	Byte Sequence
0	00
100	64
127	7F
128	80 01
16271	8F 7F
16383	FF 7F
16384	80 80 01
1048576	80 80 40
2097151	FF FF 7F
2097152	80 80 80 01
2147483648	80 80 80 80 08
4294967295	FF FF FF FF 0F

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 and 100% intensity, mapped to byte values 0 to 255.
green	u8	Green color channel between 0 and 100% intensity, mapped to byte values 0 to 255.
blue	u8	Blue color channel between 0 and 100% intensity, mapped to byte values 0 to 255.
alpha	u8	Transparency channel between 0 and 100% transparency, mapped to byte values 0 to 255.

The size of the color table is 4 * color_count.

This color encoding uses the sRGB color space.

RGB 565

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

Field	Type	Description
red	u5	Red color channel between 0 and 100% intensity, mapped to integer values 0 to 31.
green	u6	Green color channel between 0 and 100% intensity, mapped to integer values 0 to 63.
blue	u5	Blue color channel between 0 and 100% intensity, mapped to integer values 0 to 31.

The size of the color table is 2 * color_count, and all colors are fully opaque.

This color encoding uses the sRGB color space.

RGBA F32

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

Field	Type	Description
red	f32	Red color channel, using 0.0 for 0% intensity and 1.0 for 100% intensity.
green	f32	Green color channel, using 0.0 for 0% intensity and 1.0 for 100% intensity.
blue	f32	Blue color channel, using 0.0 for 0% intensity and 1.0 for 100% intensity.
alpha	f32	Transparency channel between 0 and 100% transparency, mapped to byte values 0.0 to 1.0.

The size of the color table is 16 * color_count.

This color encoding uses the scRGB color space, so the intensity is allowed to be both negative and positive for a wider color gamut.

Custom

The TinyVG specification does not describe the size nor format of this kind of color table. An 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:

Index	Name	Short description
0	end of document	This command determines the end of file.
1	fill polygon	This command fills an 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 draws a filled polygon with an outline.
9	outline fill rectangles	This command draws several filled rectangles with an outline.
10	outline fill path	This command combines the fill and draw line 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.
prim_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 prim_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 metadata or similar.

Fill Polygon

Fills a 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(prim_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 not making any sense at all, and the command could just be skipped instead of encoding it with 0 points. The offset is 1 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, that means that if for any point to be inside the polygon, a line to infinity must cross an odd 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 interpretet 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	Style(prim_style_kind)	The style that is used to fill all rectangles.
rectangles	[rectangle_count+1]Rectangle	The list of rectangles to be filled.

The offset in rectangle_count is there due to 0 rectangles not making any sense at all, and the command could just be skipped instead of encoding it with 0 rectangles. The offset is 1 to allow code sharing between other fill commands, as each fill command shares the same header.

The rectangles must be drawn first to last, which is 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	Style(prim_style_kind)	The style that is used to fill the path.
path	Path(segment_count+1)	A path with segment_count 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 1 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	Style(prim_style_kind)	The style that is used to draw the all rectangles.
line_width	Unit	The width of the line.
lines	[line_count + 1]Line	The list 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	Style(prim_style_kind)	The style that is used to draw the all rectangles.

Field	Type	Description
line_width	Unit	The width of the line.
points	[point_count + 1]Point	The points of the polygon.

Draws point_count + 1 lines with line_style. Each line is line_width units wide.

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
point_count	VarUInt	The number of points. This value is offset by 1.
line_style	Style(prim_style_kind)	The style that is used to draw the all rectangles.
line_width	Unit	The width of the line.
points	[point_count + 1]Point	The points of the line strip.

Draws point_count + 1 lines with line_style.

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
segment_count	VarUInt	The number of segments in the path. This value is offset by 1.
line_style	Style(prim_style_kind)	The style that is used to draw the all rectangles.
line_width	Unit	The width of the line.
path	Path(segment_count + 1)	A path with segment_count segments.

The outline of the path is line_width units wide.

Outline Fill Polygon

Fills a polygon and draws an 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.
sec_style_kind	u2	The secondary style used in this command.
fill_style	Style(prim_style_kind)	The style that is used to fill the polygon.
line_style	Style(sec_style_kind)	The style that is used to draw the outline of the polygon.
line_width	Unit	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.

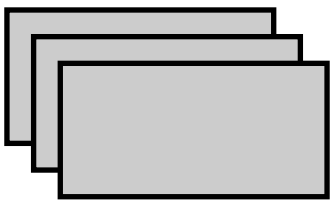
Outline Fill Rectangles

Fills and outlines a list of rectangles.

The command is structured like this:

Field	Type	Description
rect_count	u6	The number of rectangles. This value is offset by 1.
sec_style_kind	u2	The secondary style used in this command.
fill_style	Style(prim_style_kind)	The style that is used to fill the polygon.
line_style	Style(sec_style_kind)	The style that is used to draw the outline of the polygon.
line_width	Unit	The width of the line.
rectangles	[rect_count+1]Rectangle	The list of rectangles to be drawn.

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 an 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.
sec_style_kind	u2	The secondary style used in this command.
fill_style	Style(prim_style_kind)	The style that is used to fill the polygon.
line_style	Style(sec_style_kind)	The style that is used to draw the outline of the polygon.
line_width	Unit	The width of the line.
path	Path(segment_count+1)	The path that should be drawn.

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

See the Color Interpolation chapter on how to perform the color interpolation in detail.

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, the color is interpolated 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.

See the Color Interpolation chapter on how to perform the color interpolation in detail.

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:

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 Bézier 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 Bézier 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-1. Decoding a 0 means that 1 element is stored in the segment.
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.
padding	u1	Always 0
has_line_width	u1	If 1, a line width is present.
padding	u3	Always 0

Path encoding example As this is a very untypical kind of encoding, the following example will showcase how a path is encoded. The path will have 3 segments of different length. For conciseness, the encoding of each individual path component is left out. The unit format is default (16 bit coordinates) with 2 bit precision.

Component	Byte Sequence	Meaning
Segment 0 length	02	Number of elements in segment 0 is 3
Segment 1 length	03	Number of elements in segment 1 is 4
Segment 2 length	80 10	Number of elements in segment 2 is 129
Segment 0 start	00 00 00 00	Segment 0 starts at (0, 0)
Segment 0 command 0	03 …	Draw cubic bezier to …
Segment 0 command 1	02 …	Draw vertical line to …
Segment 0 command 2	00	Close path
Segment 1 start	90 10 38 FF	Segment 1 starts at (100, -50)
Segment 1 command 0	84 0A 00 …	Draw arc circle to … and change line width to 2.5
Segment 1 command 1	00	Close path
Segment 1 command 2	04 00 …	Draw arc circle to …
Segment 1 command 3	00	Close path
Segment 2 start	38 FF 90 10	Segment 2 starts at (-50, 100)
Segment 2 command 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 Bézier

The cubic bezier instruction draws a Bézier 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 Bézier 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

Draws an ellipse segment between the current and the target point.

Field	Type	Description
large_arc	u1	If 1, the large portion of the ellipse segment is drawn
sweep	u1	Determines if the ellipse segment is left- or right bending.
padding	u6	Always 0.
radius_x	Unit	The radius of the ellipse in horizontal direction.
radius_y	Unit	The radius of the ellipse in vertical direction.
rotation	Unit	The rotation of the ellipse in mathematical negative direction, in degrees.
target	Point	The end point of ellipse circle segment.

radius_x and radius_y determine the both radii of the ellipse. If the distance between the current point and target is not enough to fit any ellipse segment between the two points, radius_x and radius_y are scaled uniformly so that it fits exactly.

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

If sweep is 1, the ellipse 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.

Close Path

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

Quadratic Bézier

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.

Rendering

This chapter specifies details of the TinyVG rendering so all images will look the same on different platforms.

The following formulas and code examples use color intensity values between 0.0 and 1.0.

Linear Color Space Conversion

Certain rendering options must happen in linear color space. For each color space, a conversion routine is defined in the specification of that color space.

For sRGB, the following routines can be used:

```
RGB toColorSpace(RGB val) {
  return pow(val, 1.0 / 2.2);
}

RGB toLinear(RGB val) {
  return pow(val, 2.2);
}
```

Alpha Blending

Alpha blending describes the process of blending two transparent colors over each other. As TinyVG has a transparent background by default, transparency must be respected when blending colors together.

```
// Blends the src color over the dst color
RGBA blend(RGBA dst, RGBA src) {
  if (src.a == 0) {
      return dst;
  }
  if (src.a == 1.0) {
      return src;
  }

  const alpha = src.a + (1.0 - src.a) * dst.a;

  return RGBA(
      .r = lerpColor(src.r, dst.r, src.a, dst.a, fin_alpha),
      .g = lerpColor(src.g, dst.g, src.a, dst.a, fin_alpha),
      .b = lerpColor(src.b, dst.b, src.a, dst.a, fin_alpha),
      .a = alpha,
  );
}

float lerpColor(float src, float dst, float src_alpha, float dst_alpha, float alpha) {
    const src_val = toLinear(src);
    const dst_val = toLinear(dst);

    const value = (1.0 / alpha) * (src_alpha * src + (1.0 - src_alpha) * dst_alpha * dst);

    return toColorSpace(value);
}
```

Color Interpolation

Color interpolation is needed in gradients and must performed in linear color space. This means that the value from the color table needs to be converted to linear color space, then each color component is interpolated linearly and the final color is then determined by converting the color back to the specified color space.

```
RGBA blend(RGBA first, RGBA second, float f) {
  f = clamp(f, 0.0, 1.0);
  return RGBA {
    .rgb = toColorSpace(
      lerp(toLinear(first.rgb), toLinear(second.rgb), f)
    ),
    .a = lerp(first.a, second.a, f),
  );
}

lerp(a, b, float f) {
  return a + (b-a) * f;
}
```

Line Rendering

Lines are rendered with round line caps and use a total width. Lines in TinyVG can be seen as the Minkowski sum of a sphere with the line width a diameter and the line itself.

Lines that have a width less than a *pixel* on the final display, they should be rendered as exactly on pixel wide. Otherwise lines might get invisible, jagged or otherwise incomplete.

Revision History

1.0

- Initial release